

Queste de savoir

Un zeste de Python

1^{er} octobre 2022

Table des matières

Introduction	14
I. Premiers pas avec Python	17
Introduction	18
I.1. Présentation du langage	19
Introduction	19
I.1.1. Qui es-tu, Python ?	19
I.1.2. Réalisations concrètes	21
I.1.3. Histoire	23
I.1.4. Versions de Python	24
I.2. Installation	26
Introduction	26
I.2.1. Installation sous Windows	26
I.2.1.1. Depuis le Microsoft Store	26
I.2.1.2. Avec l'installateur officiel	26
I.2.1.3. Avec un environnement de développement	28
I.2.2. Installation sous MacOS	28
I.2.3. Installation sous GNU/Linux	29
I.2.3.1. Ubuntu / Debian / Linux Mint	29
I.2.3.2. Fedora	30
I.2.3.3. Archlinux	30
I.2.3.4. Autres distributions	30
I.2.4. Lancement de Python	30
I.2.4.1. IDLE	30
I.2.4.2. Depuis un terminal	31
I.2.5. Éditeur de texte	32
I.2.5.1. Geany	32
I.2.5.2. PyCharm	33
I.2.5.3. Autres éditeurs	34
Conclusion	34
I.3. Prise en main de l'interpréteur	35
Introduction	35
I.3.1. Interpréteur interactif	35
I.3.2. Une machine à calculer	36

I.4.	Écrire du code dans des fichiers	38
	Introduction	38
I.4.1.	Éditer un fichier Python	38
	I.4.1.1. En-têtes de fichier	38
I.4.2.	Exécution du fichier	39
	I.4.2.1. Exécution depuis le terminal	40
I.5.	Fichiers ou interpréteur interactif ?	42
	Introduction	42
I.5.1.	Interpréteur interactif	42
I.5.2.	Écriture dans des fichiers	42
	Conclusion	43
II.	Manipuler des données	44
	Introduction	45
II.1.	Retour sur la calculatrice	46
	Introduction	46
II.1.1.	Des nombres à virgule	46
II.1.2.	Autres opérateurs	47
	II.1.2.1. Priorités des opérateurs	49
II.1.3.	Fonctions	50
II.2.	Des variables pour sauvegarder nos résultats	53
	Introduction	53
II.2.1.	Une étiquette sur une valeur	53
II.2.2.	Assignations	54
II.2.3.	Conventions	56
	II.2.3.1. La variable <code>_</code>	57
II.3.	Manipuler du texte	59
	Introduction	59
II.3.1.	Chaînes de caractères	59
II.3.2.	Opérations sur les chaînes	61
II.4.	Interactions	65
	Introduction	65
II.4.1.	Entrées utilisateur	65
II.4.2.	Conversions de types	66
II.5.	Objets et méthodes	68
	Introduction	68
II.5.1.	Objets	68
II.5.2.	Méthodes	68
II.5.3.	Méthodes des chaînes de caractères	69
	II.5.3.1. <code>strip</code>	69
	II.5.3.2. <code>capitalize</code> et <code>title</code>	69

II.5.3.3.	upper et lower	70
II.5.3.4.	index	70
II.5.4.	Attributs	71
II.6.	TP : Combat en tour par tour	73
	Introduction	73
II.6.1.	Présentation générale du TP	73
II.6.2.	Initialisation du jeu	74
II.6.2.1.	Solution	74
II.6.3.	Tour de jeu	74
II.6.3.1.	Solution	75
	Conclusion	75
	Contenu masqué	76
III.	Des programmes moins déterminés	79
	Introduction	80
III.1.	Les conditions (if/elif/else)	81
	Introduction	81
III.1.1.	Test d'égalité	81
III.1.2.	Bloc conditionnel	82
III.1.2.1.	Interpréteur interactif	84
III.1.2.2.	Blocs sur une ligne	85
III.1.3.	Et sinon ?	85
III.1.4.	Structures multiples	87
III.1.4.1.	Enchaînement	87
III.1.4.2.	Imbrication	88
III.2.	Expressions booléennes	89
	Introduction	89
III.2.1.	Opérations booléennes	89
III.2.1.1.	Vocabulaire	89
III.2.1.2.	Opérateurs	89
III.2.1.3.	Priorités des opérateurs	91
III.2.2.	Booléens	91
III.2.2.1.	Algèbre de Boole	91
III.2.2.2.	Conditions et priorités	93
III.2.2.3.	Conversions implicites et explicites	94
III.2.2.4.	Comparaisons chaînées	95
III.3.	TP : Ajoutons des conditions à notre jeu	96
	Introduction	96
III.3.1.	Proposer plusieurs attaques	96
III.3.2.	Solution	96
	Contenu masqué	97

III.4. Les listes	99
Introduction	99
III.4.1. Des séquences de valeurs	99
III.4.2. Opérations sur les listes	101
III.4.2.1. Opérations élémentaires	101
III.4.2.2. Mutabilité	103
III.4.2.3. Slicing	106
III.4.3. Listes à plusieurs dimensions	108
III.4.3.1. Problème de la multiplication	109
III.5. Itérer sur nos listes avec des boucles for	111
Introduction	111
III.5.1. Le bloc for	111
III.5.2. Itération	112
III.5.2.1. Parcourir des listes	112
III.5.2.2. Itérables	114
III.5.2.3. Construire une liste	116
III.5.2.4. Boucles imbriquées	117
Conclusion	119
III.6. Boucler sur une condition (while)	120
Introduction	120
III.6.1. Boucler sur une condition	120
III.6.2. Vers l'infini et au-delà	121
III.6.3. Boucle for ou boucle while ?	123
III.7. Algorithmes	125
Introduction	125
III.7.1. Minimum d'une liste	125
III.7.2. Tri d'une liste	126
III.7.3. Un peu de dessin	127
III.7.3.1. Affichons un rectangle	127
III.7.3.2. Passons au triangle	128
III.7.3.3. Mon beau sapin, roi des forêts	129
Conclusion	130
Contenu masqué	131
III.8. TP : Ajoutons des boucles à notre jeu	133
Introduction	133
III.8.1. Itérer sur les attaques	133
III.8.1.1. Solution	134
III.8.2. Un jeu au tour par tour	134
III.8.2.1. Solution	134
Contenu masqué	135
IV. Types de données	138
Introduction	139

IV.1. Retour sur les types précédents	140
Introduction	140
IV.1.1. Généralités	140
IV.1.1.1. Conversions	140
IV.1.1.2. Comparaisons	140
IV.1.2. Booléens	141
IV.1.2.1. Conversions	141
IV.1.2.2. Opérations	141
IV.1.3. Nombres	143
IV.1.3.1. Conversions	143
IV.1.3.2. Opérations	143
IV.1.3.3. Représentations des entiers	145
IV.1.3.4. Précision des flottants	146
IV.1.3.5. Notation des flottants	147
IV.1.3.6. Nombres complexes	148
IV.1.4. Chaînes de caractères	149
IV.1.4.1. Conversions	149
IV.1.4.2. Opérations	149
IV.1.4.3. Principales méthodes	151
IV.1.4.4. Méthodes avancées	153
IV.1.5. Listes	154
IV.1.5.1. Conversions	154
IV.1.5.2. Opérations	155
IV.1.5.3. Principales méthodes	158
IV.1.5.4. Identité	160
IV.1.6. None	161
IV.2. Les dictionnaires	162
Introduction	162
IV.2.1. Des tables d'association	162
IV.2.2. Opérations sur les dictionnaires	163
IV.2.2.1. Méthodes principales	164
IV.2.2.2. Conversions	165
IV.2.3. Données composites	166
IV.2.4. Clés de dictionnaires	168
IV.3. Itérer sur un dictionnaire	170
Introduction	170
IV.3.1. Dictionnaire et boucle for	170
IV.3.2. Autres manières d'itérer	171
IV.3.2.1. Conversions	172
IV.4. Les tuples	173
Introduction	173
IV.4.1. Les tuples	173
IV.4.2. Opérations sur les tuples	174
IV.4.3. Utilisations des tuples	176

IV.5. TP	178
Introduction	178
IV.5.1. Structurer les données	178
IV.5.2. Solution	179
Contenu masqué	180
 V. Les fonctions	 183
Introduction	184
V.1. Des fonctions pour factoriser	185
Introduction	185
V.1.1. Don't Repeat Yourself	185
V.1.2. Factoriser	185
V.1.2.1. Identifier les portions logiques dans un code plus complet	186
V.1.3. Définir une fonction (bloc def)	186
V.1.4. Appel de fonction	187
Contenu masqué	188
V.2. Fonctions paramétrées	190
V.2.1. Paramètres de fonction	190
V.2.2. Espace de noms	192
V.2.3. Arguments positionnels et nommés	194
V.3. Retours de fonctions	196
V.3.1. Renvoyer une valeur avec return	196
V.3.2. Plusieurs return dans une fonction	197
V.3.3. Renvoyer plusieurs valeurs	198
V.3.3.1. Unpacking	198
V.4. Paramètres et types mutables	200
V.4.1. Rappel sur les types mutables	200
V.4.2. Paramètres mutables	200
V.4.2.1. Effets de bord	201
V.5. Fonctions de tests	203
V.5.1. Un monde rempli de bugs	203
V.5.2. Fonctions de tests	204
V.5.2.1. Assertions	204
V.5.2.2. Tests unitaires	204
V.6. TP : Intégrons des fonctions à notre application	209
V.6.1. Découpage en fonctions	209
V.6.1.1. Solution	209
V.6.2. Tests	209
V.6.2.1. Solution	210
Contenu masqué	210

VI. Entrées / sorties	213
Introduction	214
VI.1. Découper son code en modules	215
VI.1.1. Factoriser le code	215
VI.1.2. Les modules	215
VI.1.2.1. La fonction <code>help</code>	216
VI.1.3. Imports	217
VI.1.4. Bibliothèque standard	219
VI.1.5. Modules de tests	220
Contenu masqué	221
VI.2. Lire un fichier en Python	224
Introduction	224
VI.2.1. Fichiers et dossiers sur l'ordinateur	224
VI.2.2. Problématique : sauvegarder l'état de notre jeu	225
VI.2.3. Fonction <code>open</code>	225
VI.2.4. Fichiers	226
VI.2.4.1. Lire le contenu d'un fichier	226
VI.2.4.2. Fermer un fichier	227
VI.2.4.3. Bloc <code>with</code>	228
VI.3. Itérer sur un fichier	230
VI.3.1. Méthodes des fichiers	230
VI.3.2. Les fichiers sont itérables	232
Contenu masqué	233
VI.4. Écrire dans un fichier	235
VI.4.1. Écriture	235
VI.4.1.1. Écrire plusieurs lignes dans un fichier	236
VI.4.1.2. La fonction <code>print</code>	238
VI.4.2. Autres modes des fichiers	239
VI.4.2.1. Insérer à la fin du fichier	239
VI.4.2.2. Créer un fichier	240
VI.4.2.3. Lire et écrire à la fois	240
VI.5. Chaînes de formatage	242
VI.5.1. Opérations de formatage	242
VI.5.1.1. Méthode <code>format</code>	242
VI.5.1.2. Options de formatage	243
VI.5.1.3. Operateur <code>%</code>	246
VI.5.2. f-strings	246
VI.6. Gérer les exceptions (try/except)	248
Introduction	248
VI.6.1. Tout ne se passe pas comme prévu	248
VI.6.2. Éviter l'exception	249
VI.6.2.1. Limites	250

VI.6.3.	Traiter l'exception	251
VI.6.3.1.	Attraper plusieurs exceptions	253
VI.6.3.2.	Remontée d'erreurs	255
VI.7.	Formater les données	258
	Introduction	258
VI.7.1.	Format JSON	258
VI.7.1.1.	Module <code>json</code>	259
VI.7.1.2.	Avantages et inconvénients	260
VI.7.2.	Format XML	261
VI.7.2.1.	Module <code>xml</code>	261
VI.7.2.2.	Avantages et inconvénients	265
VI.7.3.	Format INI	265
VI.7.3.1.	Module <code>configparser</code>	266
VI.7.3.2.	Avantages et inconvénients	268
VI.7.4.	Format CSV	268
VI.7.4.1.	Module <code>csv</code>	268
VI.7.4.2.	Avantages et inconvénients	271
VI.7.5.	Chaînes de bytes	271
VI.7.5.1.	Encodages	274
VI.7.5.2.	Mode binaire	276
VI.7.6.	Sérialisation binaire	276
VI.7.6.1.	Module <code>pickle</code>	276
VI.7.6.2.	Avantages et inconvénients	278
VI.7.7.	Autres formats	278
VI.8.	Arguments de la ligne de commande	281
VI.8.1.	Ligne de commande	281
VI.8.1.1.	Sortie standard et sortie d'erreur	282
VI.8.2.	Parseur d'arguments	284
VI.9.	Les paquets	286
	Introduction	286
VI.9.1.	Construction d'un paquet	286
VI.9.1.1.	Imports relatifs	287
VI.9.2.	Fichier <code>init.py</code>	288
VI.9.3.	Fichier <code>main.py</code>	289
	Conclusion	290
VI.10.	TP : Sauvegarder la partie	291
VI.10.1.	Découpage en modules	291
VI.10.1.1.	Solution	291
VI.10.2.	Sauvegarde	291
VI.10.2.1.	Solution	293
VI.10.3.	Tests	293
VI.10.3.1.	Solution	293
	Contenu masqué	293

VII. Aller plus loin	303
Introduction	304
VII.1. Les autres types de données	305
Introduction	305
VII.1.1. Les ensembles	305
VII.1.1.1. Opérations	306
VII.1.1.2. Méthodes	308
VII.1.1.3. <code>frozenset</code>	309
VII.1.2. Module <code>collections</code>	310
VII.1.2.1. <code>Counter</code>	310
VII.1.2.2. <code>defaultdict</code>	313
VII.1.2.3. <code>OrderedDict</code>	314
VII.1.2.4. <code>ChainMap</code>	315
VII.1.2.5. <code>deque</code>	318
VII.1.2.6. <code>namedtuple</code>	320
VII.1.3. Types	322
VII.2. Retour sur les conditions	325
VII.2.1. Instructions et expressions	325
VII.2.2. Expressions conditionnelles	326
VII.3. Retour sur les boucles	328
VII.3.1. Cas des boucles infinies	328
VII.3.2. Contrôle du flux	328
VII.3.3. Outils	331
VII.3.3.1. Fonctions natives (<i>builtins</i>)	332
VII.3.3.2. <code>reversed</code>	333
VII.3.3.3. <code>sorted</code>	333
VII.3.3.4. <code>min</code> et <code>max</code>	335
VII.3.3.5. <code>zip</code>	336
VII.3.3.6. Module <code>itertools</code>	336
VII.3.4. Listes en intension	338
VII.3.4.1. Conditions de filtrage	339
VII.3.4.2. Boucles imbriquées	341
VII.3.4.3. Autres constructions en intension	342
VII.3.5. Itérateurs	342
VII.3.5.1. Itérables et itérateurs	342
VII.3.5.2. Fonctions <code>map</code> et <code>filter</code>	345
VII.3.5.3. Itérateurs infinis	346
VII.3.5.4. Fonction <code>iter</code>	348
VII.4. Retour sur les fonctions	350
VII.4.1. Arguments optionnels	350
VII.4.1.1. Paramètres par défaut mutables	351
VII.4.1.2. Ordre de placement des paramètres	352
VII.4.2. Arguments variadiques	352
VII.4.2.1. Opérateur <i>splat</i>	354

VII.4.3. Documentation et annotations	355
VII.4.3.1. Docstring	356
VII.4.3.2. Comment documenter ?	357
VII.4.3.3. Annotations de types	357
VII.4.3.4. Module <code>typing</code>	358
VII.4.4. Décorateurs	360
VII.4.4.1. Décorateur paramétré	361
VII.4.5. Fonctions lambdas	363
VII.4.6. Fonctions récursives	364
VII.4.6.1. Récursion infinie	366
VII.4.6.2. Récursions croisées	366
VII.5. Retour sur les variables	368
VII.5.1. Expressions d'assignation	368
VII.5.2. Annotations de types	370
VII.5.3. Scopes	371
VII.5.3.1. Variables globales	372
VII.5.3.2. Fonctions imbriquées	373
VII.5.3.3. Variables non-locales	374
VII.6. Retour sur les exceptions	375
VII.6.1. Bloc <code>except</code>	375
VII.6.1.1. Données complémentaires des exceptions	376
VII.6.2. Autres mots-clés	377
VII.6.2.1. <code>else</code>	377
VII.6.2.2. <code>finally</code>	378
VII.6.3. Bloc <code>with</code>	380
VII.6.3.1. Supprimer une exception	381
VII.6.4. Lever une exception	382
VII.6.4.1. Hiérarchie des exceptions	383
VII.7. Débogage	385
Introduction	385
VII.7.1. Programme de référence	385
VII.7.2. Introspection	388
VII.7.2.1. Informations sur la valeur	388
VII.7.2.2. Contenu de la valeur	390
VII.7.3. Déboguer «à la main»	391
VII.7.3.1. Suivre et comprendre les exceptions	391
VII.7.3.2. S'appuyer sur des tests unitaires	394
VII.7.4. Utilisation d'un débogueur (Pdb)	396
VII.7.4.1. Lancer un programme pas-à-pas avec Pdb	397
VII.7.4.2. Invoquer Pdb depuis le programme	402
VII.7.5. Déboguer avec votre IDE	404
VIII. La bibliothèque standard	406
Introduction	407

VIII.1. Tour d’horizon de la bibliothèque standard	408
Introduction	408
VIII.1.1. Fonctions natives	408
VIII.1.1.1. Module <code>operator</code>	410
VIII.1.2. Gestion des nombres	412
VIII.1.2.1. Nombres décimaux	412
VIII.1.2.2. Nombres rationnels	414
VIII.1.2.3. Hiérarchie des nombres	415
VIII.1.2.4. Bibliothèques mathématiques	416
VIII.1.3. Chemins et fichiers	417
VIII.1.3.1. Usage des chemins	418
VIII.1.3.2. Propriétés des chemins	419
VIII.1.3.3. Méthodes concrètes	421
VIII.1.3.4. Méthodes pour les répertoires	423
VIII.1.3.5. Méthodes pour les fichiers	424
VIII.1.4. Modules systèmes	426
VIII.1.4.1. Module <code>sys</code>	426
VIII.1.4.2. Module <code>shutil</code>	428
VIII.1.4.3. Module <code>os</code>	430
VIII.2. Un peu d’aléatoire	433
Introduction	433
VIII.2.1. Le module <code>random</code>	433
VIII.2.1.1. Nombres aléatoires	433
VIII.2.1.2. Opérations aléatoires	434
VIII.2.2. Distributions	436
VIII.2.2.1. Lois de distribution	436
VIII.2.2.2. Pondération	438
VIII.3. Gestion du temps	442
Introduction	442
VIII.3.1. Module <code>time</code>	442
VIII.3.1.1. Les timestamps	442
VIII.3.1.2. Structure de temps	443
VIII.3.1.3. Utilitaires du module	444
VIII.3.2. Module <code>datetime</code>	445
VIII.3.2.1. Conversions	445
VIII.3.2.2. Durées	447
VIII.3.2.3. Fuseaux horaires	448
VIII.3.3. Module <code>calendar</code>	449
Conclusion	450
VIII.4. Expressions rationnelles	451
Introduction	451
VIII.4.1. Problématique	451
VIII.4.2. Une histoire d’automates	453
VIII.4.3. Module <code>re</code>	453
VIII.4.3.1. Utilisation	454
VIII.4.3.2. <code>is_number</code>	456

VIII.4.3.3. Autres fonctions du module	457
VIII.4.4. Syntaxe des regex	460
VIII.4.4.1. Chaînes brutes (<i>raw strings</i>)	460
VIII.4.4.2. Syntaxe des motifs	461
VIII.4.4.3. Options	466
VIII.4.5. Limitations	468
Conclusion	469
VIII.5. TP : Monstre sauvage	470
Introduction	470
VIII.5.1. L'aléatoire à la rescousse !	470
VIII.5.1.1. Solution	470
VIII.5.2. Animations	471
VIII.5.2.1. Solution	472
Contenu masqué	472
VIII.6. Installer des modules complémentaires	478
Introduction	478
VIII.6.1. Installation	478
VIII.6.2. Pip, le gestionnaire de paquets Python	478
VIII.6.3. Environnements virtuels	481
IX. Annexes	483
Introduction	484
IX.1. Glossaire et mots clés	485
IX.1.1. Glossaire	485
IX.1.2. Tableau des mots-clés	494
IX.1.3. Tableau des opérateurs	500
IX.1.3.1. Opérateurs simples (expressions)	500
IX.1.3.2. Opérateurs d'assignation	504
IX.1.4. Priorité des opérateurs	505
IX.1.4.1. Ordre d'évaluation des expressions	505
IX.1.4.2. Tableau des priorités	505
IX.1.5. Autres éléments de syntaxe	509
IX.2. Notes diverses	511
Introduction	511
IX.2.1. En-têtes de fichiers	511
IX.3. Quelques modules complémentaires bien utiles	513
Introduction	513
IX.3.1. Requests	513
IX.3.2. Numpy	514
IX.3.3. Django	514
IX.3.4. Pillow	515
IX.3.5. PyGObject (PyGTK)	515

IX.3.6.	Pygame	517
IX.4.	Tests	519
	Introduction	519
IX.4.1.	Pytest	519
IX.4.2.	Unittest	521
IX.5.	Outils	523
	Introduction	523
IX.5.1.	Linters	523
IX.5.1.1.	Flake8	523
IX.5.1.2.	Pylint	523
IX.5.1.3.	Black	523
IX.5.1.4.	isort	524
IX.5.2.	mypy	524
IX.6.	Ressources	525
	Introduction	525
IX.6.1.	Liens utiles	525
IX.6.2.	Cours	525
IX.6.2.1.	Sur Zeste de Savoir	525
IX.6.2.2.	Ailleurs sur le web	525
IX.6.3.	Exercices	526
IX.6.4.	Discussions	526
IX.6.4.1.	Forums	526
IX.6.4.2.	Salons de discussions	526
IX.6.5.	Conférences	526
	Conclusion	527

Introduction

Introduction

Bien le bonjour ! Bienvenue dans le monde magique de Python !

À toi qui t'intéresses à la programmation informatique, au développement logiciel et à son apprentissage : voici un tutoriel pour te guider et te permettre d'avancer dans ta quête.

Ce cours a pour but de t'apprendre à parler le Python. Il s'agit d'un langage particulier—un langage de programmation—pour communiquer avec ton ordinateur afin de lui demander de réaliser des tâches précises (comme exécuter un calcul, récupérer des événements du clavier, afficher une image à l'écran, etc.), c'est-à-dire exécuter un programme informatique (un logiciel).

Il existe une multitude de langages de programmation (tels que le C, le Java, le PHP ou le Javascript), j'ai choisi le Python pour ce cours car il me semble être le langage idéal pour débiter puis continuer la programmation.

Peut-être connais-tu déjà l'un ou l'autre de ces langages, ou même que tu programmes au quotidien : ce cours s'adresse tout de même à toi, il vise à apprendre le Python quelque soit ton niveau de départ.

Ta quête du Python est sur le point de commencer ! Un tout nouveau monde de rêves, d'aventures et de programmation t'attend ! Dingue !

Pour nous rejoindre, rien de plus simple, rends-toi de ce pas vers [le premier chapitre](#) afin d'en découvrir plus sur le langage.



FIGURE .0.1. – Un zeste de Python.

J'ai découpé ce cours en plusieurs parties (ou paliers) pour te permettre d'avancer pas à pas dans l'apprentissage du Python, en commençant par les bases pour à terme réussir à réaliser des programmes complets.

Chaque partie se divise en chapitres, pour présenter les différentes notions du langage, avec des exercices pour les mettre en pratique.

Les parties sont généralement conclues par un chapitre de travaux pratiques (TP), pour utiliser concrètement les connaissances apprises au long des chapitres qui précèdent, dans le but de réaliser un jeu de combat au tour par tour en mode texte.

i

Tu n'as pas besoin de connaissances particulières pour démarrer ce tutoriel : savoir installer un logiciel sur ton ordinateur et connaître les bases du calcul (opérations élémentaires) sont les seuls pré-requis.

Quelques notions d'anglais sont un plus.

La difficulté augmente bien sûr au long du cours, mais j'ai fait en sorte que l'avancée soit progressive et donc que cette difficulté soit transparente. N'hésite pas à demander de l'aide si toutefois tu buttais sur un point ou étais bloqué sur un exercice.

Je suis aussi ouvert à tous retours si tu trouves que telle notion est mal expliquée ou que telle autre mériterait d'être abordée.

Car oui, ce cours ne couvre pas l'entièreté du langage Python et n'a pas pour volonté de remplacer [sa documentation](#) [↗](#). Il a simplement pour but d'apprendre les bases du langage et d'écrire des programmes complets en exploitant ces bases.

Mais ce tuto ne traite pas de la programmation objet en Python ou des notions avancées du langage, sujets pour lesquels j'ai écrit deux autres cours afin de compléter ton apprentissage :

- [La programmation orientée objet en Python](#) [↗](#).
- [Notions de Python avancées](#) [↗](#).

Je t'ai donné soif d'apprendre ? Alors installe-toi, prends un smoothie, et n'hésite pas à y ajouter un zeste de Python.

Première partie

Premiers pas avec Python

Introduction

Cette première partie a pour but d'entrer en contact avec Python pour être en mesure d'exécuter vos premières commandes.

I.1. Présentation du langage

Introduction

Avant d'entrer dans le dur du sujet, j'aimerais déjà que vous fassiez connaissance et je vais donc vous parler un peu de Python.

Vous décrire ses caractéristiques et son histoire, vous montrer de quoi il est capable.

I.1.1. Qui es-tu, Python ?



FIGURE I.1.1. – Logo de Python

Comme je le disais en introduction, Python est ce que l'on appelle un langage de programmation. L'ordinateur ne parlant pas notre langue, il est nécessaire d'adopter la sienne pour parler avec lui. Mais celle-ci est très rudimentaire, c'est le langage machine compris par le processeur.

Avec le temps, d'autres langages sont apparus autour pour pouvoir communiquer avec l'ordinateur plus simplement que par des instructions processeurs, les langages de programmation.

Je ne pourrais pas vous en faire de liste exhaustive tellement ils sont nombreux, même Wikipédia [a du mal à le faire](#) .

I. Premiers pas avec Python

Python est un langage dit de haut-niveau, c'est-à-dire qu'il s'éloigne du langage machine en ajoutant des concepts et des outils le rendant plus facile à lire et à écrire, plus proche du langage humain (de l'anglais en l'occurrence).

Python est aussi un langage portable : en dehors de certains cas exceptionnels, un programme Python peut être exécuté de la même manière sur un ordinateur Windows, Mac OS ou GNU/Linux, ainsi que sur des OS mobiles comme Android ou iOS.

Voici un exemple de code écrit en Python :

```
1 def hello(name=None):
2     if name is None:
3         print('Hello World!')
4     else:
5         print('Hello', name)
6
7 hello('Clem')
8 hello()
```

Listing 1 – Un programme tout simple pour dire bonjour en Python

i

Comme vous pouvez le remarquer dans cet exemple, plusieurs lignes commencent par des espaces.

C'est ce qui permet en Python de séparer les différentes portions de code tout en l'aérant, on appelle cela l'**indentation**, mais on y reviendra. 🍊

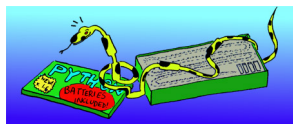


FIGURE I.1.2. – «Piles incluses»—Crédits : *Frank Stajano* [↗](#)

Python est parfois qualifié de «piles incluses», par rapport au fait qu'il est fourni de base avec beaucoup de fonctionnalités (création de fenêtres, gestion native de différents formats de fichiers, etc.).

Mais en plus de ça il dispose d'une large communauté de développeurs et développeuses, contribuant à élargir l'écosystème Python en développant un grand nombre de nouveaux outils pour répondre à des tâches particulières, que vous pourrez réutiliser à votre tour dans vos logiciels.

Cette forte communauté fournit aussi à Python une documentation très complète, qui est de plus traduite en français.

De même, vous trouverez facilement quelqu'un dans la communauté Python pour vous aider dans votre développement, en vous orientant vers des forums de discussion tels que celui de Zeste de Savoir.

On dit aussi de Python qu'il est un langage orienté objet.

Cela signifie que les valeurs que l'on y manipule sont des objets : ils ont des propriétés et des actions qui leur sont propres, ils interagissent les uns avec les autres.

I. Premiers pas avec Python

C'est une technologie libre d'utilisation et gratuite. Vous pouvez utiliser Python dans vos programmes comme vous le voulez, distribuer ou vendre ces programmes. Cela signifie aussi que vous pouvez vous-même contribuer au code de Python.

Python se démarque par sa lisibilité, mais cela représente un coût en performances car il faut dans tous les cas que le code soit converti en langage machine.

Cela ne devrait pour autant pas trop vous limiter dans ce que vous pourrez faire avec lui, comme je le montre dans la section qui suit.

De plus c'est un langage extensible, il vous sera donc toujours possible de réaliser du code dans un autre langage dont vous tireriez d'autres avantages puis de le brancher à du code Python.

I.1.2. Réalisations concrètes

La première question que vous pourriez vous poser à propos de Python serait : «Qu'est-ce qu'il est possible de faire avec ?». À laquelle je pourrais répondre «à peu près tout» mais ce ne serait pas très précis.

Python est utilisé dans de nombreux domaines, à commencer par les sites web.

Le cœur d'Instagram par exemple est entièrement écrit en Python¹, de même qu'une bonne partie de Spotify². Youtube a démarré avec Python et l'utilise encore aujourd'hui pour de nombreuses tâches, il représente aussi une part importante chez Google³.

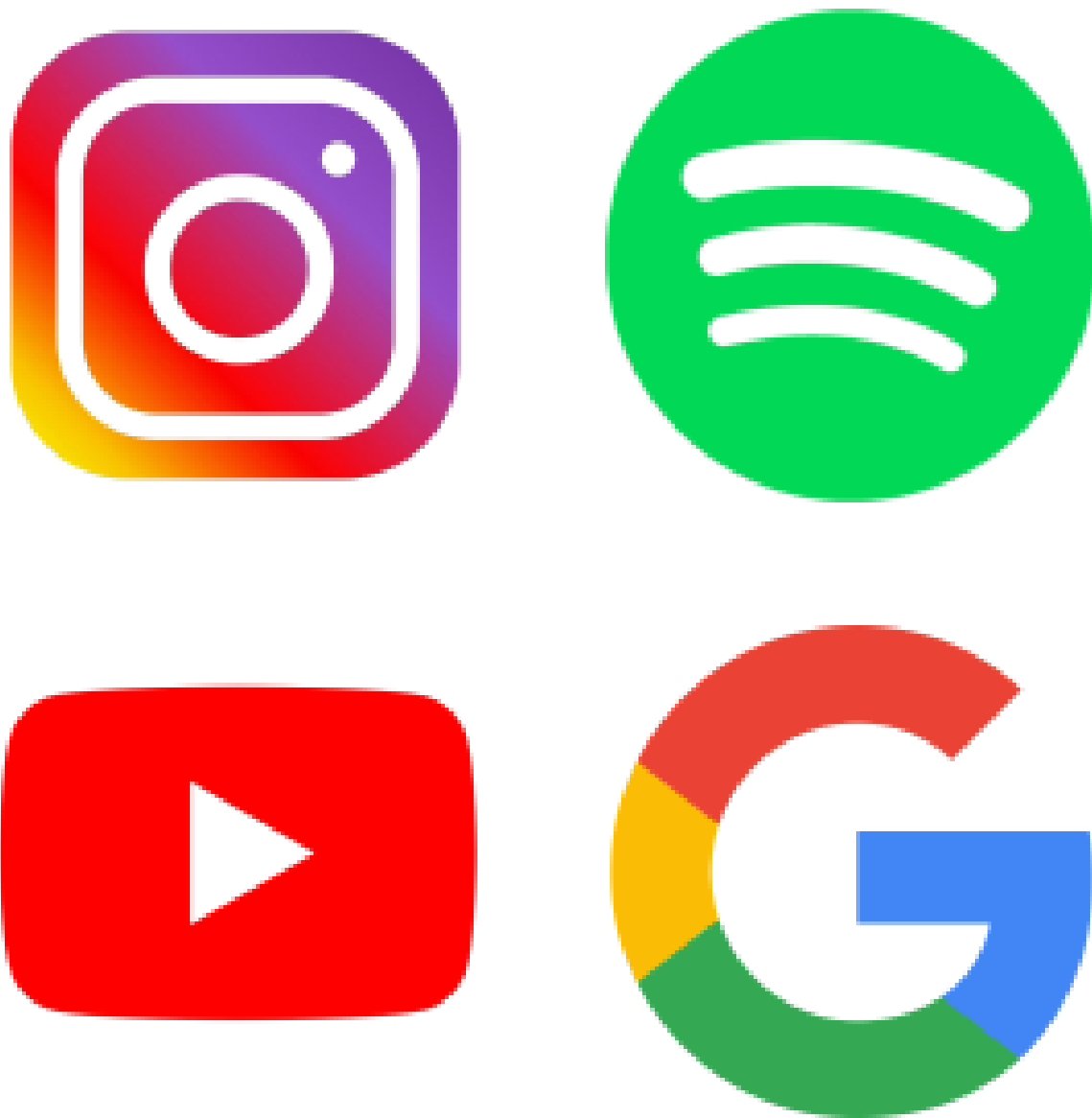


FIGURE I.1.3. – Quelques projets utilisant Python.

Zeste de Savoir lui-même tourne grâce à Python : avec Django plus précisément, un outil Python très répandu pour écrire des sites Internet.

Python est aussi très représenté dans le monde scientifique, avec des applications dans l'intelligence artificielle et le *machine learning*. Il a par exemple été impliqué dans la photographie du trou noir M87 en 2019⁴ ou le vol de l'hélicoptère Ingenuity sur Mars en 2021⁵.



FIGURE I.1.4. – Trou noir M87 (*crédits : Event Horizon Telescope Collaboration*)

Le langage est aussi beaucoup utilisé dans le développement lui-même, pour la réalisation de scripts : des petits programmes rapides pour réaliser des tâches précises.

Enfin, on le retrouve aussi dans différents logiciels, que ce soit dans le cœur du programme ou dans des modules annexes.

Le logiciel de synchronisation de fichiers Dropbox est par exemple écrit en Python. Le langage occupe aussi une place importante dans les jeux Civilization IV et Eve online³. Il intervient encore dans le moteur d’animation du logiciel de modélisation Blender, et sous forme de modules complémentaires dans le logiciel de dessin Inkscape.

I.1.3. Histoire

Place maintenant à un petit historique du langage Python, pour mieux le situer et le comprendre.

Python est né en 1991 aux Pays-Bas dans sa version 0.9.0, conçu par Guido van Rossum, développeur dans un centre de recherche néerlandais (CWI) qui commença à développer le projet à la fin des années 1980. Il s’inspira principalement du langage de programmation [ABC](#) [↗](#) qui était développé au sein de ce centre mais aussi du [langage C](#) [↗](#) et de [Modula-3](#) [↗](#).

-
1. <https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c> [↗](#)
 2. <https://engineering.atspotify.com/2013/03/20/how-we-use-python-at-spotify/> [↗](#)
 3. <https://www.python.org/about/quotes/> [↗](#)
 4. <https://numpy.org/case-studies/blackhole-image/> [↗](#)
 5. <https://github.com/readme/nasa-ingenuity-helicopter> [↗](#)



FIGURE I.1.5. – Guido van Rossum, créateur de Python—[Photo de Dan Stroud](#) .

Python tire son nom de la troupe comique des Monty Python, dont était fan Guido. On trouve d'ailleurs de nombreuses références à cette troupe dans la documentation du langage, avec des exemples usant des noms `spam` et `eggs`.

La version 1.0 de Python arriva en 1994 et ressemblait déjà beaucoup au Python d'aujourd'hui que je vais vous enseigner dans ce cours.

Suivront les versions majeures 2.0 en 2000 et 3.0 en 2008, avec bien sûr de nombreuses versions mineures intermédiaires. Elles ajouteront à Python de nouvelles influences, telles que les langages de programmation [Haskell](#) ou [SETL](#) .

La version 3 fit grand bruit puisqu'elle choisit de casser la rétro-compatibilité pour corriger des erreurs de conception des versions précédentes, il fallut une dizaine d'années pour achever la transition de Python 2 vers Python 3.

En 2001 est créée la *Python Software Foundation* (ou *PSF*) dans le but de promouvoir et de protéger le langage, supervisant son développement et récoltant les fonds pour son financement. La fondation œuvre aussi pour organiser des conférences tout autour du monde à propos de Python, les *PyCon*. On trouve par exemple la [PyConFr](#) organisée chaque année par l'*AFPy*, relai francophone de la PSF.

À la tête du développement du projet depuis sa création en tant que *BDFL* (ou «Dictateur bienveillant à vie»), Guido van Rossum s'en retire en 2018 pour des «vacances permanentes», laissant place à un comité de direction. Il reste depuis très impliqué dans les évolutions du langage.

I.1.4. Versions de Python

Ce cours est destiné à un apprentissage de Python 3. Une version au moins supérieure à 3.8 est recommandée, les précédentes n'étant plus officiellement supportées (ou seulement partiellement) par les développeurs de Python.

Une version non supportée est susceptible de comporter des bugs ou des problèmes de sécurité qui ne seront pas corrigés, il est donc toujours préférable de se maintenir à jour et d'utiliser une version récente de Python. Cela permet de plus de profiter des dernières nouveautés du langage, car Python est un langage qui évolue sans cesse et dont les mises à jour sont intégrées aux nouvelles versions.

I. Premiers pas avec Python

Ainsi, faite votre choix selon les versions proposées par votre système d'exploitation, mais sachez que je ne garantis pas le bon fonctionnement des exemples du tutoriel pour les versions antérieures à la 3.6.

Python suit depuis 2019 un cycle annuel de sortie, chaque nouvelle version paraissant au mois d'octobre. Le calendrier de support des versions de Python est disponible sur [la page de téléchargements du site officiel](#) [↗](#).

Même si la question du choix de version est aujourd'hui simple, elle s'est longtemps posée en raison de la coexistence entre Python 2 et Python 3. La version 2.7 a finalement arrêté d'être supportée au 1er janvier 2020.

Cependant, il est à noter qu'il se peut que vous rencontriez sur le net des exemples de code utilisant Python 2 et qui ne seraient alors pas compatibles avec votre version.

I.2. Installation

Introduction

Avant de pouvoir commencer à programmer avec Python, il nous faut faire comprendre ce langage à notre ordinateur. Et pour cela, nous avons besoin d'installer un logiciel qui lui servira d'interprète. Ce logiciel répond simplement au nom de Python.

Je ne vais pas revenir sur l'historique des versions de Python : à moins d'une contrainte imposée par votre système d'exploitation ou votre gestionnaire de paquets, préférez la dernière version 3.x.y en date.

i

Il existe en fait plusieurs implémentations du langage Python qui sont par exemple CPython, Jython ou encore pypy. Leurs différences se situent sur l'outillage mis en place pour faire comprendre le code Python à l'ordinateur, et non sur le langage en lui-même. Nous resterons ici sur l'implémentation officielle de Python, CPython.

I.2.1. Installation sous Windows

I.2.1.1. Depuis le Microsoft Store

Le plus simple si vous utilisez Windows 10 ou plus est de vous orienter vers le *Microsoft Store*, le gestionnaire de paquets officiel sous Windows.

Vous pourrez y trouver la dernière version du paquet «Python», publié par la *Python Software Foundation*. Ce paquet se chargera d'installer les programmes Python et IDLE (je reviendrai sur lui par la suite) qui seront tous deux présents dans le menu «Démarrer».

!

Le paquet Python publié par la *PSF* est entièrement gratuit. Si vous êtes invité à payer c'est que vous avez sélectionné un paquet frauduleux ne provenant pas de la *PSF*.

i

Le *Microsoft Store* impose certaines restrictions sur les paquets qui y sont publiés. Ainsi, Python installé de cette manière ne vous permettra pas de réaliser des programmes utilisant les emplacements partagés ou le registre.

Ces fonctionnalités ne sont pas abordées dans ce tutoriel et donc ne devraient pas poser problème, mais reportez-vous à l'installateur officiel si vous avez un tel besoin.

I.2.1.2. Avec l'installateur officiel

Pour l'installation classique sous Windows, il vous suffit de vous rendre sur le site officiel de Python dans la section «Downloads > Windows » : <https://www.python.org/downloads/win->

dows/ ↗ .

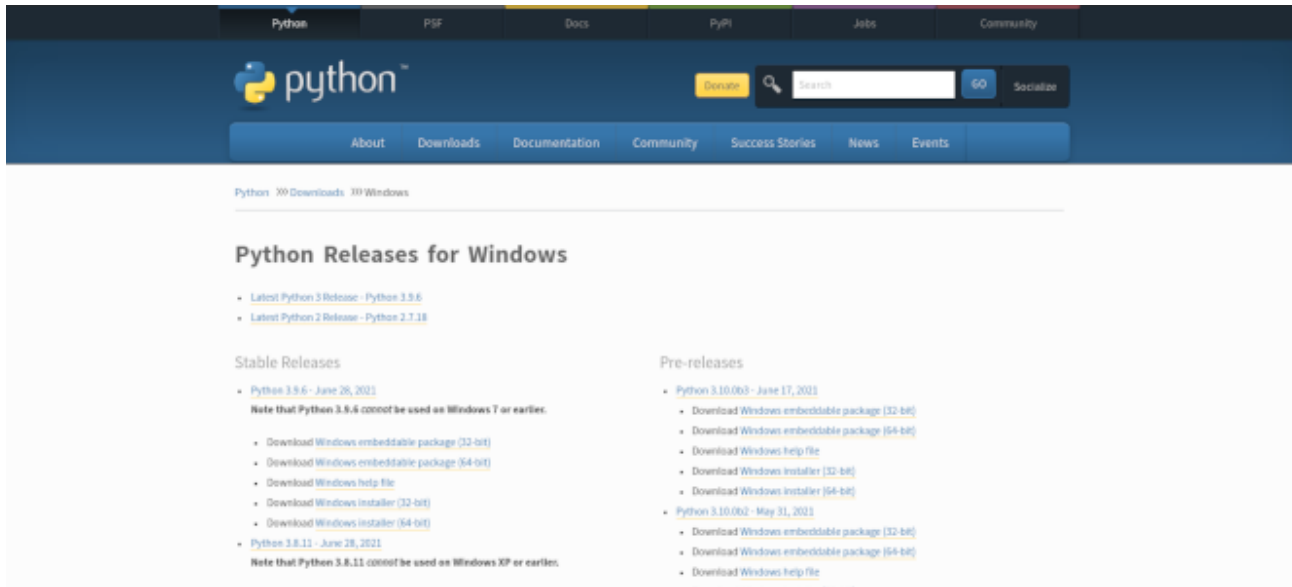


FIGURE I.2.1. – Page des téléchargements.

Là, vous pouvez sélectionner le lien «Download Windows installer (64-bit)» de la version stable (*Stable Release*) la plus récente¹. Téléchargez le fichier `.exe` pointé et exécutez-le.



Téléchargez toujours l'installateur depuis le site officiel de Python plutôt qu'une autre source, afin d'éviter tout programme frauduleux.

1. Ces informations sont données pour le cas général. Dans d'autres cas précis (version ancienne de Windows, système 32 bits), référez-vous aux recommandations données par la page de téléchargement.

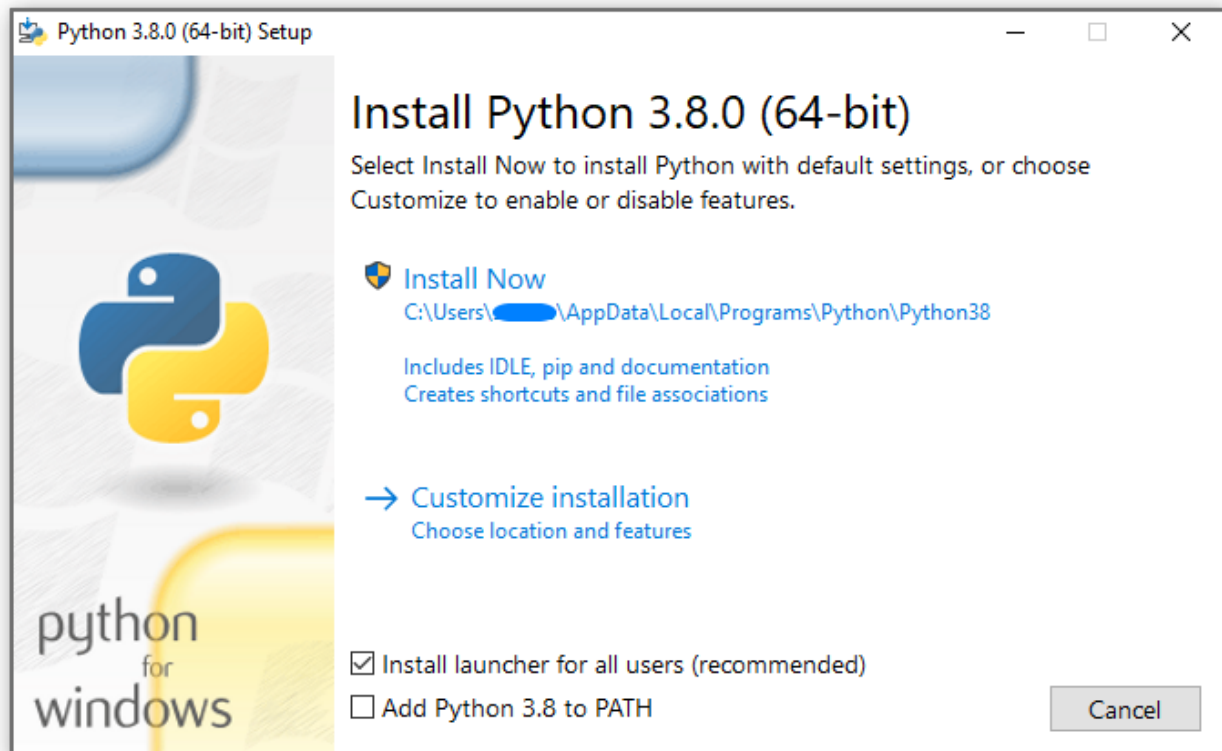


FIGURE I.2.2. – Installation de Python

Cochez alors les cases «Install launcher for all users» (installer pour tous les utilisateurs) et «Add Python to PATH» (ajouter Python aux chemins système) pour simplifier les utilisations futures. Cliquez ensuite sur «Install now» (installer maintenant) afin de procéder à l’installation de Python sur votre système.

I.2.1.3. Avec un environnement de développement

Pour des utilisations spécifiques (telles que la programmation scientifique), il peut être utile de préférer l’installation d’un environnement de développement dédié. Je vous redirige pour cela vers le tutoriel [Installer un environnement de développement python avec conda](#) de @Gabbro.

I.2.2. Installation sous MacOS

Python est souvent déjà installé sur MacOS mais en version 2.7. Or c’est de la version 3 dont nous avons besoin pour ce cours.

Pour installer Python dans sa dernière version, rendez-vous sur la page <https://www.python.org/downloads/mac-osx/> et sélectionnez le lien «universal2 installer» de la version la plus récente.

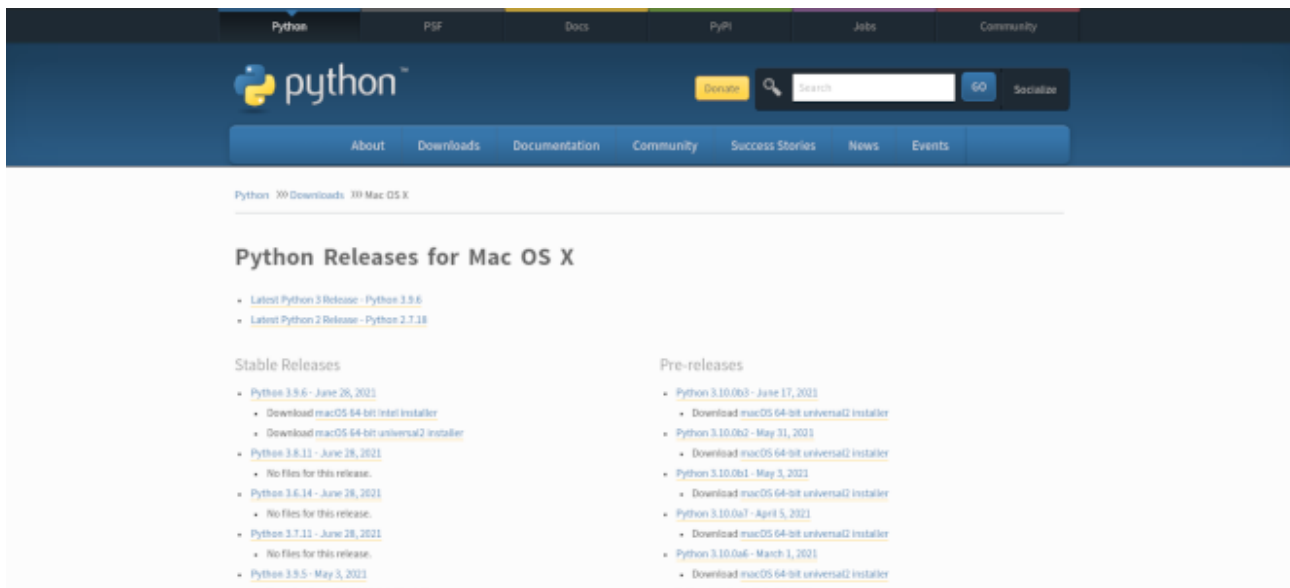


FIGURE I.2.3. – Page des téléchargements.

Après exécution de cet installateur, vous devriez trouver Python dans les applications de l'ordinateur.

I.2.3. Installation sous GNU/Linux

Si vous utilisez Linux, il est probable que Python soit déjà installé sur votre système, car il est nécessaire à certains outils d'administration. Mais il se peut qu'il ne soit pas installé dans la bonne version.

Pour savoir s'il est déjà installé, vous pouvez entrer la commande `python -V` ou `python3 -V` dans un terminal et vous assurer de voir apparaître un message `Python 3.x.y`

I.2.3.1. Ubuntu / Debian / Linux Mint

Sur la distribution Debian et ses dérivées, le gestionnaire de paquets est `apt` et le paquet dédié à Python 3 se nomme `python3`. Il vous faudra les droits administrateur pour l'installer au niveau du système.

Si vous utilisez un gestionnaire de paquets graphique, il vous suffit de faire une recherche sur `python3` et d'installer le paquet correspondant.

Sinon cela peut aussi se faire à l'aide des commandes suivantes.

```
1 sudo apt update
2 sudo apt install python3
```

La première commande ne sert qu'à mettre à jour l'index des dépôts pour éviter toute surprise.



Sur ces distributions, il faudra utiliser la commande `python3` plutôt que simplement `python` pour invoquer la version 3 de Python.

I.2.3.2. Fedora

Sur Fedora, vous pouvez installer le paquet `python` à l'aide du gestionnaire `dnf`. Vous aurez pour cela besoin des droits administrateurs.

```
1 dnf install python
```

I.2.3.3. Archlinux

Sur Archlinux, `python` est présent dans les dépôts du gestionnaire de paquets `pacman`. Vous pouvez l'installer avec la commande suivante, qui nécessite les droits administrateurs.

```
1 pacman -Sy python
```

I.2.3.4. Autres distributions

Pour les autres distributions Linux, vous pouvez vous référer à votre gestionnaire de paquets pour trouver un paquet `python` ou `python3`.

Dans le cas échéant, il est toujours possible de télécharger Python depuis ses sources [sur la page des téléchargements](#) : sélectionnez la version voulue puis cliquez sur le lien «Download», vous pouvez alors télécharger le fichier «Gzipped source tarball» au format gzip. Il faudra ensuite suivre la procédure pour compiler Python : <https://docs.python.org/fr/3/using/unix.html#building-python> .

I.2.4. Lancement de Python

I.2.4.1. IDLE

Suite à cette installation, vous devriez trouver un programme intitulé «IDLE» dans les applications installées sur votre ordinateur. Vous pouvez le trouver dans le menu Démarrer, le dossier Applications, ou autres selon votre système d'exploitation.

IDLE c'est l'environnement de développement fourni par défaut avec Python (*Interactive DeveLopment Environment*). Il va nous permettre d'exécuter facilement du code Python quelque soit votre système.

La fenêtre qui s'ouvre devrait ressembler à cela :

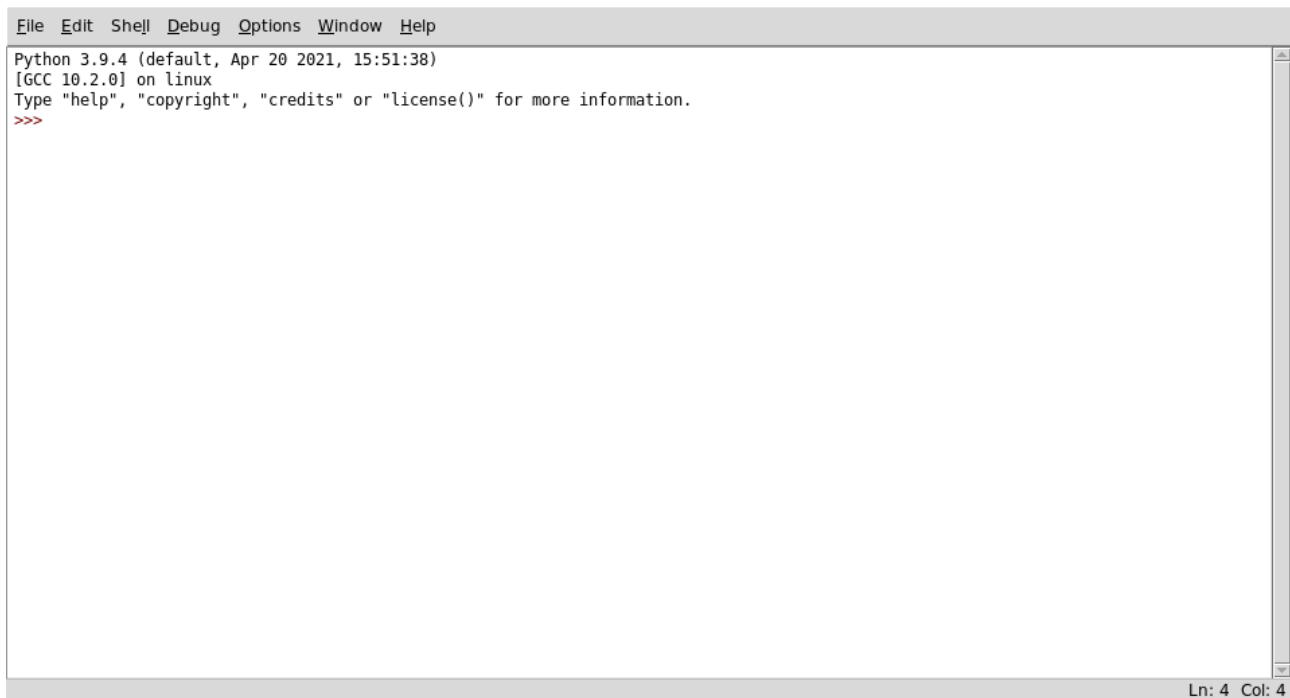


FIGURE I.2.4. – IDLE

C'est un peu austère, mais vous voilà dans un interpréteur Python ! Il n'attend que votre code.

Vérifiez bien que la version qui s'affiche est une `3.x.x`, sinon c'est que vous n'avez pas installé la bonne version et je vous renvoie aux chapitres précédents.

Le «i» de IDLE est pour «interactif», c'est-à-dire qu'il est possible d'y exécuter directement du code et d'en obtenir le résultat. Nous reviendrons dessus par la suite, mais cet interpréteur interactif nous sera très utile pour tester nos premiers codes.

Les possibilités d'IDLE ne s'arrêtent pas là et il nous sera aussi possible d'exécuter du code Python depuis un fichier.

I.2.4.2. Depuis un terminal

Si vous êtes un peu plus à l'aise avec votre système d'exploitation, vous pouvez aussi choisir d'exécuter directement Python depuis un terminal.

Le terminal varie selon votre système, il s'appellera par exemple Powershell sur Windows et Terminal sur Mac. C'est une invite de commandes qui est propre à cet OS et qui permet d'interagir avec lui.

Tout se passe en mode texte : on écrit des commandes, elles sont exécutées par le système et il nous affiche leur résultat dans la fenêtre.

Pour accéder à Python depuis un terminal, il nous suffit alors d'y exécuter la commande `python` (ou `python3` selon la version par défaut) :

```
1 % python
2 Python 3.9.4 (default, Apr 20 2021, 15:51:38)
3 [GCC 10.2.0] on linux
4 Type "help", "copyright", "credits" or "license" for more
   information.
```

```
5 >>>
```

1.2.5. Éditeur de texte

Un éditeur de texte est un programme tel que le bloc-notes qui nous permet de modifier des fichiers texte, ici nos fichiers de code Python.

Même si IDLE peut faire office d'éditeur de texte, il n'est pas le plus pratique pour ça. Il offre en effet de la coloration syntaxique sur le code (ce qui permet de mettre en évidence les mots-clés et structures du langage) mais est assez rudimentaire sur le reste.

D'autres éditeurs vont fournir des fonctionnalités supplémentaires utiles pour travailler sur vos projets, comme le fait de fournir des onglets pour naviguer entre les fichiers.

Ainsi, je vous conseille d'installer un autre éditeur pour être plus à l'aise avec votre code Python.

Mais je fais face à une question houleuse, le choix d'un éditeur de texte s'apparentant parfois à une guerre de religions. Il existe plein d'éditeurs, il y en a pour tous les goûts : des plus ou moins complets, plus ou moins légers, plus ou moins faciles, etc.

J'utilise personnellement emacs mais je ne l'impose à personne.

1.2.5.1. Geany

Ici, j'aimerais vous recommander Geany, c'est un éditeur simple à prendre en main avec quelques fonctionnalités pratiques comme le fait d'intégrer directement un terminal dans la fenêtre pour y exécuter vos programmes.

Il vous permettra aisément de vous y retrouver entre les différents fichiers de votre projet.

Geany peut-être téléchargé via son [site officiel](#) pour Windows ou Mac, et pourra être trouvé dans votre gestionnaire de logiciels sous Linux (paquets `geany` et `geany-plugins`). C'est un logiciel libre, entièrement gratuit et sans publicité.

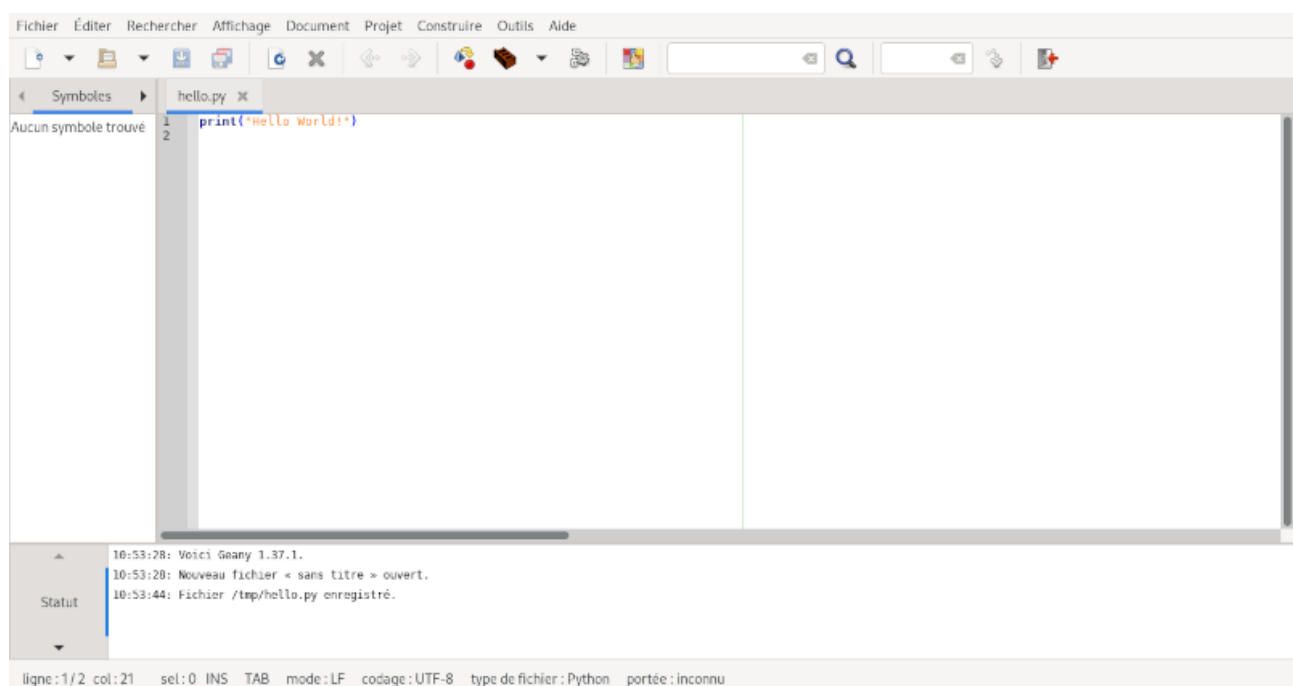


FIGURE I.2.5. – Geany

Le terminal dont je parlais se trouve dans la fenêtre des messages au bas de l'écran, vous pouvez faire défiler les onglets jusqu'à lui.

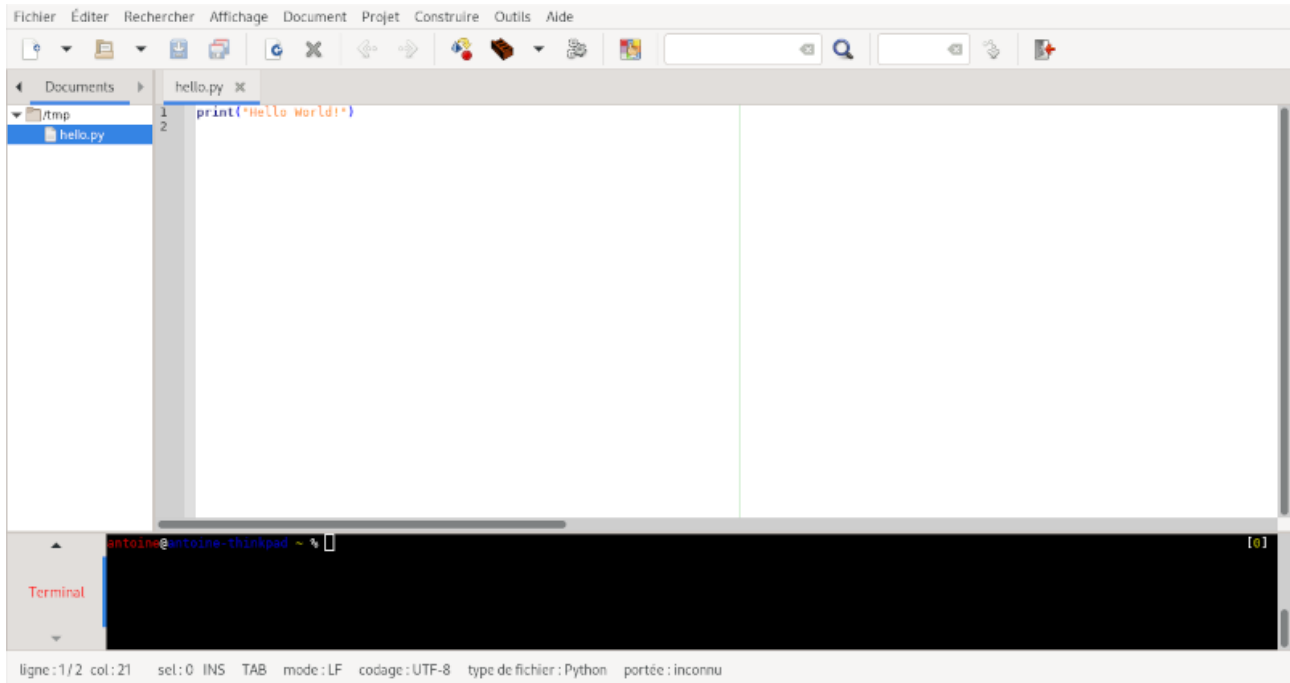


FIGURE I.2.6. – Terminal intégré à Geany

Une autre fonctionnalité intéressante de Geany est qu'il vous permet d'exécuter directement Python sur vos fichiers de code, grâce au menu *Construire* > *Execute* ou simplement par un clic sur le bouton dédié (rouages) ou la touche **F5**. Les commandes utilisées sont définies dans le menu *Construire* > *Définir les commandes de construction*.

I.2.5.2. PyCharm

J'aimerais aussi vous parler de PyCharm car c'est un éditeur de texte dédié au Python assez en vogue. Il est donc courant de le voir utilisé dans différents cours sur Internet, ou recommandé sur les forums. Il intègre aussi un débogueur directement dans l'éditeur, ce qui pourra être utile pour aider à dénicher les erreurs dans un programme.

C'est un logiciel édité par JetBrains, entreprise spécialisée dans les IDE (environnements de développement), qui est disponible en version gratuite (mais limitée). Comme toujours, vous pouvez le trouver sur le [site officiel de JetBrains](#) pour Windows et Mac, et dans votre gestionnaire d'applications sur Linux.

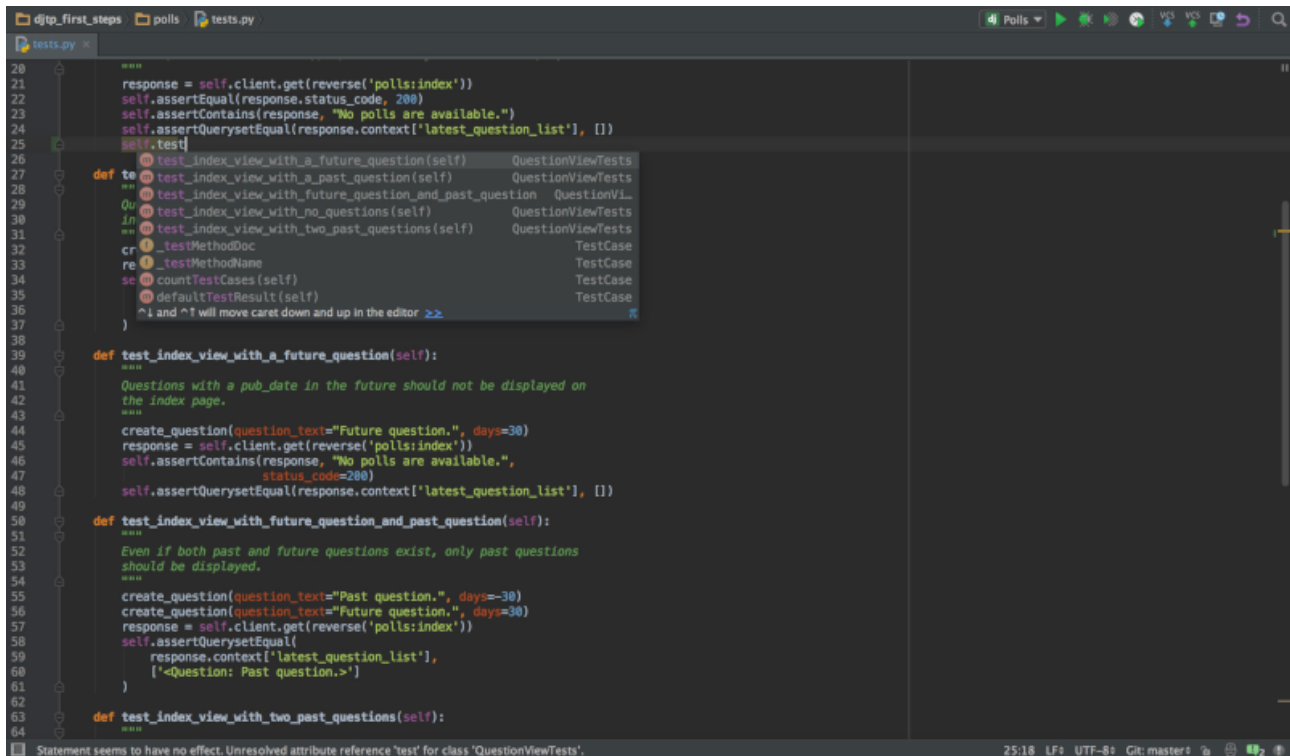


FIGURE I.2.7. – PyCharm

I.2.5.3. Autres éditeurs

Si vous possédez déjà un éditeur de texte dédié au code, vous pouvez regarder s'il gère déjà nativement la coloration pour Python ou si c'est quelque chose que vous pouvez installer afin de conserver ce même éditeur.

Conclusion

Pour de plus amples informations sur l'installation et la configuration de Python, ou pour l'installer sur d'autres systèmes d'exploitation que ceux listés ici, je vous conseille d'aller lire [la page dédiée dans la documentation](#) .

I.3. Prise en main de l'interpréteur

Introduction

Il est maintenant temps d'entrer dans le vif du sujet et de commencer à communiquer avec Python ! Pour cela nous avons besoin de lancer l'interpréteur, le programme qui comprendra et exécutera le code que nous entrerons.

I.3.1. Interpréteur interactif

Ainsi, comme nous l'avons vu dans le chapitre précédent, nous pouvons exécuter l'interpréteur interactif à l'aide du programme IDLE, ou de la commande `python` dans un terminal.

Vous vous retrouvez maintenant face à ce que l'on appelle une invite de commande. Cela se reconnaît par les `>>` en début de ligne, qu'on appelle le *prompt*. L'interpréteur attend que vous lui demandiez quelque chose.

Mais quoi ? Pour commencer, je vous propose d'entrer un nombre, Python le comprendra.

```
1 >>> 12
2 12
```

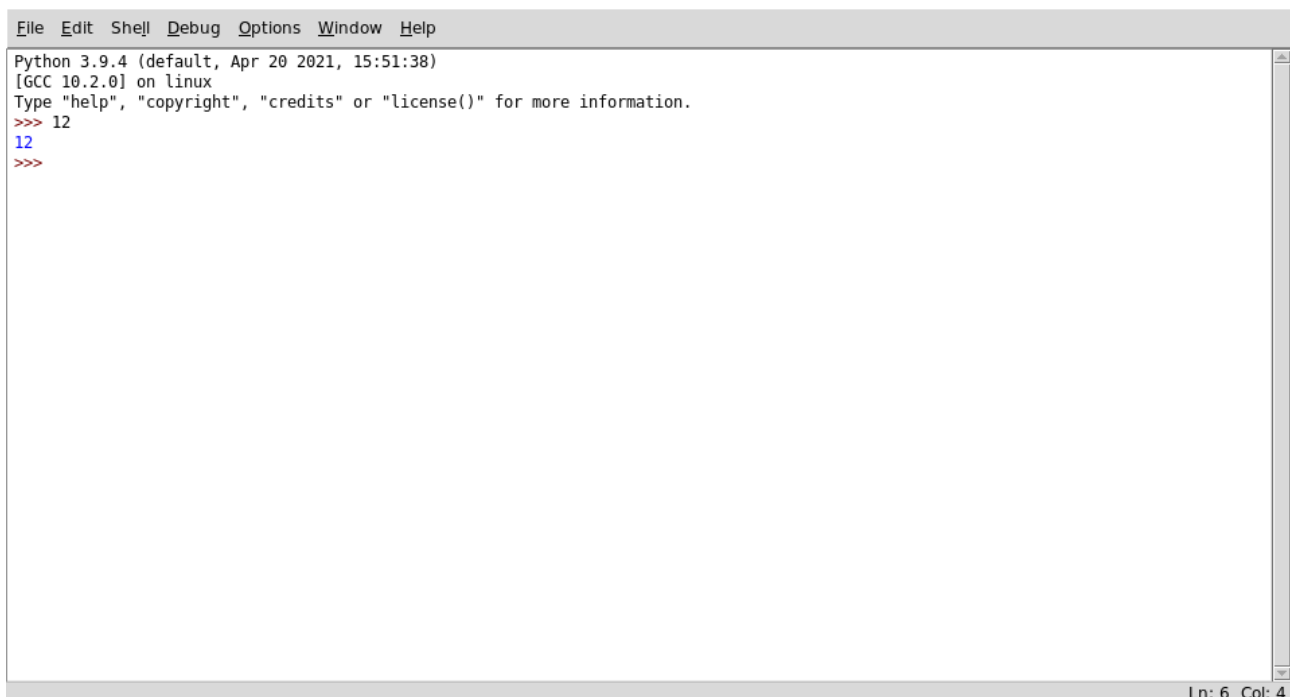


FIGURE I.3.1. – Exécution dans l'interpréteur interactif.

I. Premiers pas avec Python

Nous avons demandé à Python d'exécuter le code `12`, il a répondu `12`. Ce qu'on ne voit pas, c'est que Python a évalué le code que nous avons entré. Il l'a analysé et en a calculé le résultat. Puis il a affiché ce résultat pour que nous en ayons connaissance.

Ainsi, Python nous dit que `12` vaut `12`. Merci Python !



Notez que lorsqu'au long de ce cours je vous présenterai un code sous la forme suivante :

```
1 >>> xxx
2 yyy
```

Cela signifie qu'il faut exécuter le code `xxx` dans l'interpréteur Python et que celui-ci nous affiche alors `yyy`.

J'utiliserai donc cette notation avec des chevrons pour présenter des exemples de l'interpréteur interactif.

I.3.2. Une machine à calculer

Mais il ne se limite pas à cela, nous pouvons lui demander de calculer des opérations, comme une calculatrice. Tout ce que nous avons à faire, c'est d'exprimer le calcul dans les termes qu'il comprend.

Nous avons de la chance, Python utilise la même notation que nous pour l'addition.

```
1 >>> 5 + 3
2 8
```

Plus fort encore, on peut lui demander d'additionner plusieurs nombres !

```
1 >>> 1 + 2 + 3
2 6
```

Je mets des espaces autour des `+` mais celles-ci sont facultatives. Elles sont néanmoins préférables pour une bonne lisibilité du code.

```
1 >>> 1+2+3
2 6
```

Python connaît aussi la soustraction et les nombres négatifs.

```
1 >>> 8 - 5
2 3
3 >>> 1 - 10
4 -9
5 >>> 1 - -10
```

I. Premiers pas avec Python

```
6 11
```

Autre opération courante, Python sait évaluer la multiplication entre deux nombres. Mais il faut parler dans son langage, l'opérateur pour la multiplication est `*`.

```
1 >>> 4 * 5
2 20
3 >>> 6 * 7 * -1
4 -42
```

Et bien sûr, les priorités entre les opérations sont gérées. Quand en mathématiques on écrit « $1 + 2 \times 3$ », la multiplication est prioritaire sur l'addition, donc elle est exécutée en premier, d'où le résultat de «7». Il en est de même en Python :

```
1 >>> 1 + 2 * 3
2 7
```

Et comme en maths, on peut utiliser des parenthèses pour prioriser certaines opérations.

```
1 >>> (1 + 2) * 3
2 9
```

Les parenthèses permettent aussi de jouer sur l'associativité des opérateurs, `1 - 2 - 3` n'est pas la même chose que `1 - (2 - 3)`

```
1 >>> 1 - 2 - 3
2 -4
3 >>> 1 - (2 - 3)
4 2
```

Tout ce que nous demandons à Python doit être exprimé dans la syntaxe qu'il comprend—ici des nombres et des opérations, mais nous verrons par la suite qu'il est possible de bien plus. Dans le cas contraire, Python nous annoncera gentiment qu'il ne comprend pas ce que nous lui demandons, que la syntaxe est incorrecte.

```
1 >>> il fait beau aujourd'hui
2 File "<stdin>", line 1
3     il fait beau aujourd'hui
4         ^
5 SyntaxError: invalid syntax
6 >>> 1!
7 File "<stdin>", line 1
8     1!
9         ^
10 SyntaxError: invalid syntax
```

I.4. Écrire du code dans des fichiers

Introduction

Nous avons vu comment utiliser Python en mode interactif, mais l'interpréteur peut aussi exécuter du code contenu dans des fichiers. Cela nous sera utile pour réaliser nos premiers programmes.

Un fichier de code Python se présente comme un fichier texte avec une extension `.py`. On dit que le contenu du fichier est le code source du programme.

I.4.1. Éditer un fichier Python

Pour ouvrir un fichier avec IDLE, il suffit de cliquer sur le menu *File > New File* (ou utiliser le raccourci `Ctrl+N`). Il est aussi possible d'ouvrir un fichier existant avec *Open File* (`Ctrl+O`). Cela ouvrira une nouvelle fenêtre à côté de l'interpréteur interactif.

Vous vous retrouvez alors face à une fenêtre blanche, où il est possible d'entrer du texte, ou plutôt du code Python. On peut par exemple écrire le contenu suivant :

```
1 8 + 5
2 3 * 7
```

Listing 2 – calc.py

Là, vous pouvez rédiger votre code Python puis enregistrer le fichier avec l'option *Save* du menu (`Ctrl+S`). Pensez à nommer votre fichier avec une extension `.py`. Dans mon cas j'ai choisi `calc.py` comme nom de fichier.

Nous commençons simple pour le moment avec deux calculs faciles comme nous le faisons dans le chapitre précédent. Nos fichiers de code s'étofferont avec le temps pour devenir des programmes plus complets, mais `calc.py` est déjà un programme Python à part entière, qui ne permet que de calculer le résultat de simples opérations.

C'est à peu près le même fonctionnement si vous utilisez Geany comme éditeur de texte, vous trouvez le même genre d'options pour ouvrir un nouveau fichier, ouvrir un fichier existant et enregistrer le fichier courant, dans le menu *Fichier*.

I.4.1.1. En-têtes de fichier

Dans certains cas d'usage, si par la suite vous rencontrez des problèmes avec des caractères accentués par exemple, il peut être utile de définir des en-têtes à notre fichier Python.

Ce sont des données associées au fichier qui permettront à Python et au système d'exploitation d'interpréter correctement son contenu.

Je vous indique alors [cette section en annexe](#) qui vous en dira plus.

I.4.2. Exécution du fichier

Maintenant que nous avons écrit notre premier programme nous pouvons donc passer à la prochaine étape : l'exécuter !

Dans IDLE, cela se fait à l'aide du menu *Run > Run Module* (ou de la touche **F5**). De même dans Geany avec la commande *Execute* (**F5**).

On exécute donc le fichier à l'aide de l'interpréteur... et rien ne se passe. Enfin plus précisément on ne voit rien de particulier.

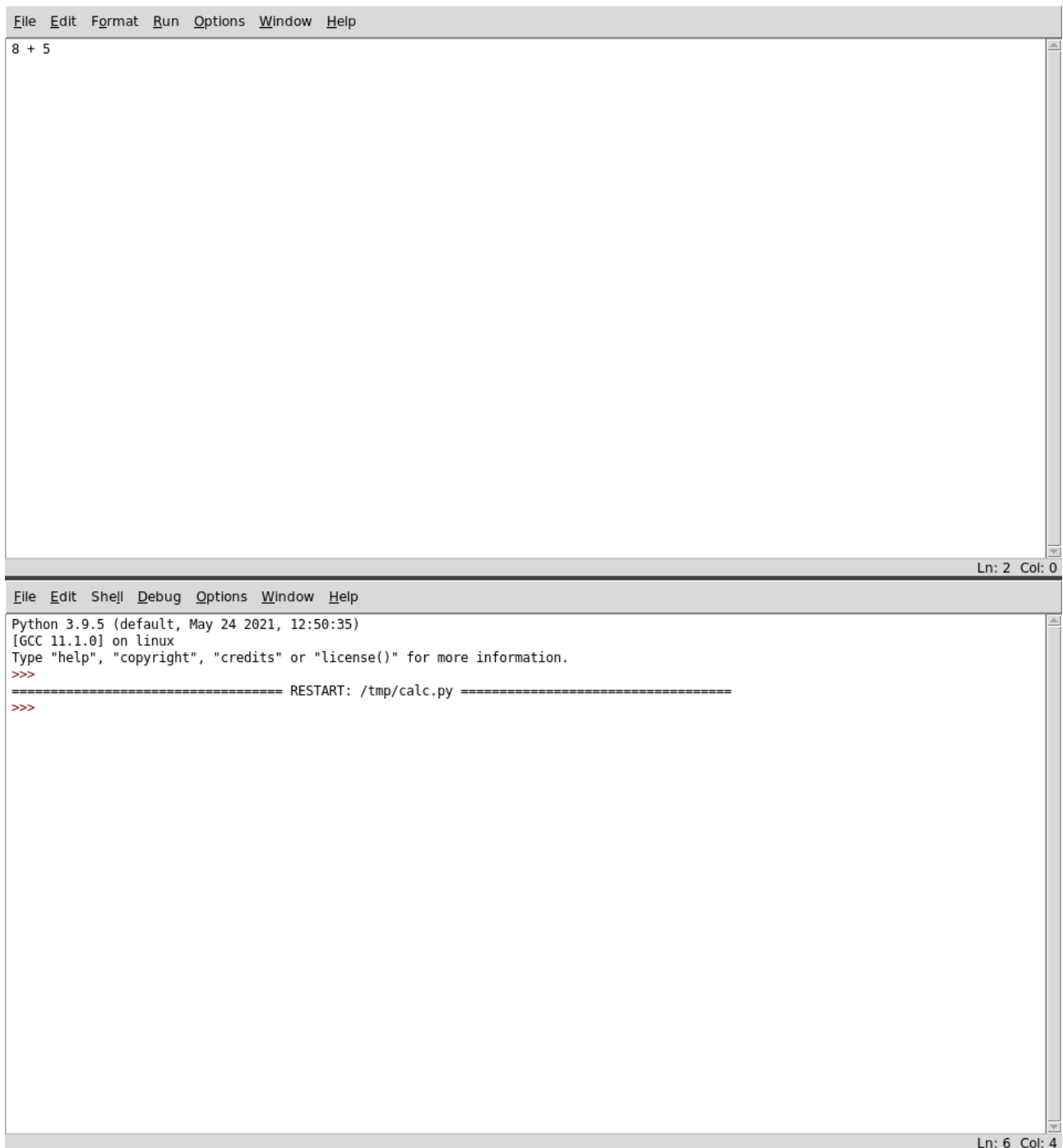


FIGURE I.4.1. – Exécution du fichier dans IDLE.

Le code a bien été exécuté mais il n'a affiché aucun résultat, donc rien de visible. Dans

I. Premiers pas avec Python

l'interpréteur interactif, la valeur calculée de chaque ligne était affichée dans le terminal, parce que c'est plus facile à suivre. Mais dans le cas d'un fichier, ça polluerait inutilement la console, on ne veut pas afficher le résultat de chaque calcul intermédiaire.

On aimerait tout de même pouvoir en afficher certains, et il existe pour cela la commande `print`. Elle permet, suivie d'une paire de parenthèses contenant une valeur, d'afficher cette valeur sur le terminal.

```
1 print(8 + 5)
2 print(3 * 7)
```

Listing 3 – calc.py

Il faut bien différencier l'affichage de l'évaluation. L'évaluation c'est le processus par lequel Python calcule le résultat d'une opération, sans nécessairement l'afficher.

I.4.2.0.1. Commentaires

Dans notre fichier, nous pouvons aussi placer des commentaires pour expliquer ce qui est fait. Les commentaires ne sont pas interprétés par Python, ils se destinent aux développeurs qui liront le fichier, et permettent de renseigner des informations ou documenter.

Un commentaire est simplement une ligne commençant par un `#` et suivie de n'importe quel texte. On peut aussi placer un commentaire derrière une ligne de code, toujours en le faisant précéder d'un `#`.

```
1 # Calcul du prix au kilo de pommes
2
3 # Nous avons acheté 500g de pommes pour 1€
4 print(1 / 0.5) # Prix total (€) / Poids des pommes (Kg)
```

Listing 4 – pommes.py

I.4.2.1. Exécution depuis le terminal

Voilà pour l'exécution depuis l'éditeur de texte, mais si vous êtes un adepte de la console vous voudrez peut-être aussi lancer votre programme depuis le terminal.

Pour cela, il suffit de lancer un terminal dans le répertoire où se trouve votre fichier de code (ou de se rendre dans le bon répertoire avec la commande `cd`) puis de lancer `python calc.py` (ou `python3` suivant la version par défaut).

```
1 % python calc.py
2 13
3 21
```



Attention sous Windows, pensez à utiliser un terminal persistant pour éviter que celui-ci ne se ferme à la fin du programme ou lorsqu'une erreur est rencontrée.



Évitez donc d'exécuter vos fichiers en double-cliquant dessus et préférez ouvrir un Powershell dans lequel vous appellerez Python comme ci-dessus.

I.5. Fichiers ou interpréteur interactif ?

Introduction

Nous avons vu les deux modes d'exécution de l'interpréteur Python, par fichiers et en interactif. Mais quel mode est le meilleur choix dans quelle situation ? C'est ce que nous allons voir dans ce chapitre.

I.5.1. Interpréteur interactif

L'interpréteur interactif est très pratique pour tester un bout de code rapidement ou explorer une valeur : on peut directement voir ce que contient une valeur, et facilement essayer telle ou telle ligne de code pour vérifier qu'elle fonctionne.

Il permet aussi de réaliser des calculs étape par étape.

Mais il n'est pas adapté pour écrire du plus long code, il gère mal l'indentation (c'est une spécificité du langage Python sur laquelle nous reviendrons par la suite) et les blocs de code à la suite.

De plus tout code tapé dans l'interpréteur interactif est à usage unique : il est possible de le retrouver dans l'historique de saisie, mais les lignes précédemment tapées ne sont plus modifiables.

i

Au long de ce cours j'utiliserai parfois le terme de *REPL* pour désigner l'interpréteur interactif. il signifie *Read-Evaluate-Print Loop*, soit un programme chargé de lire, évaluer et afficher en boucle ce qu'on lui demande.

C'est un terme plus générique que l'on rencontre aussi dans d'autres langages.

I.5.2. Écriture dans des fichiers

À l'inverse, les fichiers permettent d'éditer et de revenir sur le code, et donc de le sauvegarder. Ils permettent de gérer des codes beaucoup plus gros sans s'y perdre, parce que tout cela est géré par l'éditeur de texte.

Il est en effet possible de remonter dans le code pour aller modifier n'importe quelle ligne, puis de le réexécuter en totalité.

De plus, de nombreux éditeurs proposent une coloration syntaxique du code, ce qui permet d'avoir un rendu bien plus lisible, en identifiant facilement les mots-clés et constructions du langage.

Enfin, un programme s'écrit forcément dans des fichiers, pour pouvoir être partagé.

Mais cela demande d'écrire et de sauvegarder le fichier chaque fois que l'on veut tester une nouvelle partie du code, et d'utiliser `print` pour afficher les résultats voulus.

Conclusion

Les deux modes ont leurs avantages et leurs inconvénients, il convient donc de tirer partie des deux.

De plus ils ne sont pas exclusifs, il est ainsi possible de lancer l'interpréteur avec l'option `-i` sur un fichier, pour obtenir un interpréteur interactif à la suite de l'exécution d'un fichier. Ce qui est bien pratique pour vérifier un résultat dans un programme en cours de développement.

Par exemple si on reprend le fichier suivant :

```
1 print(8 + 5)
2 print(3 * 7)
```

Listing 5 – calc.py

On peut l'exécuter avec l'option `-i` et continuer à utiliser l'interpréteur à la suite du fichier.

```
1 % python -i calc.py
2 13
3 21
4 >>> 1 / 2
5 0.5
```

Son utilité n'apparaît peut-être pas flagrante pour le moment, mais ça nous sera utile quand nous aurons des fichiers plus complets et que nous voudrions y tester une valeur en particulier.

L'option `-c` de l'interpréteur est aussi utile pour lancer l'interpréteur et exécuter une simple ligne.

```
1 $ python -c 'print(1+5)'
2 6
```

Dans ce tutoriel, j'utiliserai toujours les signes `>>` pour présenter les codes exécutés dans l'interpréteur interactif, et donc suivis du résultat affiché. Les autres codes seront considérés comme écrits dans des fichiers.

Deuxième partie

Manipuler des données

Introduction

Au-delà de la simple calculatrice, voyons maintenant ce qu'il est possible de faire avec Python et comment réaliser nos premiers *vrais* programmes.

II.1. Retour sur la calculatrice

Introduction

En utilisant Python en tant que calculatrice, nous avons remarqué qu'il était capable d'interpréter différentes opérations sur les nombres.

Voyons maintenant ce que nous réserve d'autre cette super-calculatrice.

II.1.1. Des nombres à virgule

Nous nous sommes intéressés aux principaux opérateurs arithmétiques à l'exception de l'un d'entre-eux : l'opérateur de division (`/`). Il est un peu différent des autres parce que la division entre deux nombres entiers n'est pas nécessairement un nombre entier.

En effet, que vaut «5 divisé par 2» (`5 / 2`) ? Aucun des nombres que l'on sait représenter n'est égal à ce résultat. Il nous faut aller au-delà des nombres entiers pour découvrir le monde des nombres à virgule, ou nombres flottants. Les nombres flottants se composent d'une partie entière et d'une partie fractionnaire séparées par un point (notation anglaise). Ainsi le résultat de notre précédent calcul se note `2.5`.

```
1 >>> 5 / 2
2 2.5
```

Chaque valeur en Python est associée à un type, c'est-à-dire qu'elle appartient à une certaine catégorie. Cette catégorie régit les opérations qui sont applicables sur ses valeurs. Les nombres entiers (`int`) et les flottants (`float`) sont deux types différents, deux catégories distinctes.

```
1 >>> 8 / 2
2 4.0
```

Ainsi ici Python nous renvoie la valeur `4.0`, qui n'est pas la même chose que `4`. Les deux valeurs sont égales et représentent le même nombre, mais elles ne sont pas du même type en Python. Les opérations que nous avons vues sur les nombres entiers s'appliquent aussi aux flottants, la différence étant que le résultat sera toujours un nombre flottant.

```
1 >>> 1.1 + 3.4
2 4.5
3 >>> 4.5 * 2.7
4 12.15
5 >>> 12.15 - 0.1
6 12.05
```


II. Manipuler des données

```
7 >>> 12.05 / 0.5
8 24.1
```

Et ces deux types de nombres sont compatibles entre-eux, il est par exemple possible d'additionner un entier et un flottant. Là aussi le résultat sera un flottant, pour éviter toute perte d'information de la partie fractionnaire.

```
1 >>> 5 + 0.8
2 5.8
3 >>> 0.3 * 10
4 3.0
```

Une chose à laquelle il faut faire attention avec les nombres à virgule se situe sur les arrondis. Par exemple, il n'est pas possible de représenter la division de 8 par 3 par un nombre à virgule précis, et c'est donc le nombre le plus proche qui nous sera renvoyé par cette opération.

```
1 >>> 8 / 3
2 2.6666666666666665
```

Mais ça ne s'arrête pas là. Contrairement à nous qui avons l'habitude du système décimal, l'ordinateur stocke les nombres sous forme binaire.

Ainsi, tous les nombres décimaux que nous utilisons ne sont pas représentables par un flottant, et Python devra effectuer un arrondi.

C'est le cas de `0.1` qui est en fait égal à `0.100000000000000005...`. À l'usage, il est ainsi courant de rencontrer des cas où ces erreurs d'arrondis deviennent visibles, comme dans les exemples suivants.

```
1 >>> 0.1 + 0.1 + 0.1
2 0.30000000000000004
3 >>> 1.5 * 1.6
4 2.4000000000000004
```

Vous trouverez plus d'information sur ces erreurs dans [le tutoriel d'@Aabu dédié à l'arithmétique flottante](#) [↗](#) que je vous invite à consulter après ce cours.

II.1.2. Autres opérateurs

L'opérateur de division (`/`) entre deux nombres calcule une division décimale et renvoie un nombre flottant, mais ce n'est pas la seule opération de division possible.

En effet, Python permet aussi de réaliser une division euclidienne (ou division entière) avec les opérateurs `//` et `%`, calculant respectivement le quotient et le reste de la division.

Souvenez-vous : cela correspond à la division posée que l'on apprenait à l'école où à partir du dividende et du diviseur, par multiplications et soustractions successives, on trouvait ce quotient et ce reste (indivisible).

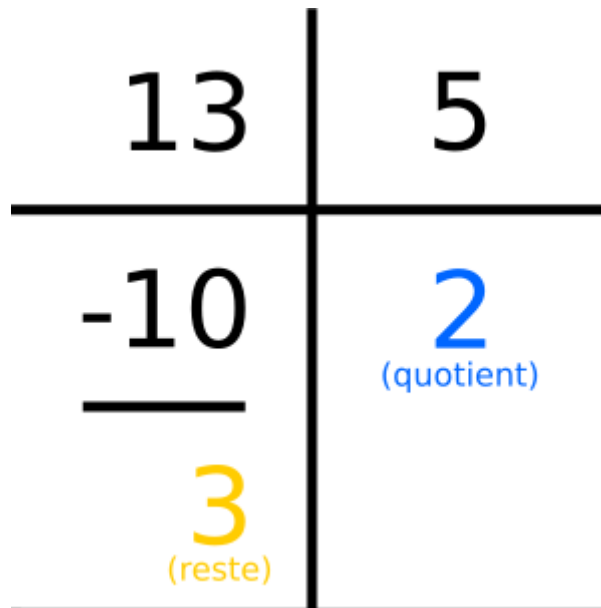


FIGURE II.1.1. – Division euclidienne

```

1 >>> 13 // 5
2 2
3 >>> 13 % 5
4 3

```

On peut vérifier notre résultat en multipliant le quotient par le diviseur et en lui ajoutant le reste.

```

1 >>> 2 * 5 + 3
2 13

```

Ces opérations renvoient des nombres entiers quand elles sont appliquées à des nombres entiers. Une autre opération mathématique courante est l'exponentiation, autrement dit la mise en puissance. Cette opération se note `**`, avec le nombre à gauche et la puissance à droite.

```

1 >>> 5 ** 2 # 5 à la puissance 2 soit 5 au carré
2 25
3 >>> 1.5 ** 3 # 1.5 au cube
4 3.375

```

Et pour les connaisseurs il est aussi possible d'utiliser des puissances flottantes, comme `0.5` pour calculer une racine carrée.

```

1 >>> 2 ** 0.5
2 1.4142135623730951

```

II.1.2.1. Priorités des opérateurs

Comme nous l'avons vu, les opérateurs ont chacun leur priorité, et celle-ci peut être changée à l'aide de parenthèses.

Ainsi, l'exponentiation est prioritaire sur la multiplication et la division, elles-mêmes prioritaires sur l'addition et la soustraction.

Prio- rité des opé- ra- teurs	Opé- ra- teur
1	(...)
2	**
3	*, /, //, %
4	+, -

Et chaque opérateur a aussi ses propres règles d'associativité. Ce sont des règles qui indiquent si, pour des opérations de même priorité, elles doivent s'exécuter de gauche à droite ou de droite à gauche.

Si elles importent peu pour l'addition et la multiplication ($(1+2)+3$ et $1+(2+3)$ ont la même valeur, de même pour $(2*3)*4$ et $2*(3*4)$), elles le sont pour les autres opérations.

Les opérations de priorités 3 et 4 (addition, soustraction, multiplication, divisions) sont toutes associatives à gauche, c'est-à-dire que les opérations de gauche sont exécutées en priorité, de façon à ce que $1 - 2 + 3$ soit égal à $(1-2) + 3$.

1	>>> 1 - 2 + 3
2	2
3	>>> (1 - 2) + 3
4	2
5	>>> 1 - (2 + 3)
6	-4
7	>>>
8	>>> 1 / 2 / 3
9	0.16666666666666666
10	>>> (1 / 2) / 3
11	0.16666666666666666
12	>>> 1 / (2 / 3)
13	1.5
14	>>>

II. Manipuler des données

```
15 >>> 1 / 2 * 3
16 1.5
17 >>> (1 / 2) * 3
18 1.5
19 >>> 1 / (2 * 3)
20 0.16666666666666666
```

À l'inverse, l'opération d'exponentiation (`**`) est associative à droite, donc les opérations sont exécutées de droite à gauche.

```
1 >>> 2 ** 3 ** 4
2 2417851639229258349412352
3 >>> 2 ** (3 ** 4)
4 2417851639229258349412352
5 >>> (2 ** 3) ** 4
6 4096
```

II.1.3. Fonctions

Mais notre calculatrice ne s'arrête pas à ces simples opérateurs, elle est aussi capable d'appliquer des fonctions sur nos nombres. Une fonction est une opération particulière à laquelle on va donner une valeur en entrée et qui va en renvoyer une nouvelle, comme en mathématiques. Par exemple, `abs` est la fonction qui calcule la valeur absolue d'un nombre (il s'agit grossièrement de la valeur de ce nombre sans le signe `+` ou `-`).

Pour appliquer une fonction sur une valeur, on écrit le nom de la fonction suivi d'une paire de parenthèses, entre lesquelles on place notre valeur.

```
1 >>> abs(-5)
2 5
3 >>> abs(3.2)
4 3.2
```

Faites bien attention aux parenthèses qui sont obligatoires pour appeler une fonction. L'appel sans parenthèses, qui est parfois d'usage en mathématiques ou dans d'autres langages de programmation, produit ici une erreur de syntaxe.

```
1 >>> abs 3.2
2 File "<stdin>", line 1
3     abs 3.2
4         ^
5 SyntaxError: invalid syntax
```

Une autre fonction sur les nombres fournie par Python est la fonction `round` qui permet de calculer l'arrondi à l'entier d'un nombre flottant.

II. Manipuler des données

```
1 >>> round(1.4)
2 1
3 >>> round(1.5)
4 2
```

Ces deux fonctions `abs` et `round` sont prédictibles : pour une même valeur en entrée le résultat sera toujours le même. On pourrait les appeler à l'infini et obtenir toujours la même chose. Le résultat d'une fonction est donc une valeur comme une autre, ici un nombre, que l'on peut alors utiliser au sein d'autres opérations.

```
1 >>> abs(-2) * (round(3.7) - 1)
2 6
```

C'est ce que l'on appelle une «**expression**», cela désigne une ligne de Python qui produit une valeur.

Cela peut-être une simple valeur (42), une opération (3 * 5) ou un appel de fonction (abs(-2)) : tous ces exemples sont des expressions, qui peuvent donc se composer les unes avec les autres dans de plus grandes expressions.

```
1 >>> 42 - 3 * 5 + abs(-2)
2 29
```

La valeur que l'on envoie à la fonction est appelée un **argument**. `abs(-5)` se lit «appel de la fonction `abs` avec l'argument -5», et 5 est la **valeur de retour** de la fonction.

Un argument est aussi une expression, et l'on peut donc faire un appel de fonction sur une opération et non juste sur une valeur littérale.

```
1 >>> abs(3 - 10)
2 7
3 >>> round(9 / 2)
4 4
```

Les exemples précédents présentaient des appels avec un unique argument. Mais certaines fonctions vont pouvoir recevoir plusieurs arguments, qui devront alors être séparés par des virgules lors de l'appel. Il convient de mettre une espace derrière la virgule pour bien aérer le code.

C'est le cas de la fonction `round`, qui prend un deuxième argument optionnel permettant de préciser combien de chiffres après la virgule on souhaite conserver. Par défaut on n'en conserve aucun.

```
1 >>> round(2.3456)
2 2
3 >>> round(2.3456, 1)
4 2.3
```

II. Manipuler des données

```
5 >>> round(2.3456, 2)
6 2.35
7 >>> round(2.3456, 3)
8 2.346
9 >>> round(2.3456, 4)
10 2.3456
```

D'autres fonctions vont recevoir plusieurs arguments, c'est le cas par exemple de `min`, qui renvoie la plus petite valeur de ses arguments. À l'inverse, `max` renvoie la plus grande valeur.

```
1 >>> min(4, 9, -2, 7)
2 -2
3 >>> max(4, 9, -2, 7)
4 9
```

Une fonction est toujours associée à un «ensemble de définition», on ne peut que lui donner des arguments qui sont cohérents avec le calcul qu'elle doit réaliser. `abs(1, 2)` et `min(1)` sont par exemple des appels qui n'ont pas de sens et qui produiront des erreurs.

```
1 >>> abs(1, 2)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: abs() takes exactly one argument (2 given)
5 >>> min(1)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: 'int' object is not iterable
```

Nous verrons par la suite ce que signifient précisément ces erreurs. Pour l'instant, retenez qu'une fonction attend un certain nombre d'arguments, de certains types. Et que déroger à ces règles produit des erreurs.

II.2. Des variables pour sauvegarder nos résultats

Introduction

Lorsque nous entrons une expression dans l'interpréteur interactif, sa valeur est calculée puis affichée dans le terminal. Mais après cela elle est perdue. Pourtant il pourrait nous être utile de conserver un résultat, afin de le réutiliser par la suite dans d'autres calculs.

Par exemple, dans un jeu, on aimerait pouvoir conserver le nombre de points de vie d'un joueur, pour l'utiliser dans le calcul des dégâts ou pour le modifier.

Comment faire alors ? Grâce aux variables bien sûr !

II.2.1. Une étiquette sur une valeur

En effet, ce sont les variables qui vont nous permettre de stocker nos résultats de calculs. Une variable, c'est juste un nom que l'on associe à une valeur, afin d'indiquer à Python de la conserver en mémoire (de ne pas l'effacer) mais aussi de pouvoir la retrouver (grâce à son nom).

On peut voir la variable comme une simple étiquette qui sera collée sur notre valeur pour indiquer comment elle se nomme.

En Python, on assigne une variable sur une valeur à l'aide de l'opérateur `=`. À gauche on écrit le nom de la variable, une suite de lettres sans espace. La valeur peut être n'importe quelle expression comme vu précédemment.

```
1 >>> result = round(8 / 3) + 2
```

On voit que l'interpréteur ne nous affiche rien cette fois-ci, parce que le résultat a été stocké dans `result`. `result` est une variable qui pointe non pas vers l'expression `round(8 / 3) + 2` mais vers le résultat de cette opération, soit le nombre 5.



FIGURE II.2.1. – Une variable est une étiquette sur une valeur.

Si l'interpréteur ne nous affiche rien, c'est aussi parce que `result = round(8 / 3) + 2` n'est pas une expression. Cette ligne définit une variable mais ne possède pas de valeur à proprement parler. On ne peut pas l'utiliser au sein d'une autre expression. On dit simplement qu'il s'agit d'une instruction.

Le nom de la variable définie devient quant à lui une valeur comme une autre, qui peut être utilisée dans différentes opérations.

Dans chaque expression, le nom de variable est évalué par Python et remplacé par sa valeur, permettant donc d'exécuter la suite du calcul.

```
1 >>> result
2 5
3 >>> result + 1
4 6
5 >>> min(result + 2, result * 2)
6 7
```

Et par extension, il est donc possible de définir une variable à l'aide de la valeur d'une autre variable :

```
1 >>> result2 = result - 1
2 >>> result2
3 4
```

II.2.2. Assignations

Comme son nom l'indique, une variable n'est pas fixée dans le temps. À tout moment, il est possible de la réassigner sur une nouvelle valeur, perdant ainsi la trace de l'ancienne.

```
1 >>> result = 6 * 7
2 >>> result
3 42
4 >>> result = 9 * 4
5 >>> result
6 36
```

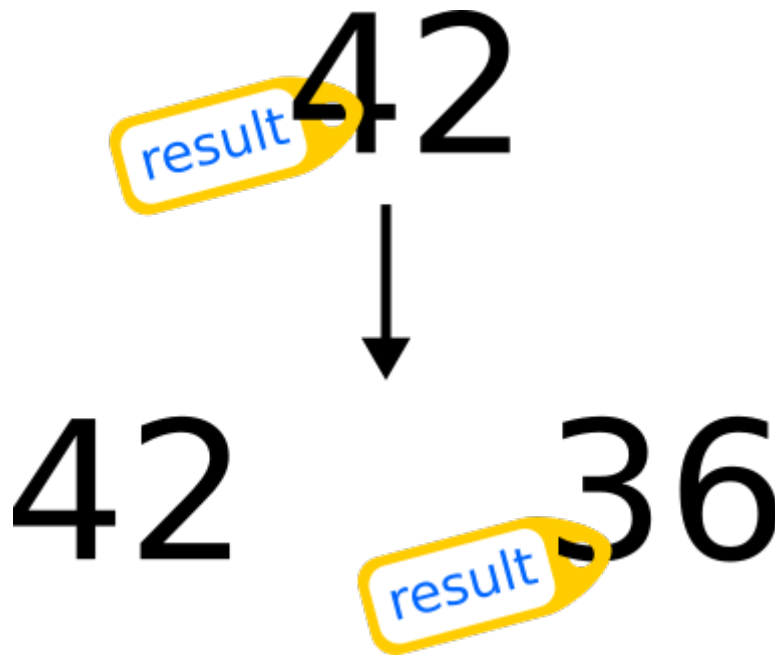



FIGURE II.2.2. – Réassignation de variable.

Mais on peut aussi utiliser une même variable à gauche et à droite de la définition.

```
1 >>> result = result + 1
```

Il ne faut donc pas voir ici le `=` comme une égalité, mais bien comme une assignation.

La ligne précédente signifie que l'on prend la valeur actuelle de `result` (36), que l'on lui ajoute 1, et que l'on assigne ce nouveau résultat à la variable `result`. `result` vaut donc maintenant 37.

Autre exemple avec la réassignation d'une variable `x`.

```
1 >>> x = 3
2 >>> x = x + 2
3 >>> x
4 5
```

Les opérations du type `x = x + y` sont d'ailleurs tellement courantes que Python leur a prévu un opérateur d'affectation spécifique : `+=`.

`x += 1` est ainsi équivalent à `x = x + 1`. On appelle cette opération une *incrément*, car on ajoute un incrément au nombre actuel.

Et cela ne se limite pas à l'addition mais comprend aussi les autres opérateurs arithmétiques qui bénéficient tous de leur propre opérateur d'affectation : `-=`, `*=`, `/=`, `//=`, `%=` et `**=`. L'opération de soustraction-assignation (`-=`) s'appelle une *décrément*.

```
1 >>> x = 0
2 >>> x -= 8
3 >>> x *= -11
4 >>> x //= 4
```

II. Manipuler des données

```
5 >>> x **= 3
6 >>> x %= 10
7 >>> x
8 8
```

Une autre propriété intéressante de l'opérateur `=` est qu'il peut être chaîné afin de définir plusieurs variables en même temps.

```
1 >>> x = y = 10
2 >>> x
3 10
4 >>> y
5 10
```

II.2.3. Conventions

Un nom de variable ne peut pas être composé de n'importe quels caractères. Il ne doit contenir que des lettres (minuscules ou majuscules), des chiffres et des *underscores* (caractère `_`). L'autre règle est que le nom ne peut pas commencer par un chiffre.

C'est-à-dire que le nom ne peut pas contenir d'espaces ou de caractères spéciaux, contrevenir à ces règles produira des erreurs de syntaxe.

```
1 >>> 0x = 1
2 File "<stdin>", line 1
3     0x = 1
4     ^
5 SyntaxError: invalid token
6 >>> x y = 1
7 File "<stdin>", line 1
8     x y = 1
9     ^
10 SyntaxError: invalid syntax
11 >>> x! = 1
12 File "<stdin>", line 1
13     x! = 1
14     ^
15 SyntaxError: invalid syntax
```

Certains noms sont aussi réservés, car ils correspondent à des mots-clés Python. Il est ainsi impossible de nommer une variable avec l'un des noms présent dans le tableau suivant.

False	import	pass
None	raise	print
True	return	yield

an	cont	fl	atry	a
nue				
		non		
as	de	fr	low	while
			cal	
as	gl			
se	de	in	with	
	se	ba		
as	li	for	yield	

```
1 Type "help", "copyright", "credits" or "license" for more
  information.
2 >>> def = 10
3     File "<stdin>", line 1
4         def = 10
5             ^
6 SyntaxError: invalid syntax
```

Il faut ajouter à cela quelques conventions de style. Il est ainsi conseillé d'éviter les lettres majuscules et accentuées dans les noms de variables. Par exemple, pour un nom de variable composé de plusieurs mots, on préférera `points_vie` à `pointsVie`.

Mais on préférera souvent utiliser l'anglais pour garder une cohérence avec les mots-clés du langage et faciliter les collaborations, notre variable se nommerait donc plutôt `health_points`.

Aussi, il est déconseillé de nommer une variable d'un même nom qu'une fonction de Python, comme `abs`, `min` ou `max`.

On évitera enfin les noms **l**, **0** ou **I** qui portent à confusion car ne sont pas bien distinguables de 1 ou 0 avec certaines polices de caractères.

i

Les différentes règles de style à appliquer en Python sont décrites dans la [PEP8](#) ⁷. Il s'agit d'un guide écrit par les créateurs de Python pour aider à comprendre ces conventions.

Une section de la PEP8 est particulièrement dédiée au nommage : <https://www.python.org/dev/peps/pep-0008/#naming-conventions> .

II.2.3.1. La variable

Autre convention, il est courant d'appeler `_` une variable dont on n'utilise pas le résultat. Cela est utile dans des cas où il est nécessaire de préciser un nom de variable mais dont on ne veut pas vraiment conserver la valeur. On verra ça par la suite avec les assignations multiples où `_` pourra servir à combler les trous.

La variable `_` a aussi un sens spécial dans l'interpréteur interactif : elle garde la trace de la dernière expression calculée et affichée.

II. Manipuler des données

```
1 >>> 1 + 2
2 3
3 >>> _
4 3
5 >>> _ + 1
6 4
7 >>> _ + 1
8 5
```

II.3. Manipuler du texte

Introduction

Et si nous apprenions maintenant à Python à parler ?

Nous nous efforçons à parler sa langue, ce serait bien qu'il fasse aussi un pas vers nous.

II.3.1. Chaînes de caractères

Nous n'avons jusqu'ici manipulé que des nombres, mais ce ne sont pas les seuls types de données utilisables en Python, bien heureusement. Bien que la mémoire de l'ordinateur ne sache traiter que des nombres, les langages de programmation offrent des abstractions pour représenter d'autres données.

Ainsi Python sait associer chaque lettre ou chaque caractère à un nombre grâce aux tables d'encodage. Et le texte, ce n'est au final qu'une suite de lettres, une séquence de caractères. On parle alors d'une chaîne de caractères.

On définit une chaîne de caractères à l'aide d'une paire de guillemets (*double-quotes*), entre lesquels on place le texte voulu.

```
1 >>> "Salut les gens !"
2 'Salut les gens !'
```

On voit Python nous répondre par cette même chaîne délimitée par des apostrophes (*quotes*). Il s'agit juste de deux syntaxes équivalentes pour représenter la même chose : une chaîne peut être délimitée par des apostrophes ou des guillemets, cela revient au même.

```
1 >>> 'toto'
2 'toto'
3 >>> "toto"
4 'toto'
```

Les chaînes de caractères sont un type de valeur et donc des expressions, qu'il est possible d'assigner à des variables.

```
1 >>> text = 'toto'
```

Si l'on appelle `print` sur une chaîne de caractères, son contenu est simplement affiché sur le terminal, sans les délimiteurs.

II. Manipuler des données

```
1 >>> print(text)
2 toto
3 >>> print('Salut les gens !')
4 Salut les gens !
```

L'avantage des deux syntaxes pour délimiter les chaînes, c'est qu'il est possible d'entourer la chaîne d'apostrophes pour lui faire contenir des guillemets, et inversement.

```
1 >>> 'Il a dit "salut"'
2 'Il a dit "salut"'
3 >>> "Oui il l'a dit"
4 "Oui il l'a dit"
```

Autrement, on aurait le droit à de belles erreurs car Python penserait en rencontrant le premier guillemet que l'on termine la chaîne, et il ne comprendrait donc pas les caractères qui suivraient.

```
1 >>> "Il a dit "salut""
2 File "<stdin>", line 1
3     "Il a dit "salut""
4                     ^
5 SyntaxError: invalid syntax
```

Mais comment alors représenter une chaîne de caractères possédant à la fois des apostrophes et des guillemets (telle que `J'ai dit "salut"`) ? La solution se situe au niveau de l'échappement. Il suffit de faire précéder un caractère d'un *backslash* (ou *antislash*, `\`) pour qu'il ne soit pas interprété par Python comme un caractère de délimitation.

```
1 >>> 'J\'ai dit "salut"'
2 'J\'ai dit "salut"'
```

Ces échappements, comme les délimiteurs, disparaissent lorsque le texte est affiché à l'aide d'un `print`.

```
1 >>> print('J\'ai dit "salut"')
2 J'ai dit "salut"
```

D'autres séquences d'échappement sont disponibles, comme `\t` pour représenter une tabulation (alinéa) ou `\n` pour un saut de ligne (*n* comme *newline*, soit *nouvelle ligne*). Il n'est en effet pas possible de revenir à la ligne dans une chaîne de caractères, et le `\n` est donc nécessaire pour insérer un saut de ligne.

```
1 >>> print('Elle a dit :\t"Salut"')
2 Elle a dit :      "Salut"
3 >>> print('Première ligne\nDeuxième ligne')
```

II. Manipuler des données

```
4 Première ligne
5 Deuxième ligne
```



Certains systèmes d'exploitation comme Windows pourraient ne pas bien interpréter le `\n` comme un saut de ligne et demander à ce qu'il soit précédé du caractère «retour-chariot» (`\r`) pour fonctionner.

```
1 >>> print('Une\r\nDeux')
2 Une
3 Deux
```

C'est un héritage de l'époque des machines à écrire où il fallait à la fois passer à la ligne suivante (nouvelle ligne) et revenir en début de ligne (retour chariot).

Et enfin, le *backslash* étant lui-même un caractère spécial, il est nécessaire de l'échapper (donc le doubler) si on veut l'insérer dans une chaîne. Comme par exemple pour un chemin de fichier sous Windows :

```
1 >>> print('C:\\Python\\projet\\example.py')
2 C:\Python\projet\example.py
```

Afin de moins avoir recours aux séquences d'échappement, il est aussi possible d'utiliser des *triple-quotes* pour définir une chaîne de caractères. Il s'agit de délimiter notre chaîne par trois apostrophes (ou trois guillemets) de chaque côté, lui permettant alors d'utiliser librement apostrophes et guillemets à l'intérieur, mais aussi des retours à la ligne.

```
1 >>> print(''''J'ai dit "salut"''')
2 J'ai dit "salut"
3 >>> print("""Une chaîne sur
4 ... plusieurs lignes
5 ... avec des ' et des " dedans""")
6 Une chaîne sur
7 plusieurs lignes
8 avec des ' et des " dedans
```



On voit des `...` apparaître à la place des `>>` dans l'interpréteur interactif. Cela signifie que l'interpréteur ne peut pas exécuter telle quelle la ligne de code entrée et qu'il attend pour cela les lignes suivantes, qui compléteront le code.

II.3.2. Opérations sur les chaînes

Une chaîne de caractères est une valeur à part entière, et comme toute valeur elle a certaines opérations qui lui sont applicables.

II. Manipuler des données

Pour commencer, la fonction `len` est une fonction de base de Python, qui peut être appelée avec une chaîne de caractères en argument. La fonction renvoie un nombre entier représentant la longueur de la chaîne, c'est-à-dire le nombre de caractères qu'elle contient.

```
1 >>> len('Hello')
2 5
3 >>> len('Hello World!')
4 12
```

C'est une fonction assez utile puisqu'elle nous permet par exemple de calculer l'espace occupé à l'écran par notre texte.

Mais d'autres opérations agissent directement sur le texte. C'est le cas de l'opérateur d'addition (+) que nous avons vu pour les nombres et qui existe aussi pour le texte, mais pour lequel il a un sens un peu différent.

On ne va en effet pas additionner deux chaînes de caractères, ça n'aurait pas de sens, mais on va les mettre l'une à la suite de l'autre. On appelle cette opération une concaténation.

```
1 >>> 'Hello' + ' ' + 'World' + '!'
2 'Hello World!'
```

Les délimiteurs ne faisant pas partie de la chaîne, il est bien sûr possible de mixer des chaînes délimitées par des apostrophes avec d'autres délimitées par des guillemets.

```
1 >>> 'abc' + "def"
2 'abcdef'
```

Nous retrouvons aussi l'opérateur de multiplication `*` pour représenter un autre type de concaténation : la répétition d'une chaîne un certain nombre de fois. `'to' * 3` est ainsi équivalent à `'to' + 'to' + 'to'`.

```
1 >>> 'to' * 3
2 'tototo'
```

On peut multiplier un texte par un nombre nul ou négatif, cela a pour effet de produire une chaîne vide. En revanche multiplier une chaîne par un nombre flottant n'a aucun sens, et Python nous le fait bien comprendre.

```
1 >>> 'toto' * 0
2 ''
3 >>> 'toto' * -10
4 ''
5 >>> 'toto' * 1.5
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: can't multiply sequence by non-int of type 'float'
```


II. Manipuler des données

Toutes les facilités vues jusqu'ici avec les opérateurs d'assignation restent bien sûr valables.

```
1 >>> msg = 'Salut '
2 >>> msg += 'tom'*2 + ' et ' + 'na'*2
3 >>> print(msg)
4 Salut tomtom et nana
```

Mais nous découvrons aussi d'autres opérateurs que nous n'avions pas vus jusque là, pour des opérations spécifiques à ce nouveau type de données.

Les chaînes de caractères formant une séquence d'éléments (des caractères), il est possible d'accéder de façon directe à chacun de ces éléments.

Cela se fait à l'aide de crochets ([]) en indiquant entre-eux la position du caractère voulu, par exemple `msg[3]`.

Il faut savoir que généralement en informatique on compte à partir de 0. Le premier caractère d'une chaîne se trouve donc à la position 0 de la séquence, le deuxième caractère à la position 1, etc. jusqu'au n -ième caractère à la position $n-1$.

```
1 >>> msg = 'Salut'
2 >>> msg[0]
3 'S'
4 >>> msg[0] + msg[1] + msg[2] + msg[3] + msg[4]
5 'Salut'
```

La valeur `'S'` renvoyée par `msg[0]` est un caractère, c'est-à-dire en Python une chaîne de taille 1.

On peut alors représenter notre chaîne de caractères `'Salut'` sous la forme d'un tableau, associant une position (un index) à chaque caractère de la chaîne :

Index	Caractère
0	's'
1	'a'
2	'l'
3	'u'
4	't'

Il est ainsi possible d'accéder à n'importe quel caractère de la chaîne à partir de son index, s'il est compris dans les bornes (de 0 à `len(msg)-1`).

```
1 >>> msg[5]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   IndexError: string index out of range
```

On observe qu'au-delà on obtient une erreur `IndexError`, soit un index invalide. On peut en

II. Manipuler des données

revanche utiliser des index négatifs pour prendre la chaîne en sens inverse : -1 correspond au dernier caractère, -2 à l'avant dernier, jusqu'à `-len(msg)` pour le premier. Chaque caractère a ainsi deux positions possibles dans la chaîne.

```
1 >>> msg[-1]
2 't'
3 >>> msg[-3]
4 'l'
5 >>> msg[-5]
6 's'
```

Index	Caractère
-5	's'
-4	'a'
-3	'l'
-2	'u'
-1	't'

II.4. Interactions

Introduction

Pour casser la monotonie de nos programmes, nous allons maintenant voir comment y ajouter un peu d'interactions.

II.4.1. Entrées utilisateur

Nous avons déjà vu la commande `print('message')` qui permet d'écrire un message sur le terminal. Il s'agit en fait d'une fonction prenant un nombre variable d'arguments et les affichant successivement dans la fenêtre de l'interpréteur. Les arguments passés peuvent être de n'importe quel type.

```
1 >>> print(10, 'text', 4.2)
2 10 text 4.2
```

Par défaut, les valeurs sont séparées par une espace. Il est toutefois possible de choisir un autre séparateur en ajoutant un argument `sep='xxx'` après tous les autres.

```
1 >>> print(10, 'text', 4.2, sep=' - ')
2 10 - text - 4.2
```

Contrairement aux autres arguments, il est ici nécessaire de préciser un nom (`sep`) pour que Python fasse la différence avec les autres valeurs : il ne doit pas considérer `' - '` comme une valeur à afficher en plus des autres mais comme le séparateur entre ces valeurs. On parle alors d'argument nommé.

Sachez aussi que l'on peut appeler la fonction `print` sans lui passer aucun argument. À quoi cela peut bien servir ? Juste à afficher une ligne vide. Cela revient à appeler `print` avec une chaîne vide.

```
1 >>> print()
2
3 >>> print('')
```

À l'inverse de `print`, il existe aussi une fonction `input` pour lire une chaîne de caractères depuis le terminal, selon ce qui est entré par l'utilisateur.

```
1 >>> input()
2 coucou
3 'coucou'
```

Après avoir entré la commande `input()`, on est invité à écrire une ligne de texte en terminant par un retour à la ligne (**Entrée**). Cette ligne est ensuite renvoyée, sous forme d'une chaîne de caractères, par la fonction `input`.

i

Le texte qui s'affiche sous la ligne `>> input()` dans l'exemple au-dessus est donc le texte entré dans le terminal par l'utilisateur. Je vous invite alors à essayer ces exemples chez vous pour bien voir comment ils se comportent. Vous pouvez y entrer ce que bon vous semble.

`input` prend aussi un argument optionnel permettant d'afficher un message juste avant de demander la saisie, comme dans l'exemple suivant.

```
1 >>> name = input('Quel est ton nom ? ')
2 Quel est ton nom ? entwanne
3 >>> print("Tu t'appelles", name)
4 Tu t'appelles entwanne
```

On comprend ainsi tout l'intérêt des variables. Jusqu'ici nous ne manipulions que des données connues du programme et les variables pouvaient sembler futiles. Mais elles vont maintenant nous servir à stocker et traiter des données venant de l'extérieur, inconnues au lancement du programme.

II.4.2. Conversions de types

Comme indiqué, `input` renvoie toujours une chaîne de caractères. Comment faire alors pour demander à l'utilisateur un nombre afin de l'utiliser dans un calcul ?

Il y a pour cela des mécanismes pour convertir (dans la mesure du possible) une valeur d'un type vers un autre. Je n'ai pour le moment présenté les types que comme des catégories regroupant des valeurs, mais ils ont en fait une existence propre en Python.

Les nombres entiers correspondent ainsi au type `int` (pour *integer*, entier), les nombres à virgule au type `float` (flottant) et les chaînes de caractère au type `str` (pour *string*, chaîne).

Chacun de ces types peut être vu et utilisé comme une fonction permettant de convertir des données vers ce type.

```
1 >>> int(4.2)
2 4
3 >>> float(4)
4 4.0
5 >>> str(4)
6 '4'
7 >>> int('10')
```

II. Manipuler des données

```
8 10
```

On voit dans ce dernier exemple que `'10'` et `10` sont des valeurs de types différents, la première est une chaîne de caractères et la seconde un nombre. Il ne s'agit donc pas de la même chose, on ne peut pas exécuter les mêmes opérations sur les deux.

```
1 >>> 10 + 1
2 11
3 >>> 10 + '1'
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   TypeError: unsupported operand type(s) for +: 'int' and 'str'
7 >>> '10' + '1'
8 '101'
```

Ainsi, pour en revenir à la demande initiale, afin traiter une entrée de l'utilisateur comme un nombre, il convient donc de convertir en `int` le retour d'`input`.

```
1 >>> n = int(input('Choisis un nombre : '))
2 Choisis un nombre : 5
3 >>> print('Le double de', n, 'vaut', n * 2)
4 Le double de 5 vaut 10
```

Cependant, toute valeur n'est pas convertible d'un type vers un autre, par exemple la chaîne de caractères `'toto'` ne correspond à aucun nombre. Lorsque la conversion est impossible, on verra survenir lors de l'appel une erreur explicitant le problème.

```
1 >>> int('toto')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ValueError: invalid literal for int() with base 10: 'toto'
```

II.5. Objets et méthodes

Introduction

Avant d'aller plus loin dans les interactions avec Python, j'aimerais que l'on prenne un peu de temps pour regarder de plus près la structure des données en Python.

II.5.1. Objets

Python est ce que l'on appelle un langage orienté objet (ou simplement langage objet). C'est-à-dire que toutes les valeurs que l'on manipule sont des objets : les entiers, les flottants ou les chaînes de caractères sont des objets.

Les objets sont définis comme des entités sur lesquels il est possible d'exécuter des actions (des opérations). Ils répondent ainsi à une interface qui définit quelles actions sont disponibles pour quels objets, c'est cette interface qu'on appelle le type.

Qu'est-ce que cela apporte ? C'est une manière de concevoir la structure d'un programme, de modéliser les interactions entre les valeurs. Il s'agit de définir les valeurs non pas selon ce qu'elles contiennent (des nombres, du texte) mais selon leur comportement : pouvoir être additionnées, pouvoir être affichées à l'écran, etc.

II.5.2. Méthodes

Les actions sur les objets sont plus généralement appelées des méthodes.

Elles sont très similaires aux fonctions, si ce n'est qu'elles appartiennent à un type et donc s'appliquent sur des objets en particulier (sur les objets de ce type).

Pour appeler une méthode sur une valeur, on fait suivre cette valeur d'un point puis du nom de la méthode, et enfin d'une paire de parenthèses comme pour les appels de fonction. On récupère de la même manière la valeur de retour de la méthode lors de l'évaluation de l'expression.

Par exemple la méthode `strip` du type `str` permet de renvoyer la chaîne de caractères en retirant les espaces présents au début et à la fin.

```
1 >>> ' hello '.strip()
2 'hello'
```

Pour que l'expression ait un sens, il faut bien sûr que la méthode existe pour cet objet. On obtient une erreur dans le cas contraire.

```
1 >>> 'hello'.toto()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
```

```
4 AttributeError: 'str' object has no attribute 'toto'
```

II.5.3. Méthodes des chaînes de caractères

Les chaînes de caractères possèdent d'autres méthodes utiles dont voici un bref aperçu. Nous en découvrirons encore bien d'autres dans la suite de ce cours.

II.5.3.1. `strip`

La méthode `strip` vue précédemment retire les espaces aux extrémités de la chaîne, mais n'affecte pas ceux qui se trouvent au milieu.

```
1 >>> ' hello world '.strip()
2 'hello world'
```

Il est possible d'appliquer la méthode sur une variable si celle-ci est assignée à une chaîne de caractères. Ou sur toute autre expression s'évaluant comme une chaîne de caractères, des parenthèses pouvant alors être nécessaires pour changer la priorité de l'opération.

```
1 >>> text = ' hello world '
2 >>> text.strip()
3 'hello world'
4 >>> input().strip()
5 coucou
6 'coucou'
7 >>> (' to' * 3).strip()
8 'to to to'
```

Cette méthode, tout comme les autres qui suivent, renvoie une nouvelle chaîne de caractères modifiée. Elle n'affecte jamais directement la chaîne sur laquelle elle est appliquée.

```
1 >>> text.strip()
2 'hello world'
3 >>> text
4 ' hello world '
```

II.5.3.2. `capitalize` et `title`

`capitalize` est une méthode qui permet de passer en majuscule le premier caractère de la chaîne (si c'est une lettre) et en minuscules tous les autres.

```
1 >>> 'coucou'.capitalize()
2 'Coucou'
```

II. Manipuler des données

```
3 >>> 'COUCOU'.capitalize()
4 'Coucou'
```

Semblable à `capitalize`, `title` effectue ce traitement sur tous les mots de la chaîne de caractères.

```
1 >>> 'bonjour à tous'.capitalize()
2 'Bonjour à tous'
3 >>> 'bonjour à tous'.title()
4 'Bonjour À Tous'
```

II.5.3.3. `upper` et `lower`

Il s'agit ici de passer la chaîne entière en majuscules ou en minuscules.

```
1 >>> 'CoUcOu'.upper()
2 'COUCOU'
3 >>> 'CoUcOu'.lower()
4 'coucou'
```

II.5.3.4. `index`

La méthode `index` permet de trouver un caractère dans la chaîne et d'en renvoyer la position. Il s'agit donc du comportement réciproque de l'opérateur `[]`.

```
1 >>> text = 'abcdef'
2 >>> text.index('d')
3 3
4 >>> text[3]
5 'd'
```

À noter que si le caractère est présent plusieurs fois dans la chaîne, c'est la première position trouvée qui est renvoyée.

```
1 >>> 'abcabc'.index('b')
2 1
```

Et une erreur survient si le caractère n'est pas trouvé.

```
1 >>> 'abcdef'.index('g')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ValueError: substring not found
```


II.5.4. Attributs

Les objets sont caractérisés par leur type et les méthodes qui lui sont applicables, mais ce n'est pas tout. Chaque objet a une existence propre, qui le différencie des autres objets du même type, et des données annexes peuvent lui être associées.

Ces données annexes sont autant de valeurs qui peuvent décrire l'état interne des objets. On les appelle des attributs.

Ils sont peu utilisés sur les objets que nous manipulons jusqu'ici (nombres et chaînes de caractères) car ces objets correspondent à de la donnée brute et n'ont pas besoin de valeurs annexes.

Leur utilité peut alors ne pas sembler évidente pour le moment, mais elle le deviendra quand on manipulera des objets plus complexes, composés de différentes données.

Il n'empêche que nos objets de base possèdent tout de même quelques attributs. Par exemple, chaque objet Python est pourvu d'un attribut `__class__` (faites bien attention aux deux *underscores* de chaque côté) qui permet d'accéder à son type.

```
1 >>> 'hello'.__class__
2 <class 'str'>
3 >>> 1.4.__class__
4 <class 'float'>
```

Vous découvrez ainsi la syntaxe pour accéder à un attribut d'un objet : on fait suivre l'objet d'un point puis du nom de l'attribut. C'est donc similaire aux méthodes, qui fonctionnent sur le même principe.

On notera cependant une petite différence pour les nombres entiers : comme le point y a déjà une signification (précéder une potentielle partie décimale) il ne peut pas être utilisé tel quel pour accéder aux attributs.

On ne peut ainsi pas écrire `42.__class__` qui ne serait pas compris par Python, il faut alors entourer le nombre de parenthèses pour lever toute ambiguïté.

```
1 >>> 42.__class__
2 File "<stdin>", line 1
3     42.__class__
4         ^
5 SyntaxError: invalid syntax
6 >>> (42).__class__
7 <class 'int'>
```

Cette exception ne s'applique bien sûr qu'aux nombres littéraux, et pas aux variables qui référencent des nombres qui n'ont aucun problème d'ambiguïté à ce niveau.

```
1 >>> x = 42
2 >>> x.__class__
3 <class 'int'>
```

Les nombres entiers possèdent aussi deux attributs `numerator` (numérateur) et `denominator` (dénominateur) qui leur permettent d'être vus comme des fractions (et donc d'être utilisés dans un contexte où une fraction serait attendue, nous découvrirons ça plus tard). Comme il s'agit

II. Manipuler des données

de nombres entiers, le dénominateur sera toujours de 1.

```
1 >>> x.numerator
2 42
3 >>> x.denominator
4 1
```

II.6. TP : Combat en tour par tour

Introduction

Il est temps maintenant de mettre en pratique ce que l'on a vu jusqu'ici avec un premier TP.

II.6.1. Présentation générale du TP

Afin de nous accompagner dans l'apprentissage du Python, voici un projet que je vous propose de réaliser au long des chapitres. L'idée est d'améliorer ce projet au fur et à mesure avec les nouvelles compétences que vous aurez acquises.

L'objectif final du projet est de réaliser un système de combat au tour par tour entre deux monstres, à la manière de Pokémon. Les deux monstres seront alors dotés d'une barre de vie (PV) et d'attaques. Les attaques se caractérisent par le nombre de points de vie qu'elles infligent en dégâts.

À chaque tour de jeu, une attaque est sélectionnée pour chaque monstre et les dégâts correspondant sont infligés à l'adversaire. Un monstre est KO quand sa barre de vie est vide. Le survivant remporte le combat.

Dans ces TP tu seras accompagné par Pythachu, Pythard et Ponytha.



FIGURE II.6.1. – Attrapez-les tous !

Pour plus de facilité, le jeu se déroulera en mode texte dans le terminal. Nous allons y aller par étapes et ce TP constitue la première étape : on va ici chercher à réaliser un seul tour de jeu.

II.6.2. Initialisation du jeu

Tout d'abord, avant de faire un premier tour de combat, il faut procéder à l'initialisation du jeu. En effet, il nous faut connaître les monstres en jeu et leur nombre initial de points de vie. Pour cela on utilisera la fonction `input` afin de demander aux deux joueurs les différentes informations. Nous les conserverons dans des variables et les afficherons en début de partie à l'aide de la fonction `print`.

```
1 Entrez le nom du 1er joueur : pythachu
2 Et son nombre de PV : 50
3 Entrez le nom du 2ème joueur : pythard
4 Et son nombre de PV : 40
5
6 ++++++
7 + Pythachu (50 PV) affronte Pythard (40 PV) +
8 ++++++
```

Vous le voyez, j'ai ajouté un cadre autour du message à afficher. On peut faire ça facilement à l'aide des opérateurs `+` et `*` que nous avons vus pour les chaînes de caractères.

La fonction `len` vous sera utile aussi pour que la largeur du cadre s'adapte à la taille du texte. On peut voir que j'ai passé en majuscule la première lettre des noms, saurez-vous retrouver la méthode qui permet ça ? Aussi, il nous faudra penser à convertir les PV saisis pour les traiter en tant que nombres.

II.6.2.1. Solution

Cette première étape devrait se faire assez facilement, et je vous laisse revoir les chapitres précédents en cas de doutes.

Voici tout de même la solution que je propose à ce début d'exercice, à comparer avec la vôtre. Je la mets en balise `secret`, suivie de quelques explications.

👁 Contenu masqué n°1

Nous pouvons maintenant passer à la suite du TP.

II.6.3. Tour de jeu

Un tour de jeu se divise en deux manches, d'abord le premier monstre attaque le second, puis l'inverse.

Nous ne sommes pour l'instant pas en mesure de traiter une liste d'attaques, nous demanderons alors simplement aux joueurs d'entrer le nombre de dégâts qu'ils souhaitent infliger à l'adversaire. Comme précédemment, on utilisera pour ça la fonction `input` et la conversion dans le type voulu.

À partir de ces dégâts, on calculera alors le nombre de points de vie restants du monstre cible, afin de les afficher dans un récapitulatif. Nous utiliserons toutes les données recueillies pour fournir le plus d'informations possibles aux joueurs.

À la suite de nos deux manches, on pourra afficher un résumé de la partie.

II. Manipuler des données

Voici ce à quoi pourrait ressembler un tour de jeu.

```
1  Pythachu, combien de dégâts infligez-vous à Pythard ? 30
2
3  ++++++
4  + Pythachu attaque Pythard qui perd 30 PV +
5  + Pythard a maintenant 10 PV               +
6  ++++++
7
8  Pythard, combien de dégâts infligez-vous à Pythachu ? 15
9
10 ++++++
11 + Pythard attaque Pythachu qui perd 15 PV +
12 + Pythachu a maintenant 35 PV             +
13 ++++++
14
15 ++++++
16 + Résultat du combat : +
17 + Pythachu a 35 PV      +
18 + Pythard a 10 PV       +
19 ++++++
```

II.6.3.1. Solution

L’affichage correct du cadre autour des messages pourrait vous donner du fil à retordre, pensez à la fonction `max` pour en connaître la taille.

Je vous propose la solution suivante à cette deuxième partie, mais prenez d’abord le temps de compléter la vôtre.

👁 Contenu masqué n°2

Conclusion

Voilà ce qui conclut notre premier TP, j’espère qu’il vous a plu, même s’il est très limité pour le moment. Vous pouvez continuer à travailler dessus pour vous exercer et l’améliorer au fil du temps.

Nous y reviendrons au cours des prochains chapitres pour lui ajouter de nouvelles fonctionnalités.

Il serait possible de réaliser plusieurs tours de jeu en dupliquant la partie de code dédiée autant de fois que l’on voudrait voir de tours, mais ce n’est pas une bonne pratique. Nous verrons par la suite comment faire cela proprement.

Contenu masqué

Contenu masqué n°1

```
1 name1 = input('Entrez le nom du 1er joueur : ').capitalize()
2 pv1 = int(input('Et son nombre de PV : '))
3
4 name2 = input('Entrez le nom du 2ème joueur : ').capitalize()
5 pv2 = int(input('Et son nombre de PV : '))
6
7 print()
8
9 message = name1 + ' (' + str(pv1) + ' PV) affronte ' + name2 +
10 ' (' + str(pv2) + ' PV)'
11 print('+' * (len(message)+4))
12 print('+', message, '+')
```

- La méthode `capitalize`, appliquée directement sur le retour d'`input` nous permet de transformer un 'pytachu' entré en 'Pythachu'.
- Les points de vie sont convertis en nombres à l'aide d'appels à `int`.
- `print` peut s'utiliser sans arguments pour juste afficher une ligne vide et séparer les informations les unes des autres.
- Pour l'affichage du cadre, on commence par forger une variable `message` qui contient le message à afficher. Ça se fait aisément à l'aide de concaténations (+) entre nos différents bouts de texte.
- À partir de la taille du message, on peut alors afficher les lignes haute et basse du cadre. Mais attention : avec les marges, elles comprennent 4 caractères de plus que le message.
- Enfin, pour l'affichage du message à proprement parler, on peut juste utiliser les différents arguments de `print`, sans concaténation.

[Retourner au texte.](#)

Contenu masqué n°2

```
1 att1 = int(input(name1 + ', combien de PV infligez-vous à ' +
2 name2 + ' ? '))
3 print()
4
5 pv2 -= att1
6 msg1 = name1 + ' attaque ' + name2 + ' qui perd ' + str(att1) +
7 ' PV'
8 msg2 = name2 + ' a maintenant ' + str(pv2) + ' PV'
9 max_size = max(len(msg1), len(msg2))
10 msg1 += ' ' * (max_size - len(msg1))
11 msg2 += ' ' * (max_size - len(msg2))
```

II. Manipuler des données

```
11 print('+ ' * (max_size+4))
12 print('+', msg1, '+')
13 print('+', msg2, '+')
14 print('+ ' * (max_size+4))
15
16 print()
17
18 att2 = int(input(name2 + ', combien de PV infligez-vous à ' +
19               name1 + ' ? '))
20
21 print()
22 pv1 -= att2
23 msg1 = name2 + ' attaque ' + name1 + ' qui perd ' + str(att2) +
24       ' PV'
25 msg2 = name1 + ' a maintenant ' + str(pv1) + ' PV'
26 max_size = max(len(msg1), len(msg2))
27 msg1 += ' ' * (max_size - len(msg1))
28 msg2 += ' ' * (max_size - len(msg2))
29 print('+ ' * (max_size+4))
30 print('+', msg1, '+')
31 print('+', msg2, '+')
32 print('+ ' * (max_size+4))
33
34 print()
35 msg1 = 'Résultat du combat : '
36 msg2 = name1 + ' a ' + str(pv1) + ' PV'
37 msg3 = name2 + ' a ' + str(pv2) + ' PV'
38 max_size = max(len(msg1), len(msg2), len(msg3))
39 msg1 += ' ' * (max_size - len(msg1))
40 msg2 += ' ' * (max_size - len(msg2))
41 msg3 += ' ' * (max_size - len(msg3))
42 print('+ ' * (max_size+4))
43 print('+', msg1, '+')
44 print('+', msg2, '+')
45 print('+', msg3, '+')
46 print('+ ' * (max_size+4))
```

- On appelle `input` avec un message formaté à l'aide de concaténations, on prend soin d'en convertir le retour en `int`.
- Ce nombre de dégâts est ensuite utilisé pour décrémenter les PV de l'ennemi.
- Pour l'affichage du cadre, celui-ci contient maintenant deux lignes différentes. Il faut alors calculer la taille maximale (`max_size`) à l'aide de la fonction `max` pour connaître la taille des lignes haute et basse.
- La longueur maximale sert aussi à calculer les marges pour que nos deux lignes s'intègrent correctement dans le cadre, en ajoutant autant d'espaces que besoin. On n'a pas peur pour cela de multiplier notre chaîne ' ' par un nombre négatif.
- On remarque pas mal de répétitions dans le code, ce n'est pas idéal et on verra comment y remédier dans un prochain chapitre.

[Retourner au texte.](#)

Troisième partie

Des programmes moins déterminés

Introduction

Interagir avec l'utilisateur, c'est bien, mais il serait encore mieux de pouvoir réagir différemment suivant ce qui est entré. C'est pourquoi nous allons maintenant voir comment rendre l'exécution de nos programmes moins linéaire.

III.1. Les conditions (if/elif/else)

Introduction

Une première étape pour aller vers plus d'interactivité est d'ajouter à notre programme des conditions. Les conditions vont nous permettre d'effectuer une action ou un autre suivant la valeur d'une expression. Par exemple «affiche "gagné" si l'utilisateur a entré le bon nombre».

III.1.1. Test d'égalité

Nous avons vu différents opérateurs arithmétiques mais il est maintenant temps de nous intéresser à une nouvelle catégorie : les opérateurs de comparaison. À commencer par l'opérateur d'égalité, noté `==`.

Cet opérateur appliqué à deux valeurs renvoie un état vrai (`True`) ou faux (`False`) indiquant si les valeurs sont égales ou non.

```
1 >>> 1 == 1
2 True
3 >>> 1 == 2
4 False
5 >>> 2 == 2
6 True
```

Le test d'égalité fonctionne pour tous les types de données et quelles que soient les expressions.

```
1 >>> 3 + 5 == 2 * 4
2 True
3 >>> word = 'abc'
4 >>> word == 'ab' + 'c'
5 True
6 >>> word == 'ab' + 'cd'
7 False
```



Attention cependant aux comparaisons avec des flottants. Si vous vous souvenez, on avait vu que les nombres flottants pouvaient comporter des erreurs d'arrondis.

Il peut ainsi arriver qu'une égalité entre flottants que l'on pense vraie ne le soit en fait pas, en raison de ces arrondis.



```
1 >>> 0.1 + 0.2 == 0.3
2 False
3 >>> 0.1 + 0.2 == 0.30000000000000004
4 True
```

De manière générale, évitez donc les tests d'égalité entre nombres flottants, nous verrons dans un prochain chapitre ce que l'on peut faire à la place.

Il est aussi possible de comparer des valeurs de types différents, mais le résultat sera souvent faux car des valeurs de types différents sont généralement considérées comme différentes (exception faite pour les nombres entiers et flottants).

```
1 >>> word == 2
2 False
3 >>> '2' == 2
4 False
5 >>> 2.0 == 2
6 True
```

Ainsi l'objectif est maintenant d'exécuter une action uniquement si un test d'égalité (une condition) est vérifié, c'est là qu'interviennent les blocs conditionnels !

III.1.2. Bloc conditionnel

Une condition en Python correspond à un bloc `if`, traduction anglaise du mot «si». Un bloc est un élément de syntaxe que nous n'avons pas encore vu jusqu'ici : il s'agit de plusieurs lignes de code réunies au sein d'une même entité logique.

Un bloc conditionnel est introduit à l'aide du mot-clé `if` suivi d'une expression et d'un signe `:`. Le contenu du bloc est constitué des lignes qui suivent, qui doivent être indentées par rapport à l'ouverture du bloc, c'est-à-dire décalées vers la droite avec des espaces pour les démarquer. On utilise conventionnellement 4 espaces.

Le contenu du bloc ne sera exécuté que si l'expression du `if` est évaluée à «vrai» (`True`).

```
1 if 2 == 2:
2     print('> Nous sommes dans le bloc conditionnel')
3     print('> Ici encore')
4
5 print('Nous sommes en dehors du bloc')
```

Ainsi le code précédent se lit :

- Si 2 est égal à 2, afficher «Nous sommes dans le bloc conditionnel» et «Ici encore».
- Dans tous les cas afficher «Nous sommes en dehors du bloc».

Et s'exécute comme suit.

III. Des programmes moins déterminés

```
1 > Nous sommes dans le bloc conditionnel
2 > Ici encore
3 Nous sommes en dehors de tout bloc
```

Listing 6 – Exécution du programme

Comme on le voit, un bloc prend fin dès la première ligne qui n'est pas indentée.



Pour cet exemple comme pour ceux qui suivront, je vous conseille d'utiliser un fichier Python plutôt que l'interpréteur interactif qui gère assez mal les problématiques d'indentation. J'y reviens juste après.

Lorsque la condition est fausse, le contenu du bloc `if` n'est jamais exécuté et on passe directement à la suite du programme.

```
1 if 1 == 2:
2     print("Cette ligne n'est jamais exécutée")
3
4 print('Cette ligne est en dehors du bloc')
```

Qui donne à l'exécution :

```
1 Cette ligne est en dehors du bloc
```

Listing 7 – Exécution du programme

Mais les exemples qui précèdent ont peu d'intérêt car les conditions sont fixées et ont donc toujours la même valeur. Il pourrait être intéressant par exemple d'interagir avec l'utilisateur à l'aide d'un `input`.

```
1 nbr = int(input('Devinez le nombre secret : '))
2
3 if nbr == 42:
4     print('Bravo, vous avez trouvé le nombre mystère !')
5
6 print('Relancez le programme pour une nouvelle partie')
```

On peut alors exécuter le programme plusieurs fois pour tester nos réponses, et avoir une exécution différente selon ce que l'on saisit.

```
1 Devinez le nombre secret : 10
2 Relancez le programme pour une nouvelle partie
```

Listing 8 – Première exécution

III. Des programmes moins déterminés

```
1 Devinez le nombre secret : 42
2 Bravo, vous avez trouvé le nombre mystère !
3 Relancez le programme pour une nouvelle partie
```

Listing 9 – Seconde exécution

III.1.2.1. Interpréteur interactif

L'interpréteur interactif peut parfois poser problème quand on utilise des blocs. Il demande en effet de laisser une ligne vide après chaque bloc (ce qui n'est pas nécessaire autrement), sans quoi vous obtiendrez une erreur de syntaxe.

```
1 >>> if 2 == 2:
2 ...     print('Gagné')
3 ...     print('Fin')
4     File "<stdin>", line 3
5         print('Fin')
6         ^
7 SyntaxError: invalid syntax
```

On remarque cela aux caractères utilisés par le prompt : quand nous sommes en dehors de tout bloc, les caractères `>>` sont utilisés. Mais une fois dans un bloc, ce prompt se transforme en `...`, signifiant que l'interpréteur attend d'autres lignes à ajouter au bloc.

Tout ce qui est tapé derrière un `...` est donc considéré par l'interpréteur interactif comme appartenant toujours au même bloc, ce qui provoque une erreur de syntaxe lorsque l'indentation est absente.

Une ligne vide permet de demander à l'interpréteur de sortir du bloc, qui serait alors exécuté immédiatement avant de passer à la suite.

```
1 >>> if 2 == 2:
2 ...     print('Gagné')
3 ...
4 Gagné
5 >>> print('Fin')
6 Fin
```

Dans un fichier, la première syntaxe est parfaitement valide, puisque ce sont les tabulations uniquement qui délimitent les blocs.

```
1 if 2 == 2:
2     print('Gagné')
3 print('Fin')
```

De la même manière, il est impossible d'avoir une ligne vide au milieu d'un bloc conditionnel dans l'interpréteur interactif, alors que cette syntaxe est valide en Python.

```
1 if 2 == 2:
2     print('Gagné')
3
4     print('Tu es trop fort')
```

i

Ces limitations peuvent être très gênantes et c'est pourquoi l'interpréteur interactif est déconseillé pour des codes complexes. Il reste toutefois très utile pour tester rapidement un petit bout de code.

III.1.2.2. Blocs sur une ligne

Il faut relever une exception au fait que le contenu d'un bloc conditionnel soit toujours indenté. Si un bloc se compose d'une seule ligne, il est possible de faire suivre cette ligne directement après le `:` du `if`, sans retour à la ligne ni indentation.

```
1 if nbr == 42: print('Bravo, vous avez trouvez le nombre !')
```

Cette forme est à déconseiller car elle fait perdre en lisibilité, mais elle reste néanmoins utile pour des cas particuliers comme une vérification rapide avec `python -c`.

```
1 % python -c "if 2 * 21 == 42: print('Bravo')"
```

```
2 Bravo
```

III.1.3. Et sinon ?

Nous avons vu quoi faire quand une condition était vraie, mais ce n'est pas le seul cas qui nous intéresse. Une condition est en effet soit vraie soit fausse, et un traitement particulier doit pouvoir être apporté à ce deuxième cas de figure.

Python fournit pour cela le bloc `else` («sinon») qui se place directement après un bloc `if`. Aucune expression n'est nécessaire derrière le mot-clé `else` (pas de condition à préciser), le signe `:` reste néanmoins obligatoire pour introduire le bloc.

Le contenu du bloc `else` sera exécuté si et seulement si la condition du `if` est fausse.

```
1 secret = 42
2 nbr = int(input('Devinez le nombre secret : '))
3
4 if nbr == secret:
5     print('Bravo, vous avez trouvez le nombre !')
6 else:
7     print('Perdu, le nombre était', secret)
8
```

III. Des programmes moins déterminés

```
9 print('Relancez le programme pour une nouvelle partie')
```

Le programme précédent se lit comme suit :

Le joueur entre un nombre.

- Si le nombre est égal à 42, afficher «Bravo [...]».
- Sinon, afficher «Perdu [...]».

Dans tous les cas, afficher «Relancez le programme [...]».

Avec nos blocs conditionnels nous avons chaque fois deux issues : soit la condition du `if` est vraie et nous entrons dans son bloc, soit elle est fausse et c'est le bloc `else` qui est exécuté. Il est en fait possible d'avoir plus d'options que cela en combinant plusieurs conditions, c'est-à-dire en testant une seconde condition quand la première est fausse. Plutôt que d'avoir un simple «si / sinon» nous pourrions avoir «si / sinon si / sinon».

Ce «sinon si» prend la forme du mot-clé `elif` (contraction de «else if» en anglais), qui s'utilise donc suivi d'une nouvelle expression conditionnelle.

```
1 secret = 42
2 nbr = int(input('Devinez le nombre secret : '))
3
4 if nbr == secret:
5     print('Bravo, vous avez trouvez le nombre !')
6 elif nbr == secret - 1:
7     print('Un peu plus...')
8 else:
9     print('Perdu, le nombre était', secret)
10
11 print('Relancez le programme pour une nouvelle partie')
```

Ainsi, dans le cas où `nbr` vaut `secret` nous afficherons «Bravo», s'il vaut `secret - 1` nous obtiendrons «Un peu plus» et nous aurons «Perdu» dans tous les autres cas.

Une structure conditionnelle peut contenir autant de blocs `elif` que nécessaire (contrairement au `else` qui ne peut être présent qu'une fois), pour tester différentes conditions à la suite.

```
1 secret = 42
2 nbr = int(input('Devinez le nombre secret : '))
3
4 if nbr == secret:
5     print('Bravo, vous avez trouvez le nombre !')
6 elif nbr == secret - 1:
7     print('Un peu plus...')
8 elif nbr == secret + 1:
9     print('Un peu moins...')
10 else:
11     print('Perdu, le nombre était', secret)
12
13 print('Relancez le programme pour une nouvelle partie')
```


III. Des programmes moins déterminés

Il faut bien noter qu'un bloc `elif` dépend du `if` et des autres `elif` qui le précèdent, son contenu ne sera donc exécuté que si toutes les conditions précédentes se sont révélées fausses. `if` étant le mot-clé qui introduit une structure conditionnelle, il doit être placé avant les `elif` / `else`. De même, `else` terminant cette structure, il se place à la suite de tous les `elif`. `elif` et `else` restent bien sûr optionnels, un bloc conditionnel peut ne contenir qu'un simple `if`. On peut aussi imaginer un `if` suivi de `elif` mais sans `else`.

```
1 secret = 'p4ssw0rd'
2 password = input('Entrez le mot de passe : ')
3
4 if password == '':
5     print('Veuillez saisir un mot de passe valide')
6 elif password == secret:
7     print('Authentification réussie')
```

III.1.4. Structures multiples

III.1.4.1. Enchaînement

Les exemples présents dans les sections qui précèdent montraient des structures conditionnelles seules : chacune est introduite par un `if`, peut comporter plusieurs clauses `elif` et peut être terminée par un `else`.

Mais dans un programme il est généralement nécessaire de tester différentes conditions et donc d'avoir plusieurs structures conditionnelles, c'est-à-dire plusieurs `if`.

Dans un fichier, on placera donc les blocs conditionnels les uns à la suite des autres, et Python comprendra qu'il s'agit d'une nouvelle structure chaque fois qu'il verra un `if`.

```
1 user = input("Entrez le nom d'utilisateur: ")
2 password = input("Entrez le mot de passe: ")
3
4 if user == 'admin':
5     print("Le compte administrateur est désactivé")
6
7 if password == '1234':
8     print("Mot de passe trop faible")
9 elif password == '4321':
10    print("C'est pas mieux")
```

Dans le code précédent, les deux structures conditionnelles sont indépendantes l'une de l'autre. Le deuxième `if` / `elif` est exécuté quelle que soit l'issue du premier `if`.

Il se lit de la manière suivante :

L'utilisateur entre un nom et un mot de passe.

- Si le nom d'utilisateur est «admin», afficher «Le compte administrateur est désactivé.»
- Si le mot de passe est «1234», afficher «Mot de passe trop faible».
- Sinon, si le mot de passe est «4321», afficher «C'est pas mieux».

III. Des programmes moins déterminés

Et comme on le voit, le `elif` se rapporte toujours au `if` qui le précède directement, il en est de même pour `else`.



Encore une fois, attention aux exemples de code qui pourraient ne pas fonctionner dans l'interpréteur interactif. Ce dernier demandera toujours de laisser une ligne vide entre deux blocs conditionnels distincts.

III.1.4.2. Imbrication

Une autre manière de combiner plusieurs blocs conditionnels consiste à les imbriquer / emboîter. Il est effectivement courant au sein d'un bloc `if` de vouloir tester une nouvelle condition pour effectuer un traitement particulier.

Pour rappel, Python délimite les blocs de code par leur indentation, c'est-à-dire les 4 espaces laissées en début de ligne. Quand il n'y a qu'un seul bloc, on ne constate qu'un niveau d'indentation.

Mais pour imbriquer une nouvelle condition sous une autre, il va nous falloir passer au niveau d'indentation suivant en ajoutant encore 4 espaces. Ainsi, Python compte le nombre d'espaces présentes en début de ligne pour déterminer dans quel bloc il se trouve.

Cela nous donne aussi une démarcation visuelle pour bien voir comment s'agencent nos blocs conditionnels.

```
1 quit = input('Voulez vous quitter le programme (oui/non) ? ')
2
3 if quit == 'oui':
4     confirm = input('Vous êtes sûr (oui/non) ? ')
5     if confirm == 'oui':
6         print('Fermeture en cours...')
7     else:
8         print('Décidez-vous !')
9 else:
10    print('Ok, on continue.')
```

Je vous invite à recopier le code qui précède dans un fichier et à l'exécuter en testant les 3 combinaisons possibles. On constate bien que la condition sur `confirm` n'est exécutée que lorsque `quit` vaut «oui», et que les `else` sont indentés au même niveau que les `if` auxquels ils se rapportent.

```
1 Voulez vous quitter le programme (oui/non) ? oui
2 Vous êtes sûr (oui/non) ? non
3 Décidez-vous !
```

Listing 10 – Exécution du programme

III.2. Expressions booléennes

Introduction

Au chapitre précédent nous avons appris à créer des conditions basées sur l'égalité entre deux valeurs. Mais l'égalité n'est pas l'unique opération conditionnelle possible et c'est ce que nous allons voir maintenant.

III.2.1. Opérations booléennes

III.2.1.1. Vocabulaire

Pour rappel, un test d'égalité peut s'évaluer à **True** (vrai) ou **False** (faux). Il s'agit de deux valeurs qui forment un nouveau type de données, le type booléen (**bool** en Python).

Ce nom provient de George Boole qui a introduit ce concept d'une valeur ne pouvant avoir que deux états, vrai ou faux.

C'est pourquoi on parle généralement d'opération ou d'expression booléenne pour qualifier une expression s'évaluant en un booléen.

Derrière les structures conditionnelles se cache aussi la notion de prédicat. C'est le nom que l'on donne à l'expression booléenne qui conditionne la suite de l'exécution du bloc.

Typiquement, un prédicat peut correspondre à une vérification d'une entrée utilisateur, pour s'assurer qu'un mot de passe est correct ou qu'un nombre se situe bien dans un certain intervalle.

III.2.1.2. Opérateurs

Outre l'égalité (**==**), plusieurs opérateurs de comparaison permettent d'obtenir des booléens. On trouve ainsi l'opérateur de différence, **!=**, qui teste si deux valeurs sont différentes l'une de l'autre.

Cet opérateur s'utilise de la même manière que l'égalité, sur des valeurs de tous types.

```
1 >>> 1 != 1
2 False
3 >>> 1 != 2
4 True
5 >>> 1 != 'abc'
6 True
7 >>> 1 != '1'
8 True
```

Sont aussi présents les opérateurs d'inégalités **<** et **>** pour tester les relations d'ordre :

— **a < b** teste si **a** est strictement inférieur à **b** ;

III. Des programmes moins déterminés

— `a > b` teste si `a` est strictement supérieur à `b`.

```
1 >>> 1 < 2
2 True
3 >>> 1 > 2
4 False
```

Cette fois-ci l'opération n'est possible qu'entre deux valeurs compatibles, c'est-à-dire ordonnables l'une par rapport à l'autre. Ce qui est par exemple le cas des entiers et des flottants.

```
1 >>> 5 < 5.1
2 True
3 >>> 1.9 > 3
4 False
```

Les chaînes de caractères sont ordonnables les unes par rapport aux autres, selon un ordre appelé «ordre lexicographique», une extension au classique ordre alphabétique.

Il est ainsi possible de tester les inégalités entre deux chaînes de caractères.

```
1 >>> 'renard' > 'loup'
2 True
3 >>> 'loup' < 'lama'
4 False
```

L'ordre lexicographique définit explicitement l'ordre de tous les caractères, selon la table unicode. Il faut savoir par exemple que bien que les lettres soient dans l'ordre alphabétique, les majuscules sont considérées comme inférieures aux minuscules, et les lettres accentuées supérieures aux autres.

```
1 >>> 'Loup' < 'lama'
2 True
3 >>> 'léopard' > 'loup'
4 True
```

Il n'est en revanche pas possible de comparer un nombre avec une chaîne de caractères, car aucune relation d'ordre n'existe entre ces deux types.

```
1 >>> 5 > '4'
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: '>' not supported between instances of 'int' and 'str'
```

Enfin ces deux opérateurs possèdent des variantes `<=` et `>=` correspondant aux opérations «inférieur ou égal» et «supérieur ou égal» en mathématiques.

```
1 >>> 1 < 1
2 False
3 >>> 1 <= 1
4 True
5 >>> 'abc' <= 'abc'
6 True
```

III.2.1.3. Priorités des opérateurs

La priorité entre opérateurs ne concerne pas que les opérations arithmétiques, les opérateurs de comparaison sont eux aussi concernés. Tous ces opérateurs possèdent la même priorité, qui se situe juste en-dessous de l'addition. Ainsi, toutes les opérations arithmétiques sont prioritaires sur les opérations de comparaison.

On a donc `3 + 5 == 2 * 4` qui est équivalent à `(3 + 5) == (2 * 4)`.

```
1 >>> 3 + 5 == 2 * 4
2 True
3 >>> (3 + 5) == (2 * 4)
4 True
```

Les opérations booléennes étant des expressions comme les autres, il est tout à fait possible d'en stocker le résultat dans des variables. Il est alors courant d'entourer l'expression de parenthèses pour bien la distinguer de l'opérateur `=` d'assignation.

```
1 >>> equal = (5 == 8)
2 >>> equal
3 False
4 >>> inferior = ('abc' < 'def')
5 >>> inferior
6 True
```

III.2.2. Booléens

Ainsi, le type `bool` est un type composé de seulement deux valeurs, `True` et `False`. Ces valeurs répondent à ce que l'on appelle [algèbre de Boole](#) ^[7], qui définit les opérations logiques possibles entre les booléens.

III.2.2.1. Algèbre de Boole

L'opération la plus élémentaire est la négation logique («NON») qui se note `not` en Python. C'est un opérateur unaire (qui ne prend qu'un seul opérande), qui consiste à calculer l'inverse du booléen : `True` devient `False` et inversement.

III. Des programmes moins déterminés

```
1 >>> not True
2 False
3 >>> not False
4 True
```

Il est courant de représenter les opérations booléennes sous forme de tables de vérité. C'est-à-dire de présenter un tableau associant à chaque valeur le résultat de l'opération.

Voici donc la table de vérité de l'opérateur `not` :

a	not a
True	False
False	True

La négation d'une égalité est alors la même chose que la différence.

```
1 >>> not 'abc' == 'def'
2 True
3 >>> 'abc' != 'def'
4 True
```

Mais il est aussi possible de combiner plusieurs booléens entre-eux, de différentes manières.

La première est la conjonction («ET») qui permet de tester si deux valeurs sont vraies. Avec `a` et `b` deux booléens, l'expression «a ET b» est vraie si et seulement si `a` est vrai et que `b` l'est aussi.

En Python, cet opérateur binaire (à deux opérandes) se note `and`.

```
1 >>> True and True
2 True
3 >>> True and False
4 False
5 >>> 'abc' == 'zzz' and 3 > 0
6 False
```

Et sa table de vérité est la suivante.

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

Enfin, l'autre opération que l'on trouve en Python sur les booléens est la disjonction («OU», soit `or` en Python). L'expression «a OU b» étant vraie si `a` est vrai ou que `b` l'est.

III. Des programmes moins déterminés

```
1 >>> True or False
2 True
3 >>> False or False
4 False
5 >>> 'abc' == 'zzz' or 3 > 0
6 True
```

On note que le «OU» est inclusif, ce qui peut se différencier de l'usage courant.

En effet quand on dit «a OU b» on a tendance à imaginer que c'est soit l'un soit l'autre (exclusif) mais pas les deux. En informatique on considère que «a OU b» est vraie aussi si **a** et **b** sont vrais.

```
1 >>> True or True
2 True
```

Voici donc la table de vérité du **or**.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

Ces opérateurs sont bien sûr composables les uns avec les autres pour former des expressions plus complexes. On utilisera généralement des parenthèses pour isoler les différentes opérations et ne pas avoir à se soucier de leur priorité (voir plus loin).

```
1 >>> (True or False) and not (True and False)
2 True
```

III.2.2.2. Conditions et priorités

Ces trois opérateurs peuvent donc s'utiliser au sein d'expressions booléennes pour introduire des blocs conditionnels et ainsi combiner en une seule clause plusieurs sous-conditions.

```
1 username = input("Nom d'utilisateur : ")
2 password = input("Mot de passe : ")
3
4 if username == 'admin' and password == 'nimda':
5     print('Vous êtes connecté')
6 else:
7     print('Échec de la connexion')
```

III. Des programmes moins déterminés

Il est à noter que `and`, `or` et `not` sont les opérateurs en Python ayant la plus faible priorité. Plus faible encore que les opérateurs de comparaison (`==`, `!=`, etc.).

C'est pourquoi l'expression dans le code qui précède est équivalente à `(username == 'admin') and (password == 'nimda')`.

Aussi, `not` est prioritaire sur `and` qui est lui-même prioritaire sur `or`, comme on peut le voir dans le code suivant.

```
1 >>> not True and False
2 False
3 >>> True or True and False
4 True
```

Mais ce comportement peut différer d'un langage de programmation à un autre, c'est pourquoi on utilisera toujours des parenthèses autour des sous-expressions booléennes combinant ces différents opérateurs, pour plus de clarté.

```
1 >>> (not True) and False
2 False
3 >>> True or (True and False)
4 True
```

III.2.2.3. Conversions implicites et explicites

Bien que nous n'ayons pour le moment utilisé de blocs `if` qu'avec des expressions booléennes, il faut savoir que ceux-ci acceptent n'importe quelle expression, par exemple ici avec un `int`.

```
1 >>> if 5 + 3 * 4:
2 ...     print('Ça marche')
3 ...
4 Ça marche
```

En fait, toute valeur Python est implicitement convertible en booléen, et c'est cette conversion qu'opère Python sur les expressions qu'il rencontre dans un bloc conditionnel.

Ainsi, le nombre zéro (`0` ou `0.0`) et la chaîne vide (`''`) s'évaluent à `False`. Alors que tous les autres nombres (même négatifs) et chaînes de caractères s'évaluent à `True`.

Cette facilité permet de simplifier certaines conditions, comme pour tester si une chaîne entrée n'est pas vide.

```
1 name = input('Bonjour, qui est tu ? ')
2
3 if name:
4     print('Bonjour', name)
5 else:
6     print('Erreur de saisie')
```

Dans cet exemple, `if name:` est équivalent à `if name != ''`, car une chaîne vide s'évaluera

III. Des programmes moins déterminés

toujours à `False`. On préférera donc généralement utiliser cette version raccourcie plutôt qu'ajouter une comparaison inutile.

Il reste bien sûr possible—quand cela est nécessaire—de convertir explicitement une valeur en booléen, en utilisant le type `bool` comme une fonction sur la valeur que l'on souhaite convertir. La conversion se fera selon les mêmes règles que celles décrites au-dessus.

```
1 >>> bool('hello')
2 True
3 >>> bool('')
4 False
5 >>> bool(-5.8)
6 True
7 >>> bool(0)
8 False
```

De la même manière, il n'est pas utile de comparer un booléen à `True` ou `False` dans une condition, ce qui ne fait que rallonger l'expression sans y apporter plus de sens. Avec `result` le résultat d'une opération booléenne (`result = (name == 'admin')`), on écrira donc simplement `if result: ...` et jamais `if result == True: ...`.

Et on écrira `if not result: ...` plutôt que `if result == False: ...`.

III.2.2.4. Comparaisons chaînées

Les opérateurs de comparaison que l'on a vus peuvent s'enchaîner afin de créer plus facilement des opérations booléennes entre plusieurs valeurs.

Par exemple, si l'on souhaite tester l'égalité entre trois valeurs `a`, `b` et `c`, on pourra écrire `a == b == c` plutôt que `a == b and b == c`.

```
1 >>> 10 == 10 == 10
2 True
3 >>> 10 == 10 == 5
4 False
```

Ou encore pour tester une inégalité, `0 < temp < 100` est plus simple à lire que `0 < temp and temp < 100`.

```
1 >>> 0 < 25 < 100
2 True
3 >>> 0 < -25 < 100
4 False
5 >>> 0 < 125 < 100
6 False
```

III.3. TP : Ajoutons des conditions à notre jeu

Introduction

Ces blocs conditionnels vont nous être d'une grande aide dans le développement de notre jeu de combat au tour par tour, puisque nous allons pouvoir avoir un programme vraiment interactif. Il sera alors possible de réagir différemment suivant les données saisies par l'utilisateur. Et pour commencer, nous allons introduire un choix d'attaque, chaque attaque ayant un nombre de dégâts infligés différent.

III.3.1. Proposer plusieurs attaques

L'idée maintenant va donc être d'avoir un nombre de dégâts pour chaque attaque, et de proposer à l'utilisateur l'une ou l'autre des attaques.

Ce qu'on voudrait c'est afficher une sorte de menu proposant les différentes attaques. Par exemple on pourrait utiliser le message suivant lors de l'input :

```
1 Quelle attaque voulez-vous utiliser ?
2 1. Charge (-20 PV)
3 2. Tonnerre (-50 PV)
```

Une condition permettrait ensuite de savoir quelle attaque a été choisie ('1' ou '2') et d'agir en conséquence en infligeant les dégâts à l'adversaire.

En bonus on pourrait même autoriser d'entrer le nom de l'attaque plutôt que son numéro, je vous laisse y réfléchir.

Notre jeu ne comporte encore qu'un seul tour, mais on pourrait aussi conclure la fin du tour en annonçant le vainqueur, à l'aide d'une condition sur le nombre de PV..

III.3.2. Solution

J'ai ici volontairement allégé le programme par rapport à la solution du précédent TP, en retirant tout ce qui avait trait au formatage des chaînes de caractères. Mais n'hésitez pas à reprendre votre programme précédent pour le compléter avec ces nouvelles fonctionnalités.

☞ Contenu masqué n°3

Et à l'utilisation, on a bien un programme de combat un peu plus dynamique.

III. Des programmes moins déterminés

```
1 Entrez le nom du 1er joueur : Pythachu
2 Et son nombre de PV : 100
3 Entrez le nom du 2ème joueur : Ponytha
4 Et son nombre de PV : 100
5
6 Pythachu affronte Ponytha
7
8 Pythachu quelle attaque voulez-vous utiliser ?
9 1. Charge (-20 PV)
10 2. Tonnerre (-50 PV)
11 > 2
12 Pythachu attaque Ponytha qui perd 50 PV
13 Ponytha quelle attaque voulez-vous utiliser ?
14 1. Charge (-20 PV)
15 2. Tonnerre (-50 PV)
16 > Charge
17 Ponytha attaque Pythachu qui perd 20 PV
18 Pythachu remporte le combat
```

Mais il est difficile avec le code actuel d'ajouter de nouvelles attaques et l'on voit encore beaucoup de répétitions dans ce code. Pas d'inquiétudes, nous corrigerons tout cela dans les chapitres qui viennent.

Contenu masqué

Contenu masqué n°3

```
1 name1 = input('Entrez le nom du 1er joueur : ').capitalize()
2 pv1 = int(input('Et son nombre de PV : '))
3
4 name2 = input('Entrez le nom du 2ème joueur : ').capitalize()
5 pv2 = int(input('Et son nombre de PV : '))
6
7 print()
8 print(name1, 'affronte', name2)
9 print()
10
11 menu = '''quelle attaque voulez-vous utiliser ?
12 1. Charge (-20 PV)
13 2. Tonnerre (-50 PV)'''
14
15 # Joueur 1
16
17 print(name1, menu)
18 att1 = input('> ').lower()
19
```

III. Des programmes moins déterminés

```
20 if att1 == '1' or att1 == 'charge':
21     damages = 20
22 elif att1 == '2' or att1 == 'tonnerre':
23     damages = 50
24 else:
25     print('Erreur de saisie')
26     damages = 0
27
28 pv2 -= damages
29 print(name1, 'attaque', name2, 'qui perd', damages, 'PV')
30
31 # Joueur 2
32
33 print(name2, menu)
34 att2 = input('> ').lower()
35
36 if att2 == '1' or att2 == 'charge':
37     damages = 20
38 elif att2 == '2' or att2 == 'tonnerre':
39     damages = 50
40 else:
41     print('Erreur de saisie')
42     damages = 0
43
44 pv1 -= damages
45 print(name2, 'attaque', name1, 'qui perd', damages, 'PV')
46
47 if pv1 == pv2:
48     print('Match nul')
49 elif pv1 > pv2:
50     print(name1, 'remporte le combat')
51 else:
52     print(name2, 'remporte le combat')
```

[Retourner au texte.](#)

III.4. Les listes

Introduction

Place à présent à un nouveau type de données, les listes, qui vont nous permettre de construire des valeurs plus complexes. Les listes vont en effet nous servir à composer plusieurs valeurs en une seule.

III.4.1. Des séquences de valeurs

Une liste en Python peut être vue comme une séquence de valeurs. Imaginez une simple ligne de tableau avec des cases, chaque case contenant une valeur.

5	3	2	8	6	7	3
---	---	---	---	---	---	---

Ceci est la représentation d'une liste de 7 nombres entiers. On la noterait en Python de la manière suivante :

```
1 numbers = [5, 3, 2, 8, 6, 7, 3]
```

On utilise donc des crochets pour délimiter la liste, et des virgules pour séparer les valeurs les unes des autres.

Chaque case de la liste est associée à une position (ou **index**). Ainsi la case en première position contient la valeur 5, celle en deuxième position contient la valeur 3, etc. L'ordre des éléments dans une liste est donc important, et celui-ci est libre (mes valeurs n'ont par exemple pas besoin d'être rangées en ordre croissant).

```
1 >>> [1, 2, 3]
2 [1, 2, 3]
3 >>> [2, 3, 1]
4 [2, 3, 1]
```

On note que la case en septième (dernière) position contient aussi la valeur 3. Une même valeur peut être présente dans la liste à plusieurs positions.

La liste peut être vue comme une généralisation des chaînes de caractères : là où la chaîne est une séquence de caractères, la liste peut contenir des valeurs de tous types. L'exemple précédent ne montre qu'une liste composée de nombres entiers (`int`), mais n'importe quelle valeur peut être contenue dans une liste.

III. Des programmes moins déterminés

```
1 >>> ['abc', 'def']
2 ['abc', 'def']
3 >>> [4.5, 1.8, -3.2]
4 [4.5, 1.8, -3.2]
```

Il faut voir les listes comme des ensembles de valeurs distinctes les unes des autres mais qui forment un tout. Elles sont le reflet même des listes de la vie courante : une liste de courses, une liste d'élèves, une liste de notes, etc.

```
1 courses = ['pain', 'œufs', 'lait', 'pâtes', 'tomates']
2 eleves = ['Julie', 'Martin', 'Sami', 'Natacha']
3 notes = [12, 9, 16, 13]
```

On peut aussi construire une liste composée de valeurs de types différents. On verra par la suite que l'important est d'avoir une manière unique de traiter l'ensemble des éléments.

```
1 notes = [12, 8.5, 16, 12.5]
2 items = ['salut', 42, True, 1.5]
```

Une liste peut aussi ne contenir aucun élément (liste vide), on la définit alors à l'aide d'une simple paire de crochets [].

Un autre cas particulier est celui des listes contenant un seul élément, où la virgule est facultative puisqu'il n'y a pas de valeurs à séparer.

```
1 >>> []
2 []
3 >>> [4]
4 [4]
5 >>> ['salut',]
6 ['salut']
```

Quand on initialise une liste avec beaucoup d'éléments, il arrive que la ligne de définition soit assez longue.

```
1 words = ['sur', 'zeste', 'de', 'savoir', 'vous', 'pouvez',
           'trouver', 'des', 'contenus', 'sur', 'des', 'sujets', 'variés']
```

Il est alors intéressant d'aérer le tout pour que ça devienne plus lisible. Python permet d'effectuer des retours à la ligne dans la définition d'une liste, entre les valeurs.

Attention, un retour à la ligne ne remplace pas la virgule séparant les valeurs, qui reste obligatoire.

On prendra l'habitude d'indenter les valeurs par rapport à la ligne d'ouverture de la liste.

III. Des programmes moins déterminés

```
1 words = [  
2     'sur',  
3     'zeste',  
4     'de',  
5     'savoir',  
6     'vous',  
7     'pouvez',  
8     'trouver',  
9     'des',  
10    'contenus',  
11    'sur',  
12    'des',  
13    'sujets',  
14    'variés',  
15 ]
```

Seule la dernière virgule, puisque suivie d'aucune valeur, est facultative. Je la laisse par commodité et pour ne pas faire de différences entre les lignes.

Une liste se définit aussi par le nombre d'éléments qu'elle contient, sa taille. Cette taille sera amenée à évoluer au cours du déroulement du programme, la liste pouvant gagner ou perdre des éléments suivant certaines opérations.

III.4.2. Opérations sur les listes

III.4.2.1. Opérations élémentaires

Tout comme les chaînes de caractères, les listes possèdent donc une taille. Là encore, il est possible de connaître cette taille à l'aide d'un appel à la fonction `len`.

```
1 >>> len(numbers)  
2 7  
3 >>> len(words)  
4 13
```

Comme pour les chaînes toujours, il est possible d'accéder aux éléments de la liste à l'aide de l'opérateur `[]` associé à une position. 0 correspondant à la première position, 1 à la deuxième, etc.

```
1 >>> numbers[4]  
2 6  
3 >>> print(words[8])  
4 contenus
```

Les index négatifs sont aussi acceptés.

III. Des programmes moins déterminés

```
1 >>> words[-2]
2 'sujets'
```

On peut tester l'égalité entre deux listes à l'aide des opérateurs `==` et `!=`. Deux listes sont égales si elles contiennent les mêmes valeurs dans le même ordre.

```
1 >>> [1, 2, 3] == [1, 2, 3]
2 True
3 >>> [1, 2, 3] == [3, 2, 1]
4 False
5 >>> [1, 2, 3] != [3, 2, 1]
6 True
```

Comme les chaînes de caractères, les listes sont aussi concaténables les unes aux autres, permettant de construire une grande liste en agrégeant des plus petites. De même qu'elles sont concaténables par multiplication avec un nombre entier.

```
1 >>> [1, 1, 2, 3] + [5, 8, 13] + [21]
2 [1, 1, 2, 3, 5, 8, 13, 21]
3 >>> ['ab', 'cd'] * 3
4 ['ab', 'cd', 'ab', 'cd', 'ab', 'cd']
```

En plus de ça, les listes possèdent aussi différentes méthodes, par exemple pour rechercher et compter les éléments :

- `index` renvoie la position d'une valeur dans la liste. Cette position correspond au premier élément trouvé (si la valeur est présente plusieurs fois), et la méthode produit une erreur si la valeur n'est pas trouvée.

```
1 >>> numbers.index(2)
2 2
3 >>> numbers.index(3)
4 1
5 >>> numbers.index(7)
6 5
7 >>> numbers.index(9)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 ValueError: 9 is not in list
11 >>> words.index('savoir')
12 3
```

- `count` compte et renvoie le nombre d'occurrences d'un élément dans la liste (donc 0 si l'élément n'est pas présent).


```
1 >>> numbers.count(3)
2 2
3 >>> numbers.count(8)
4 1
5 >>> numbers.count(9)
6 0
7 >>> words.count('des')
8 2
```

III.4.2.2. Mutabilité

Les listes sont des objets dits mutables, c'est-à-dire modifiables, ce qui n'est pas le cas des autres types de données que nous avons vus jusqu'ici. En effet, sur les précédentes données que nous manipulions, leur valeur ne pouvait pas changer une fois qu'elles avaient été définies.

Nous pouvions redéfinir une variable vers une nouvelle valeur (`a = 10`; `a += 1`), mais la valeur en question restait inchangée (`10` valait toujours `10`).

Sur les listes, nous pouvons par exemple librement remplacer certains éléments par d'autres, grâce à l'opérateur d'indexation (`[]`) couplé à une affectation (`=`).

```
1 >>> words = ['salut', 'les', 'amis']
2 >>> words[2] = 'copains'
3 >>> words
4 ['salut', 'les', 'copains']
```

Ici c'est bien la valeur même de la liste qui a été modifiée : on a altéré son contenu pour remplacer un élément, mais `words` est toujours la même liste.

On peut mettre cet état de fait en évidence si l'on a deux variables qui référencent la même liste.

```
1 >>> numbers = copy = [1, 2, 3, 4]
2 >>> numbers[0] = 10
3 >>> numbers
4 [10, 2, 3, 4]
5 >>> copy
6 [10, 2, 3, 4]
```



C'est d'ailleurs un comportement qui est souvent perçu comme une erreur par les débutants, mais il faut bien comprendre que `numbers` et `copy` sont deux étiquettes sur une même liste. Ainsi, une modification de `numbers` est également une modification de `copy`.

```
1 >>> numbers = copy = [1, 2, 3, 4]
```

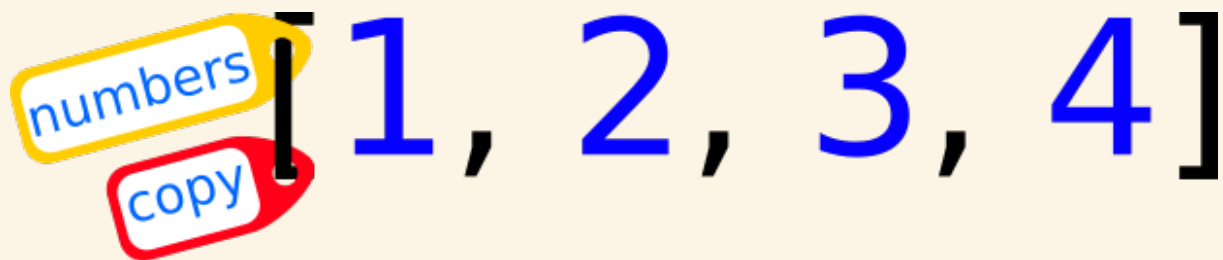


FIGURE III.4.1. – Deux étiquettes sur une même liste.

```
2 >>> numbers[0] = 10
```

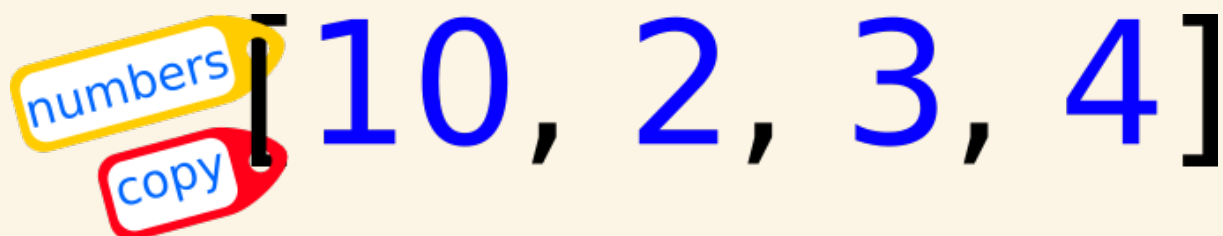


FIGURE III.4.2. – Les deux étiquettes sont affectées.

Nos listes étant modifiables, elles proposent aussi certaines opérations pour insérer ou supprimer des éléments.

La méthode `append` permet comme son nom l'indique d'ajouter un nouvel élément en fin de liste (à la dernière position), augmentant donc de 1 la taille de la liste.

```
1 >>> letters = ['a', 'b', 'c', 'd']
2 >>> len(letters)
3 4
4 >>> letters.append('e')
5 >>> letters
6 ['a', 'b', 'c', 'd', 'e']
7 >>> len(letters)
8 5
```

Plus généralement, on trouve la méthode `insert` qui permet d'insérer un élément à une position (un index) particulière dans la liste, décalant ainsi s'il y en a les éléments à sa droite d'un cran.

```
1 >>> letters.insert(0, 'à')
2 >>> letters
3 ['à', 'a', 'b', 'c', 'd', 'e']
4 >>> letters.insert(6, 'é')
5 >>> letters
6 ['à', 'a', 'b', 'c', 'd', 'e', 'é']
```

III. Des programmes moins déterminés

```
7 >>> letters.insert(3, 'ê')
8 >>> letters
9 ['à', 'a', 'b', 'ê', 'c', 'd', 'e', 'é']
10 >>> letters.insert(-2, 'ď')
11 >>> letters
12 ['à', 'a', 'b', 'ê', 'c', 'd', 'ď', 'e', 'é']
```

Comme vous le voyez, les index négatifs sont aussi acceptés. Si la position est plus grande que la taille de la liste, la valeur sera insérée la fin. De même, la valeur sera insérée au début pour une position négative dépassant la limite.

```
1 >>> letters.insert(20, 'f')
2 >>> letters
3 ['à', 'a', 'b', 'ê', 'c', 'd', 'ď', 'e', 'é', 'f']
4 >>> letters.insert(-50, 'â')
5 >>> letters
6 ['â', 'à', 'a', 'b', 'ê', 'c', 'd', 'ď', 'e', 'é', 'f']
```

La méthode `pop` sert quant à elle à supprimer un élément de la liste. Utilisée sans argument, elle en supprimera le dernier élément. La méthode renvoie l'élément qui vient d'être supprimé, ce qui permet de le conserver dans une variable par exemple.

```
1 >>> letters.pop()
2 'f'
3 >>> deleted = letters.pop()
4 >>> print(deleted, 'a été supprimée')
5 é a été supprimée
6 >>> letters
7 ['â', 'à', 'a', 'b', 'ê', 'c', 'd', 'ď', 'e']
```

Mais la méthode peut aussi être appelée avec une position en argument, pour supprimer une valeur à un index particulier.

```
1 >>> letters.pop(0)
2 'â'
3 >>> letters
4 ['à', 'a', 'b', 'ê', 'c', 'd', 'ď', 'e']
```

On notera aussi l'opérateur `del` permettant lui aussi de supprimer une valeur mais sans la renvoyer.

```
1 >>> del letters[3]
2 >>> letters
3 ['à', 'a', 'b', 'c', 'd', 'ď', 'e']
```



L'opérateur `del` est d'ailleurs un opérateur qui permet de supprimer une variable. `del foo` revient à désaffecter la variable `foo` qui n'existe alors plus dans la suite du programme.

```
1 >>> foo = 'abc'
2 >>> del foo
3 >>> foo
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   NameError: name 'foo' is not defined
```

`del` ne supprime pas la valeur à proprement parler qui peut toujours être référencée par une autre variable.

```
1 >>> foo = bar = [1, 2, 3]
2 >>> del foo
3 >>> bar
4 [1, 2, 3]
```

III.4.2.3. Slicing

Nous avons vu pour l'instant comment accéder facilement à un élément d'une liste à partir de son index, grâce à l'opérateur d'indexation (`[]`). Mais cet opérateur est plus puissant que cela et permet des utilisations plus avancées.

III.4.2.3.1. Obtenir une partie d'une liste

Il est en effet possible d'extraire plusieurs éléments en un seul appel, à l'aide d'une syntaxe particulière. Il s'agit de préciser entre les crochets une position de début et une position de fin, séparées par un signe `:`. On appelle cela le *slicing* (ou «découpage»).

La valeur renvoyée sera la liste des éléments compris entre ces deux positions (démarrant à la position de début et s'arrêtant juste avant la position de fin).

```
1 >>> numbers = [1, 1, 2, 3, 5, 8, 13, 21]
2 >>> numbers[1:4]
3 [1, 2, 3]
4 >>> numbers[0:7]
5 [1, 1, 2, 3, 5, 8, 13]
```

On voit bien que `numbers[1:4]` nous renvoie la liste des éléments d'index compris entre 1 et 3 (inclus). Ces opérations n'affectent pas la liste d'origine qui reste inchangée.

```
1 >>> numbers
2 [1, 1, 2, 3, 5, 8, 13, 21]
```

III. Des programmes moins déterminés

Une fois de plus, il est possible d'utiliser des index négatifs pour se positionner à partir de la fin de la liste.

```
1 >>> numbers[-5:-1]
2 [3, 5, 8, 13]
3 >>> numbers[1:-2]
4 [1, 2, 3, 5, 8]
```

Une autre facilité est que l'on peut omettre la position de début ou la position de fin. Sans position de début on considère que l'on part du début de la liste (index 0) et sans fin que l'on va jusqu'à la fin (index `len(numbers)`).

```
1 >>> numbers[3:]
2 [3, 5, 8, 13, 21]
3 >>> numbers[:-3]
4 [1, 1, 2, 3, 5]
```

Si l'on omet le début et la fin, on récupère une liste contenant tous les éléments de la liste d'origine.

```
1 >>> numbers[:]
2 [1, 1, 2, 3, 5, 8, 13, 21]
```

On peut enfin préciser une troisième valeur qui est le «pas» (par défaut de 1). Ce pas indique combien d'index on passe entre chaque élément. Un pas de 3 signifie que l'on ne considère qu'un élément sur 3.

Ainsi, `[1:8:3]` correspondra aux index 1, 4 et 7 (3 de différence entre chaque index)

```
1 >>> numbers[1:8:3]
2 [1, 5, 21]
```

Ou encore `[::2]` permettra d'extraire un élément sur deux de la liste initiale. En effet cela permet d'extraire l'élément d'index 0, puis 2, puis 4, etc.

```
1 >>> numbers[::2]
2 [1, 2, 5, 13]
```

Le pas est calculé à partir de l'index de départ, le résultat sera donc différent avec `[1::2]` qui considérera en premier l'élément d'index 1, puis 3, puis 5, etc.

```
1 >>> numbers[1::2]
2 [1, 3, 8, 21]
```

III. Des programmes moins déterminés

III.4.2.3.2. Modifier une partie d'une liste

Voilà pour ce qui est des accès en lecture, mais ces opérations sont aussi possibles pour la modification.

```
1 >>> numbers[:2] = [2, 0]
2 >>> numbers
3 [2, 0, 2, 3, 5, 8, 13, 21]
```

La liste que l'on assigne n'a pas besoin de faire la même taille que le nombre d'éléments concernés par le *slicing*, ce qui peut alors modifier la longueur de la liste d'origine.

```
1 >>> numbers[-1:] = [21, 34, 55]
2 >>> numbers
3 [2, 0, 2, 3, 5, 8, 13, 21, 34, 55]
4 >>> numbers[1:5] = []
5 >>> numbers
6 [2, 8, 13, 21, 34, 55]
```

Et ces opérations concernent aussi l'opérateur `del`.

```
1 >>> del numbers[1:-1]
2 >>> numbers
3 [2, 55]
```

Enfin, l'opération de *slicing* (en lecture seulement) est aussi disponible sur les chaînes de caractères, renvoyant donc une chaîne composée des caractères aux positions comprises dans l'intervalle..

```
1 >>> 'pouetpouet'[3:-2]
2 'etpou'
```

Pour plus d'informations sur le *slicing* en Python, je vous invite à découvrir ce tutoriel : [Les slices en Python](#) ↗ .

III.4.3. Listes à plusieurs dimensions

Je présentais en introduction les listes comme des séquences, des lignes d'éléments. L'analogie est bonne, d'autant que nos listes précédentes ne contenaient que des types de données simples : nombres ou chaînes de caractères.

Mais les listes peuvent contenir toutes sortes de données, même des plus complexes comme... d'autres listes.

```
1 >>> items = [1, 2, [3, [4]]]
2 >>> items
```

III. Des programmes moins déterminés

```
3 [1, 2, [3, [4]]]
```

Pour accéder aux éléments des sous-listes, on pourra simplement chaîner les opérateurs `[]`.

```
1 >>> items[2][1][0]
2 4
3 >>> items[2][0] = 5
4 >>> items
5 [1, 2, [5, [4]]]
```

Quand une liste est composée uniquement de sous-listes, elle peut alors prendre la forme d'un tableau. Comme ici avec une liste représentant un plateau de morpion.

```
1 morpion = [
2     ['x', ' ', ' '],
3     ['o', 'o', ' '],
4     ['x', ' ', ' '],
5 ]
```

Que l'on peut représenter sous la forme du tableau suivant.

	x		
	c	c	
	x		

Il s'agit ici d'un tableau à deux dimensions (lignes et colonnes). Mais les listes n'ont pas de limite et l'on pourrait alors voir d'autres subdivisions s'il y avait un niveau supplémentaire de listes.

III.4.3.1. Problème de la multiplication

?

Je vous parlais de l'opérateur de multiplication des listes pour les concaténer, mais que se passe-t-il si on l'utilise sur des listes à plusieurs dimensions ?

Eh bien ça ne fonctionne pas comme prévu !

En effet cet opérateur ne crée pas de copies mais duplique les références à une même valeur. La même sous-liste est alors répétée plusieurs fois dans la liste, provoquant des comportements inattendus en cas de modifications.

```
1 >>> grid = [[1, 2, 3]] * 2
2 >>> grid
3 [[1, 2, 3], [1, 2, 3]]
```

III. Des programmes moins déterminés

```
4 >>> grid[0].append(4)
5 >>> grid
6 [[1, 2, 3, 4], [1, 2, 3, 4]]
```

Le code précédent étant en fait équivalent à :

```
1 >>> line = [1, 2, 3]
2 >>> grid = [line, line]
3 >>> grid
4 [[1, 2, 3], [1, 2, 3]]
5 >>> line.append(4)
6 >>> grid
7 [[1, 2, 3, 4], [1, 2, 3, 4]]
```

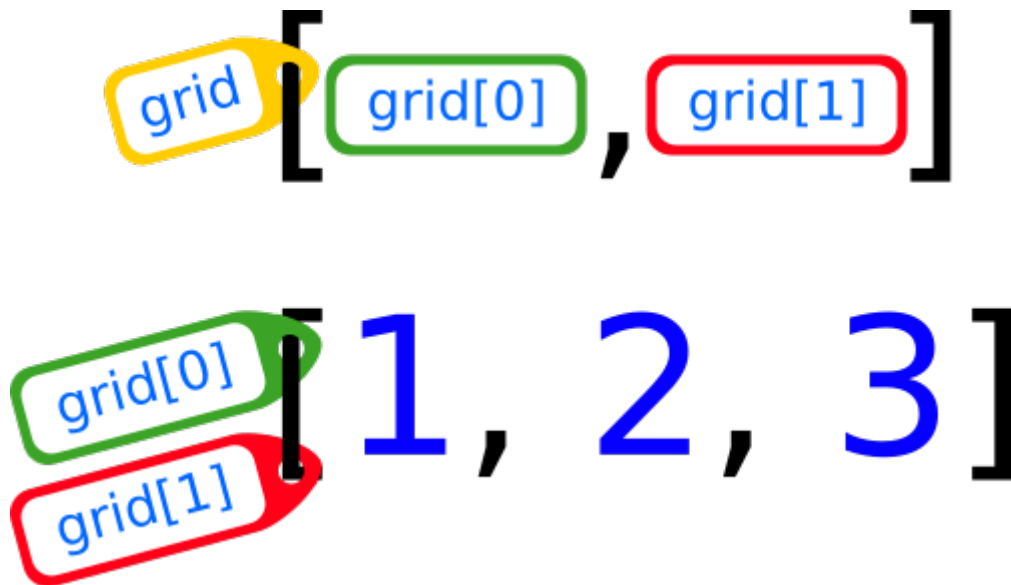


FIGURE III.4.3. – Étiquettes dupliquées entre les lignes.

Ce comportement de duplication des références n'est pas propre aux listes multi-dimensionnelles. Un code tel que `[0] * 10` duplique aussi 10 fois la référence à la valeur `0`, mais cela ne pose pas de problème particulier car les nombres ne sont pas des valeurs modifiables. Le comportement apparaît donc problématique dans le cas des sous-listes en raison de leur mutabilité.

Nous verrons dans le chapitre prochain comment contrer ce problème en construisant nos listes itérativement, en attendant je vous conseille de simplement ne pas utiliser la multiplication dans des cas comme celui-ci.

```
1 >>> grid = [[1, 2, 3], [1, 2, 3]]
2 >>> grid[0].append(4)
3 >>> grid
4 [[1, 2, 3, 4], [1, 2, 3]]
```


III.5. Itérer sur nos listes avec des boucles **for**

Introduction

On sait maintenant représenter une liste de valeurs. Mais vous pourrez me demander : quel intérêt de stocker 10 valeurs dans une liste plutôt que d'avoir 10 variables différentes ?

Et la réponse vient avec ce chapitre : grâce aux listes, on va pouvoir appliquer un même traitement sur toutes les valeurs, sans répéter de code.

III.5.1. Le bloc **for**

Une liste possède donc un nombre «indéterminé» de valeurs—on connaît sa taille au moment de l'exécution mais pas quand on écrit le code. Pour réaliser un traitement (par exemple afficher un message) pour chacune des valeurs de la liste, il nous faudrait alors pouvoir la parcourir d'élément en élément.

En programmation ce genre de construction s'appelle une boucle, soit un bloc de code qui sera répété un certain nombre de fois. Ici un bloc exécuté pour chaque élément de notre liste.

On a pour cela en Python le bloc **for** qui permet de parcourir une liste, d'itérer sur ses éléments. Sa syntaxe est la suivante :

```
1 for element in values:  
2     ...
```

values étant ici notre liste, et **element** une variable qui sera successivement (à chaque tour de boucle) assignée à chaque élément de la liste.

for introduit un bloc, la ligne se termine donc par un **:** et est suivie d'un bloc indenté.

```
1 >>> numbers = [1, 1, 2, 3, 5, 8, 13]  
2 >>> for elem in numbers:  
3     ...     print('Nombre actuel :', elem)  
4     ...  
5 Nombre actuel : 1  
6 Nombre actuel : 1  
7 Nombre actuel : 2  
8 Nombre actuel : 3  
9 Nombre actuel : 5  
10 Nombre actuel : 8  
11 Nombre actuel : 13
```

On peut voir l'itération comme un curseur qui se déplace le long de notre liste.

III. Des programmes moins déterminés

`elem` est à l'intérieur de la boucle une variable tout ce qu'il y a de plus standard, on peut l'utiliser dans toutes nos opérations usuelles.

```
1 >>> for elem in numbers:
2 ...     result = 2 * elem + 1
3 ...     print(result)
4 ...
5 3
6 3
7 5
8 7
9 11
10 17
11 27
```

Comme toute variable, il nous est aussi possible de la redéfinir, mais attention : il ne s'agit que d'une variable assignée à un élément de la liste, elle n'est en aucune manière liée à la liste. Donc redéfinir la variable n'aura aucun effet sur la liste qui restera inchangée. Et la redéfinition n'est que temporaire, puisque la variable sera assignée à une nouvelle valeur de la liste à la prochaine itération.

```
1 >>> for elem in numbers:
2 ...     elem += 1
3 ...     print(elem)
4 ...
5 2
6 2
7 3
8 4
9 6
10 9
11 14
12 >>> numbers
13 [1, 1, 2, 3, 5, 8, 13]
```

III.5.2. Itération

III.5.2.1. Parcourir des listes

Avec ce bloc, il nous est ainsi possible d'itérer sur une liste et d'appliquer un même traitement à toutes les valeurs. Mais est-ce que c'est toujours ça que l'on veut ? Pas nécessairement, non, il peut être utile de différencier les cas.

Heureusement, nous avons pour cela les conditions avec lesquelles nous allons séparer nos cas.

Par exemple, on pourrait imaginer un mécanisme de recherche dans une liste. Le code parcourerait tous les éléments jusqu'à trouver celui ou ceux qui répondent à notre critère.

Disons par exemple que nous voulions trouver un nombre impair dans une liste de nombres. Dans le bloc de notre boucle, nous testerons si la valeur actuelle est impaire, et la conserverons

III. Des programmes moins déterminés

dans une variable si tel est le cas.

Ainsi, à la fin de la boucle, cette variable définie uniquement sous condition sera toujours assignée à notre valeur.

```
1 >>> numbers = [8, 2, 6, 3, 4, 0]
2 >>> for number in numbers:
3 ...     if number % 2 == 1:
4 ...         found = number
5 ...
6 >>> print('Trouvé : ', found)
7 Trouvé : 3
```

Vous pouvez changer l'ordre des éléments de la liste, le résultat est toujours le même.

On a par contre un petit soucis si notre liste contient plusieurs éléments impairs.

```
1 >>> numbers = [8, 2, 6, 3, 4, 0, 5]
2 >>> for number in numbers:
3 ...     if number % 2 == 1:
4 ...         found = number
5 ...
6 >>> print('Trouvé : ', found)
7 Trouvé : 5
```

Et oui, la condition est certes vraie pour 3 mais elle l'est aussi pour 5. Ainsi, on définit une première fois `found = 3` mais on l'écrase ensuite pour lui assigner 5, et on perd toute trace du 3.



Attention aussi, la variable `found` n'est ici définie que dans le cas où l'on rentre dans la condition.

Ainsi, si notre liste ne contient pas de nombre impair, la variable `found` ne sera pas définie. On pourrait résoudre ce problème en ajoutant `found = -1` avant notre boucle, donnant une valeur par défaut à la variable.

Il serait bien de pouvoir conserver toutes les valeurs qui correspondent à notre recherche. On n'aurait pas un type de donnée pour contenir un nombre indéterminé de valeurs ? La liste bien entendu !

```
1 >>> found = []
2 >>> for number in numbers:
3 ...     if number % 2 == 1:
4 ...         found.append(number)
5 ...
6 >>> found
7 [3, 5]
```

Passons maintenant à un exemple plus complexe et tentons d'identifier le plus grand nombre dans une liste. On va ainsi itérer sur les nombres, et s'il est le plus grand, on le conserve dans une variable.

III. Des programmes moins déterminés

Sur le principe c'est très bien, mais comment saura-t-on sur le moment qu'il est le plus grand de tous les nombres ? C'est difficile à déterminer, il nous faudrait à chaque fois reparcourir toute la liste pour voir si l'on trouve un autre nombre encore plus grand... ça fait beaucoup d'opérations.

Mais ce qu'on peut facilement déterminer, c'est s'il est le plus grand nombre jusqu'ici. En effet, on peut conserver dans une variable le plus grand nombre trouvé, et le mettre à jour chaque fois qu'on tombe sur un nombre qui lui est supérieur. À la fin du parcours, on est sûr que notre variable contient le plus grand nombre de la liste.

```
1 >>> numbers = [3, 2, 5, 8, 4, 7, 9, 1, 6]
2 >>> max_number = 0
3 >>> for number in numbers:
4 ...     if number > max_number:
5 ...         max_number = number
6 ...         print("Le plus grand nombre trouvé jusqu'ici est",
7 ...               max_number)
8 Le plus grand nombre trouvé jusqu'ici est 3
9 Le plus grand nombre trouvé jusqu'ici est 5
10 Le plus grand nombre trouvé jusqu'ici est 8
11 Le plus grand nombre trouvé jusqu'ici est 9
12 >>> max_number
13 9
```

On notera que cette opération existe déjà en Python et est réalisée par la fonction `max`.

```
1 >>> max(numbers)
2 9
```

III.5.2.2. Itérables

Les listes ne sont pas le seul type de données que l'on peut utiliser dans une boucle `for`, cela fonctionne aussi avec des chaînes de caractères par exemple, pour les parcourir caractère par caractère.

```
1 >>> for char in 'Hello':
2 ...     print(char)
3 ...
4 H
5 e
6 l
7 l
8 o
```

Plus généralement on parle d'**itérables** pour désigner les types que l'on peut parcourir avec un bloc `for`.

Un nouveau type de données va nous être bien utile ici, c'est le `range`. Un `range` représente un

III. Des programmes moins déterminés

intervalle entre deux nombres entiers, on peut le voir comme la liste des nombres entre ces deux bornes.

L'intervalle formé entre 1 et 10 se note par exemple `range(1, 10)`. Il faut savoir que c'est un intervalle fermé à gauche mais ouvert à droite, il contient ainsi 1 mais pas 10 (il s'arrête à 9).

```
1 >>> for n in range(1, 10):
2     ...     print(n)
3     ...
4 1
5 2
6 3
7 4
8 5
9 6
10 7
11 8
12 9
```

Avec ça, nous pouvons donc avoir une boucle itérant sur des nombres. Et encore une fois `n` est ici une variable redéfinie à chaque tour de boucle. On peut utiliser sa valeur dans nos calculs, comme ici pour la table de multiplication par 3.

```
1 >>> for n in range(1, 11):
2     ...     print('3 ×', n, '=', 3 * n)
3     ...
4 3 × 1 = 3
5 3 × 2 = 6
6 3 × 3 = 9
7 3 × 4 = 12
8 3 × 5 = 15
9 3 × 6 = 18
10 3 × 7 = 21
11 3 × 8 = 24
12 3 × 9 = 27
13 3 × 10 = 30
```

i

Il faut savoir qu'il est courant, dans des petites boucles (quelques lignes), d'utiliser un nom de variable court pour itérer sur nos valeurs. Ne soyez donc pas surpris de rencontrer des `i` (indice) ou `n` (*number*) utilisés à cet effet pour itérer sur des nombres, ou des `s` (*string*) pour itérer sur des chaînes de caractères.

Le premier argument donné à `range` est optionnel, et vaut 0 s'il est omis. Ainsi, `range(5)` est équivalent à `range(0, 5)`.

```
1 >>> for n in range(5):
2 ...     print(n)
3 ...
4 0
5 1
6 2
7 3
8 4
```

Et comme pour le *slicing*, les intervalles possèdent un pas optionnel, qui représente le nombre de valeurs à passer entre chaque élément. Un intervalle de 0 à 10 avec un pas de 2 représentera donc tous les nombres pairs de cet intervalle.

```
1 >>> for n in range(0, 10, 2):
2 ...     print(n)
3 ...
4 0
5 2
6 4
7 6
8 8
```

III.5.2.3. Construire une liste

À l'aide de `range` et de la méthode `append` des listes, on peut alors facilement construire une liste itérativement, en ajoutant un nouvel élément à chaque tour de boucle. Par exemple ici une liste de cinq zéros (équivalente à `[0] * 5`).

```
1 >>> zeros = []
2 >>> for _ in range(5):
3 ...     zeros.append(0)
4 ...
5 >>> zeros
6 [0, 0, 0, 0, 0]
```

i

Pour rappel, `_` est le nom de variable usuel pour une valeur que l'on n'utilise pas. On n'a en effet pas besoin ici de savoir quelle est la valeur de l'itération en cours, tout ce qui nous importe est de faire deux itérations.

Ou la liste des carrés des 10 premiers entiers naturels.

```
1 >>> squares = []
2 >>> for i in range(10):
```

III. Des programmes moins déterminés

```
3 ...     squares.append(i**2)
4 ...
5 >>> squares
6 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

C'est aussi la solution à notre problème de multiplication des listes multi-dimensionnelles, puisque nous avons maintenant un moyen d'instancier séparément chacune des sous-listes !

```
1 >>> grid = []
2 >>> for _ in range(2):
3 ...     grid.append([1, 2, 3])
4 ...
5 >>> grid
6 [[1, 2, 3], [1, 2, 3]]
7 >>> grid[0].append(4)
8 >>> grid
9 [[1, 2, 3, 4], [1, 2, 3]]
```

III.5.2.4. Boucles imbriquées

De la même manière que pour les conditions, les boucles peuvent être imbriquées les unes aux autres. Ce qui va permettre d'avoir un traitement répétitif dans une autre répétition.

Pour revenir à l'exemple des tables de multiplication, on peut ainsi représenter toutes les tables de 1 à 3.

```
1 >>> for a in range(1, 4):
2 ...     for b in range(1, 11):
3 ...         print(a, 'x', b, '=', a * b)
4 ...         print('---')
5 ...
6 1 x 1 = 1
7 1 x 2 = 2
8 1 x 3 = 3
9 1 x 4 = 4
10 1 x 5 = 5
11 1 x 6 = 6
12 1 x 7 = 7
13 1 x 8 = 8
14 1 x 9 = 9
15 1 x 10 = 10
16 ---
17 2 x 1 = 2
18 2 x 2 = 4
19 2 x 3 = 6
20 2 x 4 = 8
21 2 x 5 = 10
22 2 x 6 = 12
```

III. Des programmes moins déterminés

```
23 2 × 7 = 14
24 2 × 8 = 16
25 2 × 9 = 18
26 2 × 10 = 20
27 ---
28 3 × 1 = 3
29 3 × 2 = 6
30 3 × 3 = 9
31 3 × 4 = 12
32 3 × 5 = 15
33 3 × 6 = 18
34 3 × 7 = 21
35 3 × 8 = 24
36 3 × 9 = 27
37 3 × 10 = 30
38 ---
```

On remarque que le second `print` est en dehors de la deuxième boucle, il est ainsi exécuté à chaque itération de la première et permet de marquer une séparation entre chaque table.

Les boucles imbriquées nous permettent aussi de réaliser toutes sortes de combinaisons entre plusieurs ensembles de données.

```
1 >>> names = ['Jeanne', 'Paul', 'Max']
2 >>> fruits = ['pommes', 'poires', 'cerises', 'fraises']
3 >>>
4 >>> for name in names:
5 ...     for fruit in fruits:
6 ...         print(name, 'aime les', fruit)
7 ...
8 Jeanne aime les pommes
9 Jeanne aime les poires
10 Jeanne aime les cerises
11 Jeanne aime les fraises
12 Paul aime les pommes
13 Paul aime les poires
14 Paul aime les cerises
15 Paul aime les fraises
16 Max aime les pommes
17 Max aime les poires
18 Max aime les cerises
19 Max aime les fraises
```

Et bien sûr, on peut ajouter toutes sortes de conditions au sein de nos boucles.

```
1 >>> for name in names:
2 ...     for fruit in fruits:
3 ...         if name == 'Paul' and (fruit == 'pommes' or fruit ==
...         'cerises'):
```


III. Des programmes moins déterminés

```
4 ...         print(name, "n'aime pas les", fruit)
5 ...         else:
6 ...         print(name, "aime les", fruit)
7 ...
8 Jeanne aime les pommes
9 Jeanne aime les poires
10 Jeanne aime les cerises
11 Jeanne aime les fraises
12 Paul n'aime pas les pommes
13 Paul aime les poires
14 Paul n'aime pas les cerises
15 Paul aime les fraises
16 Max aime les pommes
17 Max aime les poires
18 Max aime les cerises
19 Max aime les fraises
```

Conclusion

Les boucles `for` et le mécanisme d'itération forment un pan très important du langage Python, c'est pourquoi nous aurons l'occasion de revenir dessus à de nombreuses reprises.

III.6. Boucler sur une condition (while)

Introduction

Les boucles `for` permettent de parcourir les éléments d'un itérable, mais un autre type de boucle est possible : les boucles testant une condition.

À la manière d'un `if` qui exécute un bloc si une condition est vraie, il s'agira ici d'exécuter un bloc tant que cette condition est vraie.

III.6.1. Boucler sur une condition

C'est ainsi qu'entre en scène la boucle `while` (qui signifie littéralement «tant que») et qui sert à boucler sur un prédicat. Un bloc `while` est alors assez similaire à un `if` : on a le mot-clé `while` suivi d'une expression conditionnelle et d'un `:`, puis les lignes indentées qui correspondent au contenu du bloc.

Ce contenu sera exécuté en boucle tant que le prédicat est vrai, celui-ci étant testé à nouveau avant chaque itération. Il convient donc dans le contenu du bloc de faire varier les valeurs utilisées par le prédicat, afin qu'il finisse par être faux et que l'on sorte de la boucle.

```
1 answer = 'o'
2
3 while answer == 'o':
4     print('Vous êtes dans la boucle')
5     answer = input('Souhaitez-vous rester dans la boucle (o/n) ? ')
6
7 print('Vous êtes sorti de la boucle')
```

```
1 Vous êtes dans la boucle
2 Souhaitez-vous rester dans la boucle (o/n) ? o
3 Vous êtes dans la boucle
4 Souhaitez-vous rester dans la boucle (o/n) ? o
5 Vous êtes dans la boucle
6 Souhaitez-vous rester dans la boucle (o/n) ? n
7 Vous êtes sorti de la boucle
```

À la première ligne de mon fichier, j'initialise une variable `answer` à `'o'`. Sans elle, la condition de mon `while` ne serait pas valide puisque `answer` n'aurait pas encore été définie (elle n'est définie ensuite que dans le bloc du `while`).

Je lui donne comme valeur `'o'` pour que la condition du `while` soit vraie et que l'on puisse entrer dans le bloc. Si j'initialise la variable à `'n'`, alors la condition sera fausse dès le premier test et le contenu du bloc jamais exécuté.

III. Des programmes moins déterminés

Pour chaque tour de boucle, l'expression conditionnelle est à nouveau évaluée. Si elle s'évalue à faux, la boucle s'arrête immédiatement. Ainsi dans mon exemple, à la 4ème itération de la boucle, `answer` vaut maintenant `'n'` (c'est la valeur qui lui a été donnée dans la 3ème itération). L'expression étant fausse, la boucle se termine et son contenu n'est pas exécuté pour la 4ème itération.

Mais l'expression conditionnelle n'est testée qu'au tout début de chaque itération, pas au milieu de celle-ci, ce qui fait que la boucle ne peut se terminer qu'à un moment bien précis. Regardons par exemple le code qui suit :

```
1 pv = 50
2
3 print('Pythachu a', pv, 'PV')
4
5 while pv > 0:
6     print('Pythachu perd 20 PV')
7     pv -= 20
8     print('Il lui reste maintenant', pv, 'PV')
9
10 print('Pythachu est KO, avec', pv, 'PV')
```

```
1 Pythachu a 50 PV
2 Pythachu perd 20 PV
3 Il lui reste maintenant 30 PV
4 Pythachu perd 20 PV
5 Il lui reste maintenant 10 PV
6 Pythachu perd 20 PV
7 Il lui reste maintenant -10 PV
8 Pythachu est KO, avec -10 PV
```

On constate bien qu'au cours de la 3ème itération, le nombre de PV devient inférieur à 0. Mais l'itération continue (on affiche le message), ce n'est qu'à l'itération suivante que l'expression est recalculée et que la boucle se termine.

On remarque aussi que la valeur `pv` existe toujours après la boucle, et possède la dernière valeur qui lui a été assignée.

III.6.2. Vers l'infini et au-delà

Notre boucle `while` s'arrête quand le prédicat devient faux. Mais que se passe-t-il alors si celui-ci est toujours vrai ?

Notre boucle se retrouve alors à tourner indéfiniment...

```
1 >>> while True:
2     ...     print("Vers l'infini et au-delà !")
3     ...
4 Vers l'infini et au-delà !
5 Vers l'infini et au-delà !
```

III. Des programmes moins déterminés

```
6 Vers l'infini et au-delà !
7 Vers l'infini et au-delà !
8 Vers l'infini et au-delà !
9 Vers l'infini et au-delà !
10 Vers l'infini et au-delà !
11 [...]
```

i

Quand votre programme rencontre une boucle infinie, utilisez la combinaison de touches **Ctrl**+**C** pour l'interrompre.

Bien sûr nous avons ici écrit volontairement une boucle infinie, mais celles-ci sont plus insidieuses et peuvent parfois se cacher là où on ne les attend pas. Prenons par exemple le programme suivant qui a pour but de calculer la factorielle¹ d'un nombre.

```
1 n = int(input('Entrez un nombre : '))
2
3 i = n
4 fact = 1
5 while i != 0:
6     fact *= i
7     i -= 1
8
9 print('La factorielle de', n, 'vaut', fact)
```

Listing 11 – factorielle.py

Ce code fonctionne très bien pour des entiers naturels :

```
1 % python factorielle.py
2 Entrez un nombre : 5
3 La factorielle de 5 vaut 120
4 % python factorielle.py
5 Entrez un nombre : 1
6 La factorielle de 1 vaut 1
```

Mais dans le cas où l'on entre un nombre négatif, le programme se met à boucler indéfiniment et l'on doit le couper avec un **Ctrl**+**C**.

```
1 % python factorielle.py
2 Entrez un nombre : -1
3 ^CTraceback (most recent call last):
4   File "factorielle.py", line 6, in <module>
5     fact *= i
6 KeyboardInterrupt
```

1. La factorielle est la fonction mathématique calculant le produit des nombres entiers de 1 à n. Ainsi la factorielle de 5 est $1 \times 2 \times 3 \times 4 \times 5$ soit 120.

III. Des programmes moins déterminés

En effet, pour un nombre négatif la condition `n != 0` sera toujours vraie puisque le nombre est décrémenté à chaque tour de boucle (il restera négatif et ne sera jamais nul). Dans l'idéal il faudrait donc traiter les nombres négatifs comme une erreur et afficher un avertissement dans ces cas-là pour prévenir toute boucle infinie.

Le soucis c'est qu'à l'exécution il n'est théoriquement pas possible de savoir si une boucle va s'arrêter ou non, c'est un problème indécidable ([problème de l'arrêt](#)) : dans le cas précédent on ne sait pas quelle valeur sera donnée à notre programme puisqu'elle dépend d'une saisie de l'utilisateur.

Ainsi, il faut être prudent et faire très attention aux conditions utilisées pour les `while` et bien s'assurer que celles-ci finissent toujours par devenir fausses.

Mais nous découvrirons par la suite qu'il y a des usages légitimes de boucles infinies, et des moyens de contrôler le déroulement de la boucle.

III.6.3. Boucle for ou boucle while ?

Avec nos deux types de boucles, on pourrait se demander quand utiliser l'une et quand utiliser l'autre.

Un bloc `while` correspond à l'action de répéter. On répète un même traitement et on fait varier les paramètres.

Il peut s'agir d'attendre une certaine entrée utilisateur ou d'affiner un calcul par exemple. Le but est alors que la boucle `while` serve à construire/calculer une valeur que l'on réutilisera par la suite.

On peut par exemple imaginer une boucle `while` pour calculer itérativement la racine carrée de 2 selon la [méthode de Héron](#), en s'arrêtant quand une certaine précision a été atteinte.

```
1 x = 1
2
3 while abs(2 - x**2) > 0.001:
4     x = (x + 2/x) / 2
5
6 print('La racine carrée de 2 vaut environ', x)
```

Pour tout le reste, il y a la boucle `for`.

Un bloc `for` correspond à l'action d'itérer, de parcourir des éléments.

Quand on souhaite exécuter une action pour chaque valeur d'une séquence identifiable (éléments d'une liste, caractères d'une chaîne, nombres d'un intervalle, etc.) c'est un `for` qui doit être utilisé, pour tout ce qui peut s'apparenter à de l'itération sur des valeurs.

Dans l'exemple des boucles infinies, nous n'aurions par exemple pas rencontré de problème en utilisant une boucle `for`. Ces dernières sont en effet plus facilement prédictibles si l'on sait que l'on va itérer sur un ensemble fini d'éléments (tel qu'un intervalle de nombres).

```
1 n = int(input('Entrez un nombre : '))
2
3 fact = 1
4 for i in range(2, n + 1):
5     fact *= i
```

III. Des programmes moins déterminés

```
6  
7 print('La factorielle de', n, 'vaut', fact)
```

III.7. Algorithmes

Introduction

Dans ce cours je ne veux pas seulement vous apprendre à écrire du code Python mais plus généralement à apprendre la programmation, et donc le raisonnement qui va avec.

Un programme informatique dans la vie courante va souvent consister à résoudre un problème particulier ou de multiples sous-problèmes : calculer un score, résoudre une équation, transformer une image, etc.

Pour résoudre ces problèmes on va bien sûr utiliser les outils proposés par le langage (les constructions telles que les conditions et les boucles, les types de données, etc.) mais cela ne fait pas tout, il faut leur donner un sens.

Ainsi on va devoir développer une suite logique d'opérations, comme une recette de cuisine, pour expliquer à l'ordinateur comment résoudre le problème, quel raisonnement appliquer.

Cette recette ou ce raisonnement, c'est ce que l'on appelle un algorithme.

Il y a des algorithmes génériques pour résoudre tous types de problèmes (nous allons en voir quelques uns ici), mais il est souvent nécessaire de les adapter, de les utiliser les uns avec les autres pour en former de nouveaux.

Pour continuer sur l'analogie de la recette de cuisine, il nous faut réaliser des choux d'une part et de la crème pâtissière de l'autre pour faire des choux à la crème, et spécialiser la crème selon le parfum que l'on veut lui donner.

Dans le cas d'un programme informatique il nous faut aussi réfléchir à ce que l'on a en entrée du programme (un nombre ? un fichier ? une adresse web ?) et ce que l'on souhaite en sortie (afficher un résultat ? créer un fichier ?).

III.7.1. Minimum d'une liste

On a déjà réalisé quelques algorithmes dans les derniers chapitres. Pour illustrer les boucles `for` je vous montrais par exemple comment identifier le maximum d'une liste. Identifier le minimum suit donc le même principe.

On avait imaginé deux solutions au problème. La première consisterait à itérer sur tous les éléments de la liste, et à vérifier pour chaque élément qu'il est le minimum en le comparant avec les autres éléments.

S'il existe un autre élément plus petit, alors l'élément courant ne peut pas être le minimum.

```
1 numbers = [3, 2, 5, 8, 4, 7, 9, 1, 6]
2 minimum = numbers[0]
3
4 for i in numbers[1:]:
5     is_minimum = True
6     for j in numbers:
```

III. Des programmes moins déterminés

```
7         if j < i:
8             is_minimum = False
9         if is_minimum:
10             minimum = i
11
12 print('Le minimum est', minimum)
```

Cet algorithme fonctionne mais est particulièrement inefficace : il nous faut re-parcourir la liste complète pour chaque élément. Cela revient à dire que pour une liste de N éléments on va devoir effectuer un total de N^2 comparaisons.

L'autre algorithme que nous avons implémenté était bien meilleur. On pouvait simplement ne comparer chaque élément qu'avec les éléments précédents.

Et comme le minimum des éléments précédents est déjà connu à chaque instant, on peut juste comparer chaque élément avec le minimum déjà connu.

```
1 numbers = [3, 2, 5, 8, 4, 7, 9, 1, 6]
2 minimum = numbers[0]
3
4 for i in numbers[1:]:
5     if i < minimum:
6         minimum = i
7
8 print('Le minimum est', minimum)
```

En plus d'être plus efficace (on ne réalise ici que N comparaisons pour une liste de taille N), le code est bien plus concis et lisible.

Même si la différence de performances ne saute pas aux yeux, parce que nos listes sont de tailles modestes, cela peut devenir problématique quand on commence à beaucoup l'utiliser ou l'appliquer à des données en plus grande quantité.

Cette notion de nombre d'opérations effectuées en fonction du nombre d'éléments en entrée est ce que l'on appelle la complexité algorithmique¹.

On dit de notre premier algorithme qu'il a une complexité quadratique (N^2 opérations pour N éléments), et du second qu'il a une complexité linéaire (N opérations).

i

Cet algorithme n'est présenté que comme un exercice, dans la vie de tous les jours il serait bien sûr inutile puisque Python possède déjà une fonction `min` qui fait le boulot.

III.7.2. Tri d'une liste

Maintenant que l'on sait identifier le plus petit élément d'une liste on peut passer à un problème un peu plus compliqué, celui du tri d'une liste. C'est un problème algorithmique très classique qui consiste à ranger dans l'ordre les éléments d'une liste.

Une approche simple consiste à trouver le minimum pour le retirer de la liste et l'ajouter à une nouvelle liste d'éléments triés. L'opération est alors recommencée jusqu'à épuiser la liste de

1. Pour en apprendre davantage sur la complexité algorithmique, je vous invite à consulter [ce tutoriel](#) .

III. Des programmes moins déterminés

départ.

C'est ce qu'on appelle le tri par sélection.

```
1 numbers = [3, 2, 5, 8, 4, 7, 9, 1, 6]
2 sorted_numbers = []
3
4 while numbers:
5     m = min(numbers)
6     sorted_numbers.append(m)
7     i = numbers.index(m)
8     del numbers[i]
9
10 print(sorted_numbers)
```

Cet algorithme n'est pas le plus efficace puisqu'il effectue environ N^2 opérations (le calcul du minimum comptant comme N comparaisons) pour une liste de N éléments. Il faut savoir que d'autres algorithmes plus performants existent tels que le tri rapide et le tri fusion, mais qui sont hors de portée pour le moment.

Python implémente lui-même son propre algorithme de tri inspiré des tris précédents, le *Tim sort* (du nom de Tim Peters, contributeur de Python). Ce tri est accessible par la fonction `sorted`.

```
1 >>> numbers = [3, 2, 5, 8, 4, 7, 9, 1, 6]
2 >>> sorted(numbers)
3 [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

i

Ainsi, avant de vous lancer tête baissée dans un algorithme, pensez à regarder si Python ne propose pas déjà une solution pour vous.

III.7.3. Un peu de dessin

Et si nous nous intéressions à des algorithmes un peu plus visuels ? Dans cette section je vous propose de dessiner diverses formes géométriques, à l'aide de caractères affichés dans le terminal.

III.7.3.1. Affichons un rectangle

Un premier exercice simple consiste à afficher un rectangle dans le terminal. À partir d'une largeur et d'une hauteur données (par exemple via des `input()`), on peut afficher les différents caractères pour dessiner les bords de notre rectangle.

On pourrait par exemple utiliser le caractère `+` pour les coins, `-` pour les lignes horizontales et `|` pour les lignes verticales.

Voici ce que donnerait un rectangle de dimensions 10×5 :

```

1  +-----+
2  |         |
3  |         |
4  |         |
5  +-----+

```

Listing 12 – Rectangle 10×5

i

Comme vous le voyez, les coins sont compris dans les dimensions du rectangle. Pensez alors aux cas «dégénérés» tels que des rectangles de dimensions 5×1 , 5×2 , 1×5 , 1×1 , 0×0 , etc.

Et voici une solution possible pour cet exercice :

👁 Contenu masqué n°4

III.7.3.2. Passons au triangle

Augmentons un peu la difficulté et cherchons maintenant à afficher un triangle. Il s'agit d'un triangle équilatéral défini par sa hauteur. Voici par exemple un triangle de hauteur 10 :

```

1  *
2      *  *
3      *  *
4      *  *
5      *  *
6      *  *
7      *  *
8      *  *
9      *  *
10 *****

```

Listing 13 – Triangle de hauteur 10

i

Chaque ligne du triangle fait ainsi toujours deux caractères de plus que la précédente : * donne suite à * * puis * * *, etc.

Et comme vous le voyez, les lignes devront être centrées pour avoir un joli résultat.

i

Pour connaître la marge à appliquer à gauche de chaque ligne, pensez à calculer en amont la taille de la dernière ligne.

Sans plus attendre, la solution :

👁 Contenu masqué n°5

III.7.3.3. Mon beau sapin, roi des forêts

Encore un cran de difficulté supplémentaire, mais le code du précédent triangle va nous être bien utile. On voudrait maintenant faire dessiner un sapin à notre programme.

Un sapin serait composé de triangle et trapèzes empilés, ainsi qu'un rectangle pour le tronc. Il serait là encore défini par une hauteur, désignant le nombre de sections du sapin ainsi que la taille de son tronc.

Vous trouverez ci-dessous des exemples de sapins de différentes tailles.

```

1 *
2   ***
3   *****
4   *********
5   |

```

Listing 14 – Sapin de taille 1

```

1 *
2   ***
3   *****
4   *********
5   *****
6   *********
7   *********
8   *****
9   *****
10  *****
11  *****
12  *****
13  *****
14  *****
15  *****
16  |||
17  |||
18  |||

```

Listing 15 – Sapin de taille 3

```

1 *
2   ***
3   *****
4   *****

```

```
5          *****
6         *******
7        *********
8       *********
9      *********
10     *********
11    *********
12   *********
13  *********
14  *********
15  *********
16  *********
17  *********
18  *********
19  *********
20  *********
21  *********
22  *********
23      |||||
24      |||||
25      |||||
26      |||||
```

Listing 16 – Sapin de taille 4

i

Triangles et trapèzes se dessinent à peu près de la même manière.

!

Attention aux tailles paires pour les dimensions du tronc.

Je ne vous propose pas de solution pour cet exercice, mais sachez que vous pouvez le retrouver sur la plateforme HackInScience : [Le sapin](#) .

i

[HackInScience](#) est un site d'exercices algorithmiques à réaliser avec Python où vous trouverez beaucoup d'exercices de ce genre. Vous pouvez y exécuter directement vos codes et valider vos solutions.

Un exercice validé permet d'accéder aux solutions partagées par les autres membres du site.

Conclusion

Voilà ce qui conclut ce chapitre, mais n'hésitez pas à vous reporter aux exercices ou à des plateformes comme [HackInScience](#) ou [exercism.io](#) pour continuer à vous entraîner et pratiquer l'algorithmique.

Contenu masqué

Contenu masqué n°4

```
1 width = int(input('Entrez la largeur : '))
2 height = int(input('Entrez la hauteur : '))
3
4 for y in range(height):
5     line = ''
6
7     if y == 0 or y == height - 1:
8         if width > 0:
9             line += '+'
10            line += '-' * (width - 2)
11            if width > 1:
12                line += '+'
13        else:
14            if width > 0:
15                line += '|'
16            line += ' ' * (width - 2)
17            if width > 1:
18                line += '|'
19
20    print(line)
```

[Retourner au texte.](#)

Contenu masqué n°5

```
1 height = int(input('Entrez la hauteur : '))
2
3 max_len = 2 * height - 1
4
5 width = 1
6
7 for y in range(height):
8     padding = (max_len - width) // 2
9     line = ' ' * padding
10
11    if width > 2 and y != height - 1:
12        line += '*' + ' ' * (width - 2) + '*'
13    else:
14        line += '*' * width
15
16    print(line)
17    width += 2
```

[Retourner au texte.](#)

III.8. TP : Ajoutons des boucles à notre jeu

Introduction

Les boucles vont nous permettre d'améliorer grandement notre jeu, puisque nous allons enfin pouvoir mettre en place des actions répétitives.

III.8.1. Itérer sur les attaques

Nous allons donc modifier le code de notre TP pour mettre les attaques disponibles sous la forme d'une liste. Ou même plutôt deux listes : une pour les noms d'attaques et une pour les dégâts associés.

```
1 attack_names = ['charge', 'tonnerre']
2 attack_damages = [20, 50]
```

Ainsi, il suffira d'itérer sur cette liste pour proposer chaque attaque au joueur, plutôt que de devoir les écrire manuellement une par une.

```
1 1. charge
2 2. tonnerre
```

En plus de ça, on pourra identifier l'attaque par son index (sa position dans la liste) et donc directement savoir quelle est l'attaque sélectionnée. En effet, si l'utilisateur entre 2, on peut savoir que ça correspond au nombre 2 (conversion en nombre avec `int`) soit à l'index 1 (puisqu'on commence toujours à l'index 0), et donc on peut exécuter l'attaque «tonnerre» sans avoir un bloc conditionnel par attaque.

```
1 >>> attack_idx = int(input('Quelle attaque ? ')) - 1
2 Quelle attaque ? 2
3 >>> attack_names[attack_idx]
4 'tonnerre'
```

On peut aussi ajouter une boucle `while` pour vérifier la validité de la saisie : si l'utilisateur entre un numéro d'attaque incorrect, il serait bon de lui demander à nouveau de choisir une attaque plutôt que de couper le programme.

On utilisera pour ça la méthode `isdigit` des chaînes de caractères qui renvoie un booléen indiquant si la chaîne représente un nombre ou non (ce qui permet d'effectuer la conversion sans erreur), on testera aussi si ce nombre est dans le bon intervalle.

III. Des programmes moins déterminés

```
1 >>> 'abc'.isdigit()
2 False
3 >>> '123'.isdigit()
4 True
5 >>> n = 2
6 >>> 1 <= n <= len(attack_names)
7 True
8 >>> n = 3
9 >>> 1 <= n <= len(attack_names)
10 False
```

L'utilisation d'expressions booléennes (à base d'opérations `not` et `or`) nous sera alors utile pour écrire la condition sous laquelle une saisie sera invalide.

Je vous laisse compléter cette partie avant de passer à la section suivante pour continuer à améliorer notre programme.

III.8.1.1. Solution

Voici la solution pour cette étape de l'exercice, le reste du code étant inchangé.

👁 Contenu masqué n°6

III.8.2. Un jeu au tour par tour

On va enfin vraiment avoir un jeu en tour par tour !

Grâce au `while`, il nous est en effet possible de répéter le processus de jeu jusqu'à ce que l'un des monstres soit KO. Une ébauche était présentée dans le chapitre sur les boucles `while` et il nous suffit de la compléter ici.

On souhaiterait donc que notre programme boucle tant que les deux monstres ont encore des points de vie, c'est-à-dire que leurs PV sont strictement supérieurs à zéro. À l'intérieur de la boucle, on saura donc que nos deux monstres sont encore en vie.

Un petit détail auquel il nous faudra cependant penser : le premier monstre attaque avant le second et lui retire des points de vie. Lors du tour du second monstre il est donc possible qu'il soit déjà KO, ce qui est censé l'empêcher d'attaquer. On ajoutera donc une condition pour éviter de rencontrer un bug avec ce cas.

En fin de jeu, on peut aussi retirer la condition de match nul puisque celle-ci ne pourra plus se produire : un monstre sera forcément KO avant l'autre.

On se permettra enfin d'ajouter un récapitulatif après chaque attaque pour nous rappeler où nous en sommes dans les points de vie.

III.8.2.1. Solution

Voici sans plus attendre la solution de notre jeu qui en est maintenant vraiment un. 🍊

👁 Contenu masqué n°7

i

Si vous faites bien attention, vous remarquerez que l'on peut arriver dans un cas où les PV d'un monstre sont négatifs.

Ce n'est pas très grave parce que cela ne change rien au déroulement du jeu, mais cela peut facilement se régler à l'aide d'une condition ou d'un appel à la fonction `max` : `pv2 = max(pv2 - damages, 0)`.

Contenu masqué

Contenu masqué n°6

```
1 ...
2
3 attack_names = ['charge', 'tonnerre']
4 attack_damages = [20, 50]
5
6 menu = 'quelle attaque voulez-vous utiliser ?'
7
8 # Joueur 1
9
10 print(name1, menu)
11 i = 1
12 for name in attack_names:
13     print(i, name.capitalize(), -attack_damages[i - 1], 'PV')
14     i += 1
15
16 att1 = input('> ')
17 while not att1.isdigit() or not 1 <= int(att1) <=
18     len(attack_names):
19     print('Attaque invalide, veuillez resaisir le numéro')
20     att1 = input('> ')
21
22 att1_idx = int(att1) - 1
23 damages = attack_damages[att1_idx]
24
25 pv2 -= damages
26 print(name1, 'attaque', name2, 'qui perd', damages, 'PV')
27
28 # Même chose pour le joueur 2
29 ...
```

[Retourner au texte.](#)

Contenu masqué n°7

```
1 name1 = input('Entrez le nom du 1er joueur : ').capitalize()
2
3 pv1_str = input('Et son nombre de PV : ')
4 while not pv1_str.isdigit():
5     print('Nombre de PV invalide (doit être un nombre positif)')
6     pv1_str = input('Entrez à nouveau : ')
7 pv1 = int(pv1_str)
8
9 name2 = input('Entrez le nom du 2ème joueur : ').capitalize()
10
11 pv2_str = input('Et son nombre de PV : ')
12 while not pv2_str.isdigit():
13     print('Nombre de PV invalide (doit être un nombre positif)')
14     pv2_str = input('Entrez à nouveau : ')
15 pv2 = int(pv2_str)
16
17 print()
18 print(name1, 'affronte', name2)
19 print()
20
21 attack_names = ['charge', 'tonnerre']
22 attack DAMAGES = [20, 50]
23
24 menu = 'quelle attaque voulez-vous utiliser ?'
25
26 while pv1 > 0 and pv2 > 0:
27     # Joueur 1
28
29     print(name1, menu)
30     i = 1
31     for name in attack_names:
32         print(i, name.capitalize(), -attack DAMAGES[i - 1], 'PV')
33         i += 1
34
35     att1 = input('> ')
36     while not att1.isdigit() or not 1 <= int(att1) <=
37         len(attack_names):
38         print('Attaque invalide, veuillez resaisir le numéro')
39         att1 = input('> ')
40
41     att1_idx = int(att1) - 1
42     damages = attack DAMAGES[att1_idx]
43
44     pv2 -= damages
45     print(name1, 'attaque', name2, 'qui perd', damages,
46         'PV, il lui en reste', pv2)
```

III. Des programmes moins déterminés

```
46     # Joueur 2
47     if pv2 > 0:
48         print(name2, menu)
49         i = 1
50         for name in attack_names:
51             print(i, name.capitalize(), -attack_damages[i - 1],
52                   'PV')
53             i += 1
54
55         att2 = input('> ')
56         while not att2.isdigit() or not 1 <= int(att2) <=
57             len(attack_names):
58             print('Attaque invalide, veuillez resaisir le numéro')
59             att2 = input('> ')
60
61         att2_idx = int(att2) - 1
62         damages = attack_damages[att2_idx]
63
64         pv1 -= damages
65         print(name2, 'attaque', name1, 'qui perd', damages,
66               'PV, il lui en reste', pv1)
67
68     if pv1 > pv2:
69         print(name1, 'remporte le combat')
70     else:
71         print(name2, 'remporte le combat')
```

[Retourner au texte.](#)

Quatrième partie

Types de données

Introduction

Python est un langage disposant de nombreux types de données par défaut, répondant à différents besoins. Ce chapitre a pour but d'en présenter les principaux pour découvrir à quoi ils servent et comment les utiliser.

IV.1. Retour sur les types précédents

Introduction

Mais avant d'aller plus loin, il serait bon de refaire un tour d'horizon des types vus précédemment, de revoir leurs opérations et méthodes.

IV.1.1. Généralités

Une valeur en Python peut-être le résultat de n'importe quelle expression (opération, appel de fonction, accès à une variable, etc.). Toute valeur possède un type qui définit les opérations qui lui sont applicables.

IV.1.1.1. Conversions

Certaines valeurs peuvent être converties d'un type vers un autre. Pour cela, les types s'utilisent comme des fonctions, où la valeur à convertir est donnée en argument.

```
1 >>> int('123')
2 123
3 >>> float(42)
4 42.0
5 >>> list('abc')
6 ['a', 'b', 'c']
7 >>> str(1.5)
8 '1.5'
```

IV.1.1.2. Comparaisons

Les opérateurs d'égalité (==) et de différence (!=) sont applicables à tous les types.

```
1 >>> 2 == 2
2 True
3 >>> 1.5 == 3.7
4 False
5 >>> 'abc' != 'def'
6 True
7 >>> [1, 2, 3] == [1, 2, 3]
8 True
```

Ces opérations sont de plus compatibles entre valeurs de types différents.

```
1 >>> 2 == 2.0
2 True
3 >>> 2 == 2.5
4 False
5 >>> 2 != '2'
6 True
7 >>> 2 == [2]
8 False
```

IV.1.2. Booléens

Le type booléen (`bool`) permet de représenter les valeurs `True` (vrai) et `False` (faux). Ces valeurs peuvent être le résultat d'une opération booléenne comme les opérateurs de comparaison (`==`, `!=`, `<`, etc.).

IV.1.2.1. Conversions

Toute valeur Python peut être interprétée comme un booléen (par conversion implicite) :

- Le nombre zéro (`0`, `0.0`) vaut `False`.
- Les conteneurs vides (chaîne vide, liste vide) valent `False`.
- Toute valeur qui ne vaut pas explicitement `False` vaut `True`.

Il est aussi possible de convertir explicitement une valeur en booléen en faisant appel au type booléen. Les règles de conversion sont les mêmes que celles énoncées ci-dessus.

```
1 >>> bool(5)
2 True
3 >>> bool(0.0)
4 False
5 >>> bool('abc')
6 True
7 >>> bool([])
8 False
```

IV.1.2.2. Opérations

Trois opérateurs existent sur les booléens : `not` (unaire), `and` et `or` (binaires), répondant chacun à une table de vérité.

a	not a
True	False
False	True

IV. Types de données

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

Ces opérateurs sont présentés comme renvoyant toujours `True` ou `False`, mais ce n'est en vérité pas toujours le cas. Étant donné que toute valeur peut être vue comme un booléen, il suffit de renvoyer une valeur qui sera interprétée comme `True` ou `False`.

Prenons le cas de `and` avec une opération `a and b` :

- Si `a` vaut `False`, le résultat sera forcément `False`. Donc `and` fait un raccourci et ne regarde même pas `b`. Dans ce cas la valeur renvoyée est `a` (qui peut être interprétée comme `False`).
- Si `a` vaut `True`, alors `and` renverra simplement `b`, puisque la conversion de `b` en booléen sera le résultat de l'opération.

```
1 >>> [] and 5
2 []
3 >>> ['foo'] and 5
4 5
5 >>> ['foo'] and 0
6 0
7 >>> 5 and True
8 True
```

Le même genre de raccourci existe pour `or`, qui renvoie `a` si `a` vaut `True` et `b` sinon.

```
1 >>> ['foo'] or 5
2 ['foo']
3 >>> [] or 5
4 5
5 >>> [] or 0
6 0
7 >>> 0 or True
```



```
8 True
```

On notera enfin en termes de conversions que les booléens eux-mêmes sont aussi implicitement convertis en nombres lorsqu'utilisés comme tels. On aura ainsi `True` converti en `1` et `False` en `0`.

```
1 >>> False + 2 * True
2 2
```

IV.1.3. Nombres

Nous avons rencontré deux types pour représenter les nombres : les entiers et les flottants (nombres à virgule). Les premiers ne représentent que des nombres entiers, avec une précision infinie (il n'y a pas de limite), les seconds représentent des nombres réels mais avec une précision limitée.

IV.1.3.1. Conversions

On peut facilement convertir des valeurs en `int` ou `float` en faisant appel à ces types comme à des fonctions.

```
1 >>> int('42')
2 42
3 >>> float('4.5')
4 4.5
5 >>> int(4.5)
6 4
7 >>> int(float('4.5'))
8 4
```

Toute valeur n'est pas convertible en nombre, des erreurs peuvent donc survenir si l'argument est invalide.

```
1 >>> int('4.5') # 4.5 n'est pas un nombre entier valide
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ValueError: invalid literal for int() with base 10: '4.5'
```

IV.1.3.2. Opérations

Beaucoup d'opérations sont applicables sur les nombres, et les deux types sont compatibles entre-eux. On retrouve d'abord les opérations arithmétiques: addition (+), soustraction (-), multiplication (*), division (/), division euclidienne (//), modulo (%) et puissance (**).

IV. Types de données

```
1 >>> 1 + 5
2 6
3 >>> 4.3 - 2
4 2.3
5 >>> 1.5 * 4
6 6.0
7 >>> 7 / 5
8 1.4
9 >>> 7 // 5
10 1
11 >>> 7 % 5
12 2
13 >>> 2 ** 3
14 8
```

On a aussi tous les opérateurs de comparaison, qui renvoient des valeurs booléennes : l'égalité (`==`), la différence (`!=`), l'infériorité (`<`, `<=`) et la supériorité (`>`, `>=`).

```
1 >>> 3 == 3.0
2 True
3 >>> 1 != 2
4 True
5 >>> 3 < 4.0
6 True
7 >>> 4 > 4.0
8 False
9 >>> 4 >= 4.0
10 True
```

Et d'autres opérateurs ne sont accessibles que via des fonctions : la valeur absolue (`abs`), l'arrondi (`round`), le minimum (`min`), le maximum (`max`) et la division euclidienne avec reste (`divmod`).

```
1 >>> abs(-1.2)
2 1.2
3 >>> round(1/3, 4)
4 0.3333
5 >>> min(1.5, -6, 3.7)
6 -6
7 >>> max(1.5, -6, 3.7)
8 3.7
9 >>> divmod(7, 5)
10 (1, 2)
```

IV.1.3.3. Représentations des entiers

On sait que l'on peut convertir un nombre en chaîne de caractères en appelant `str`, qui utilise la représentation décimale (en base 10), mais d'autres représentations sont possibles pour les entiers. `bin` permet d'avoir la représentation binaire (base 2), `oct` pour l'octale (base 8) et `hex` pour l'hexadécimale (base 16).

```
1 >>> bin(42)
2 '0b101010'
3 >>> oct(42)
4 '0o52'
5 >>> hex(42)
6 '0x2a'
```

Ces représentations sont d'ailleurs parfaitement valides pour être entrées en tant que nombre dans l'interpréteur, qui les analysera donc selon leur préfixe (`0b`, `0o` ou `0x`).

```
1 >>> 0b101010
2 42
3 >>> 0o52
4 42
5 >>> 0x2a
6 42
```

Puis qu'on en est à parler des bases, tout nombre peut ainsi être considéré comme une succession de bits (tel que la représentation renvoyée par `bin`). Un bit étant soit 0 soit 1, on peut même parler de tableau de booléens.

Différents opérateurs—à rapprocher des opérateurs booléens—tirent parti de cette particularité pour offrir des opérations bit-à-bit sur les nombres.

Ainsi, l'opérateur ET (&) calcule le nombre résultat de l'application d'un ET binaire (`and`) entre chaque bit de deux nombres.

```
1 >>> bin(0b101010 & 0b111000)
2 '0b101000'
```

beginarrayrcccccccolorblue1colorblue0colorblue1010&111000

\models 1 0 1 0 0 0

J'utilise ici des représentations binaires pour que le calcul soit plus lisible, mais l'opérateur s'applique simplement sur des entiers et renvoie un entier.

```
1 >>> 42 & 56
2 40
```

De la même manière, on a les opérateurs OU-inclusif (`|`) et OU-exclusif/XOR (`^`).

IV. Types de données

```
1 >>> bin(0b101010 | 0b111000)
2 '0b111010'
```

beginarrayrccccccolorblue1colorblue0colorblue1010|111000

\models 1 1 1 0 1 0

```
1 >>> bin(0b101010 ^ 0b111000)
2 '0b10010'
```

beginarrayrccccccolorblue1colorblue0colorblue1010111000

\models 0 1 0 0 1 0

i

Notez que le premier zéro n'apparaît pas dans le résultat renvoyé par Python pour le XOR, mais `0b10010` et `0b010010` sont bien deux représentations du même nombre (18).

D'autres opérations bit-à-bit sont encore possibles (`~`, `<`, `>`), vous pourrez en apprendre plus sur [cette page dédiée aux opérateurs](#) ↗.

IV.1.3.4. Précision des flottants

Les nombres flottants en Python ont une précision limitée, c'est-à-dire qu'ils auront du mal à représenter des nombres trop grands ou avec trop de chiffres après la virgule.

```
1 >>> 0.100000000000000001
2 0.1
```

On voit ici que le dernier 1 s'est perdu. C'est dû au fait que ces nombres sont stockés sur une zone mémoire de taille fixe, et que des arrondis sont nécessaires dans certains cas.

On peut le voir aussi sur d'autres opérations qui produisent normalement des nombres infinis.

```
1 >>> 1/3
2 0.3333333333333333
3 >>> 7/6
4 1.1666666666666667
```

Par ailleurs, les nombres y sont stockés en base 2, et certains nombres qui nous paraissent finis (`0.1`) ne le sont pas en binaire (il faut une infinité de chiffre derrière la virgule pour représenter `0.1` en base 2). C'est pourquoi des arrondis sont effectués sur ces nombres. Ils ne sont pas toujours visibles, mais ils peuvent apparaître à certains moments et être source de bugs.

IV. Types de données

```
1 >>> 0.1 + 0.1 + 0.1
2 0.30000000000000004
```

En raison de ces arrondis il est plutôt déconseillé de comparer deux flottants avec `==`, puisque cela pourrait amener à un résultat incohérent. Nous verrons dans la suite du cours comment résoudre ce problème.

```
1 >>> 0.1 + 0.1 + 0.1 == 0.3
2 False
```

IV.1.3.5. Notation des flottants

123.456 est la notation habituelle des nombres flottants, mais une autre est possible : la notation scientifique. Il s'agit de représenter un nombre avec un exposant d'une puissance de 10. Cela aide à écrire les nombres très grands ou très petits.

Par exemple, `3.2e5` est égal à `3.5 * 10**5` soit `320000.0`, et `4e-3` à `4.0 * 10**-3` donc `0.004`

```
1 >>> 3.2e5
2 320000.0
3 >>> 4e-3
4 0.004
```

Pour certains nombres, trop grands/petits pour être représentés correctement avec la notation habituelle, Python basculera automatiquement en notation scientifique.

```
1 >>> 9.6 ** 100
2 1.6870319358849588e+98
3 >>> 2 / 10000000000
4 2e-10
```

Enfin, il est aussi possible avec les flottants de représenter les infinis (positif et négatif), mais ils ne sont pas directement accessibles. On peut accéder à l'infini positif à l'aide de l'expression `float('inf')`.

```
1 >>> inf = float('inf')
2 >>> inf
3 inf
4 >>> inf + 2
5 inf
6 >>> inf * inf
7 inf
8 >>> 1 / inf
```

IV. Types de données

```
9 0.0
```

L'infini sera toujours supérieur à n'importe quel autre nombre.

```
1 >>> inf > 10**100
2 True
```

De façon similaire, on retrouve l'infini négatif avec `float('-inf')`.

IV.1.3.6. Nombres complexes

i

Vous pouvez passer cette section si vous n'êtes pas familiers des nombres complexes, ce n'est pas important pour la suite.

Python embarque aussi nativement les nombres complexes qui sont accessibles via le suffixe `j` pour représenter la partie imaginaire. Les complexes sont un sur-ensemble des flottants, et les mêmes opérations sont donc applicables sur eux.

```
1 >>> 1+2j + 4+5j
2 (5+7j)
3 >>> 0.5j + 3.2+9.3j
4 (3.2+9.8j)
5 >>> (1+2j) * (4+5j)
6 (-6+13j)
7 >>> 1j*1j
8 (-1+0j)
9 >>> (1+2j) ** 2
10 (-3+4j)
```

Par ailleurs, on trouve sur ces nombres des attributs `real` et `imag` pour accéder aux parties réelle et imaginaire, et une méthode `conjugate` pour calculer le conjugué.

```
1 >>> c = 1+2j
2 >>> c.real
3 1.0
4 >>> c.imag
5 2.0
6 >>> c.conjugate()
7 (1-2j)
```

Bien sûr, les nombres complexes ne sont pas ordonnables entre-eux.

```
1 >>> 1+2j < 2+1j
2 Traceback (most recent call last):
```

```
3 File "<stdin>", line 1, in <module>
4 TypeError: '<' not supported between instances of 'complex' and
  'complex'
```

Enfin, la fonction `abs` (valeur absolue) permet aussi de calculer le module d'un nombre complexe.

```
1 >>> abs(3+4j)
2 5.0
```

IV.1.4. Chaînes de caractères

La chaîne de caractère est le type utilisé pour représenter du texte, on peut la voir comme une séquence (ou un tableau) de caractères.

IV.1.4.1. Conversions

Toute valeur Python est convertible en chaîne de caractères, en faisant appel à `str`.

```
1 >>> str(True)
2 'True'
3 >>> str(4)
4 '4'
5 >>> str(1.5)
6 '1.5'
7 >>> str('foo')
8 'foo'
9 >>> str([1, 2, 3])
10 '[1, 2, 3]'
```

C'est ainsi que `print` procède d'ailleurs pour afficher n'importe quelle valeur.

IV.1.4.2. Opérations

La longueur d'une chaîne peut être obtenue par un appel à la fonction `len`.

```
1 >>> len('foo')
2 3
3 >>> len('hello world')
4 11
5 >>> len('')
6 0
```

IV. Types de données

IV.1.4.2.1. Indexation

On peut accéder aux différents caractères de la chaîne à l'aide de l'opérateur d'indexation `[]` accompagné d'un index (une position dans la chaîne, à partir de 0). Cet index peut être négatif pour parcourir la chaîne depuis la fin.

```
1 >>> 'hello world'[1]
2 'e'
3 >>> 'hello world'[-3]
4 'r'
```

On peut préciser un intervalle d'index grâce au *slicing* avec la syntaxe `debut:fin:pas` où chaque élément est optionnel.

```
1 >>> 'hello world'[3:]
2 'lo world'
3 >>> 'hello world'[:-4]
4 'hello w'
5 >>> 'hello world'[1:8:2]
6 'el o'
```

IV.1.4.2.2. Concaténation

Il est possible de concaténer (mettre à la suite) plusieurs chaînes de caractères avec l'opérateur `+`.

```
1 >>> 'hello' + ' ' + 'world' + '!'
2 'hello world!'
```

On peut aussi «multiplier» une chaîne par un nombre entier n pour obtenir n concaténations de cette même chaîne.

```
1 >>> 'hello ' * 3
2 'hello hello hello '
```

IV.1.4.2.3. Relations d'ordre

Les chaînes de caractères sont ordonnées les unes par rapport aux autres, il est donc possible d'utiliser les opérateurs `<`, `>`, `<=` et `>=` entre deux chaînes.

La comparaison est faite en fonction de l'ordre lexicographique, une extension de l'ordre alphabétique.

```
1 >>> 'abc' < 'def'
2 True
```


IV. Types de données

```
3 >>> 'abc' > 'def'
4 False
```

IV.1.4.2.4. Appartenance

L'opérateur `in` permet de tester si une chaîne contient un caractère ou une sous-chaîne (ou vu autrement, si cette sous-chaîne appartient à la chaîne). L'opération renvoie un booléen.

```
1 >>> 'h' in 'hello'
2 True
3 >>> 'lo' in 'hello'
4 True
5 >>> 'la' in 'hello'
6 False
```

IV.1.4.3. Principales méthodes

Les méthodes `lstrip`, `rstrip` et `strip` permettent respectivement de renvoyer une nouvelle chaîne en supprimant les espaces au début, à la fin ou des deux côtés.

```
1 >>> ' foo bar '.lstrip()
2 'foo bar '
3 >>> ' foo bar '.rstrip()
4 ' foo bar'
5 >>> ' foo bar '.strip()
6 'foo bar'
```

Elles acceptent un argument optionnel pour supprimer des caractères en particulier plutôt que des espaces.

```
1 >>> '...hello...'.strip('.')
2 'hello'
3 >>> '.-hello-..'.strip('.-')
4 'hello'
```



Attention, l'argument donné à `strip` spécifie un ensemble de caractères à supprimer et nom une chaîne précise. `s.strip('.-')` et `s.strip('-.')` sont équivalents.

Les méthodes `upper`, `lower`, `capitalize` et `title` permettent d'obtenir une nouvelle chaîne en changeant la casse des caractères.

IV. Types de données

```
1 >>> 'HeLlO wOrLd!'.upper()
2 'HELLO WORLD!'
3 >>> 'HeLlO wOrLd!'.lower()
4 'hello world!'
5 >>> 'HeLlO wOrLd!'.capitalize()
6 'Hello world!'
7 >>> 'HeLlO wOrLd!'.title()
8 'Hello World!'
```

`index` et `find` servent à trouver la première position d'un caractère (ou d'une sous-chaîne) dans une chaîne.

`index` produit une erreur si le caractère n'est pas trouvé, `find` renvoie `-1`.

```
1 >>> 'hello world'.index('o')
2 4
3 >>> 'hello world'.find('h')
4 0
5 >>> 'hello world'.index('world')
6 6
7 >>> 'hello world'.find('world')
8 6
9 >>> 'hello'.index('w')
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12   ValueError: substring not found
13 >>> 'hello'.find('w')
14 -1
```

Il est possible de compter le nombre d'occurrences d'un caractère (ou d'une sous-chaîne) avec la méthode `count`.

```
1 >>> 'hello world'.count('o')
2 2
3 >>> 'toto'.count('to')
4 2
```

On peut tester spécifiquement si une chaîne commence ou termine par une autre avec les méthodes `startswith` et `endswith`. Ces méthodes renvoient un booléen.

```
1 >>> 'hello world'.startswith('hello')
2 True
3 >>> 'hello world'.endswith('hello')
4 False
5 >>> 'hello world'.startswith('world')
6 False
7 >>> 'hello world'.endswith('world')
```

8	True
---	------

i

Depuis Python 3.9, les chaînes de caractères possèdent aussi des méthodes `removeprefix` et `removesuffix` qui permettent de retirer une sous-chaîne au début ou à la fin de notre chaîne.

Ces méthodes ne produisent pas d'erreur si la sous-chaîne n'est pas trouvée et renvoient juste la chaîne telle quelle.

```
1 >>> 'helloworld'.removeprefix('hello')
2 'world'
3 >>> 'helloworld'.removesuffix('world')
4 'hello'
5 >>> 'helloworld'.removeprefix('world')
6 'helloworld'
```

Différents tests sont possibles sur les chaînes de caractères pour savoir si elles sont composées de caractères alphanumériques (`isalnum`), alphabétiques (`isalpha`), numériques (`isdigit`) et d'autres encore.

```
1 >>> 'salut123'.isalnum()
2 True
3 >>> 'salut 123'.isalnum()
4 False
5 >>> 'salut'.isalpha()
6 True
7 >>> 'salut123'.isalpha()
8 False
9 >>> '123'.isdigit()
10 True
```

IV.1.4.4. Méthodes avancées

La méthode `replace` permet de renvoyer une copie de la chaîne en remplaçant un caractère (ou une sous-chaîne) par un autre.

```
1 >>> 'hello world'.replace('o', 'a')
2 'hella world'
3 >>> 'hello world'.replace('ll', 'xx')
4 'hexxo world'
5 >>> 'hello world'.replace('ll', '')
6 'heo world'
```

On peut découper une chaîne de caractères vers une liste de chaînes à partir d'un séparateur (caractère ou sous-chaîne) avec la méthode `split`. Par défaut, le séparateur est l'espace.

IV. Types de données

```
1 >>> 'hello world'.split()
2 ['hello', 'world']
3 >>> 'abc:def:ghi'.split(':')
4 ['abc', 'def', 'ghi']
5 >>> 'abc : def : ghi'.split(' : ')
6 ['abc', 'def', 'ghi']
```

Ce séparateur ne peut pas être une chaîne vide.

```
1 >>> 'hello'.split('')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 ValueError: empty separator
```

Enfin, il est possible d'unir les chaînes de caractère d'une liste autour d'un séparateur en utilisant la méthode `join` sur ce séparateur.

```
1 >>> ''.join(['hello', 'world'])
2 'hello world'
3 >>> ':'.join(['abc', 'def', 'ghi'])
4 'abc:def:ghi'
```

La chaîne vide est ici acceptée pour concaténer directement les chaînes.

```
1 >>> ''.join(['h', 'e', 'l', 'l', 'o'])
2 'hello'
```

IV.1.5. Listes

Les listes représentent des tableaux de valeurs de tous types. Contrairement aux types précédents, les listes sont des objets modifiables (leur valeur peut varier avec le temps).

IV.1.5.1. Conversions

Une chaîne de caractères étant une séquence, elle peut être convertie en liste de caractères en faisant appel à `list`.

```
1 >>> list('hello')
2 ['h', 'e', 'l', 'l', 'o']
```

Cela peut justement permettre de récupérer l'équivalent modifiable d'une chaîne de caractères.

```
1 >>> txt = list('hello')
2 >>> txt[1] = 'a'
3 >>> ''.join(txt)
4 'hallo'
```

IV.1.5.2. Opérations

On retrouve pour les listes les opérations d'indexation (`[]`), de concaténation (`+`, `*`) et d'appartenance (`in`).

L'indexation permet cependant de modifier une liste en assignant une valeur à une position et d'en supprimer avec `del`.

```
1 >>> values = [3, 4, 5]
2 >>> values[0]
3 3
4 >>> values[1:]
5 [4, 5]
6 >>> values[-1] = 6
7 >>> del values[0]
8 >>> values
9 [4, 6]
10 >>> [1, 2] + values
11 [1, 2, 4, 6]
12 >>> values * 2
13 [4, 6, 4, 6]
14 >>> 4 in [1, 2, 3]
15 False
16 >>> 4 in [1, 2, 4]
17 True
```

Contrairement aux chaînes de caractères, l'opérateur `in` n'ira pas chercher de sous-liste dans une liste.

```
1 >>> [1, 2] in [1, 2, 3]
2 False
3 >>> [1, 2] in [[1, 2], [3, 4]]
4 True
```

Au niveau de la multiplication d'une liste par un nombre, il faut bien faire attention aux cas de références multiples. Quand on multiplie ainsi une liste, on ne copie pas les éléments qu'elle contient, mais on ne fait que les dupliquer. On a donc plusieurs fois un même objet dans la liste.

Ce n'est pas gênant pour des valeurs non modifiables (nombres, chaînes), mais si une liste contient d'autres listes cela peut vite devenir problématique.

IV. Types de données

```
1 >>> table = [[0, 0, 0]] * 2
2 >>> table
3 [[0, 0, 0], [0, 0, 0]]
4 >>> table[0][1] = 5
5 >>> table
6 [[0, 5, 0], [0, 5, 0]]
```

Les opérateurs d'ordre (<, >) sont aussi utilisables entre deux listes, leur résultat dépend de la comparaison entre les éléments des listes, par ordre lexicographique.

C'est-à-dire qu'on commence par comparer les premiers éléments des deux listes : s'ils sont différents, alors la liste dont l'élément est le plus grand est considérée comme supérieure.

```
1 >>> [3, 0, 0] > [1, 9, 9]
2 True
3 >>> [3, 0, 0] < [1, 9, 9]
4 False
5 >>> [3, 0, 0] < [4, 9]
6 True
7 >>> [1, 2, 3] < [2]
8 True
9 >>> ['abc', 'def'] < ['ghi']
10 True
```

Mais s'ils sont égaux, l'opération continue en passant aux éléments suivants, et ainsi de suite jusqu'à l'épuisement de l'une des listes. Une liste qui est épuisée avant l'autre est considérée comme inférieure. Ainsi [1, 2, 3] est inférieure à [1, 2, 3, 4].

```
1 >>> [1, 2, 3] < [1, 2, 4]
2 True
3 >>> [1, 2, 3] < [1, 2, 2]
4 False
5 >>> [1, 2, 3] < [1, 2, 3, 4]
6 True
7 >>> [1, 2, 3, 9] < [1, 2, 4]
8 True
9 >>> [1, 2, 3] < [1]
10 False
11 >>> ['abc', 'def'] > ['abc']
12 True
```

Dans le cas où les éléments des deux listes ne sont pas ordonnables, on obtient une erreur de type signifiant que la comparaison est impossible.

```
1 >>> [1, 2] < [1, 'a']
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
```

IV. Types de données

```
4 TypeError: '<' not supported between instances of 'int' and 'str'
```

Et on retrouve bien sûr les opérateurs d'inégalités `<=` et `>=`.

```
1 >>> [3, 2, 1] > [3, 2, 1]
2 False
3 >>> [3, 2, 1] >= [3, 2, 1]
4 True
5 >>> [3, 2, 1] <= [3, 2, 1]
6 True
```

D'autres opérateurs prennent la forme de fonctions. C'est le cas de `len` pour récupérer la taille d'une liste.

```
1 >>> len(['a', 'b', 'c'])
2 3
```

On a aussi les fonctions `min` et `max` pour récupérer le plus petit ou le plus grand élément d'une liste.

```
1 >>> min([3, 1, 2])
2 1
3 >>> max(['z', 'c', 'a', 'y'])
4 'z'
```

`sum` est une fonction qui opère sur une liste de nombres et en calcule la somme.

```
1 >>> sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
2 55
```

Enfin je voulais aussi vous présenter les fonctions `all` et `any`, qui agissent comme des `and/or` sur l'ensemble des éléments d'une liste, mais renvoient un booléen dans tous les cas. `all` vérifie que tous les éléments sont vrais, et `any` qu'au moins un élément est vrai.

Ainsi, `all([a, b, c, d])` est équivalent à `a and b and c and d` et `any([a, b, c, d])` à `a or b or c or d`.

```
1 >>> all([1, 2, 3, 4])
2 True
3 >>> all([1, 2, 3, 4])
4 True
5 >>> all([0, 1, 2, 3, 4])
6 False
7 >>> any([0, 1, 2, 3, 4])
8 True
```

IV. Types de données

```
9 >>> any([0])
10 False
```

Attention cependant au comportement sur les listes vides : `all` s'attend à ce que tous les éléments soient vrais ; mais si la liste ne contient aucun élément, alors techniquement ils sont bien tous vrais. De même pour `any` qui veut au moins un élément vrai, ce qui ne peut pas être le cas s'il n'y a aucun élément.

```
1 >>> all([])
2 True
3 >>> any([])
4 False
```

IV.1.5.3. Principales méthodes

Venons-en maintenant à quelques méthodes sur les listes.

Comme sur les chaînes, on a une méthode `index` pour rechercher le premier index d'un élément.

```
1 >>> values = ['a', 'b', 'c', 'd']
2 >>> values.index('c')
3 2
```

Les méthodes `append`, `insert`, `pop` et `clear` permettent de modifier la liste en ajoutant / insérant / supprimant un élément, ou en la vidant.

```
1 >>> values.append('e')
2 >>> values.insert(3, 'ç')
3 >>> values.pop(1)
4 'b'
5 >>> values
6 ['a', 'c', 'ç', 'd', 'e']
7 >>> values.clear()
8 >>> values
9 []
```

Les listes ont aussi une méthode `remove` pour supprimer un élément en fonction de sa valeur plutôt que son index.

```
1 >>> values = ['a', 'b', 'c', 'd']
2 >>> values.remove('c')
3 >>> values
4 ['a', 'b', 'd']
```

La méthode `extend` permet d'ajouter une liste d'éléments à la fin, ce qui revient à concaténer la liste donnée en argument dans la liste actuelle.

IV. Types de données

```
1 >>> values.extend(['c', 'e', 'f'])
2 >>> values
3 ['a', 'b', 'd', 'c', 'e', 'f']
```

Quelques méthodes permettent de faire varier l'ordre des éléments dans la liste. C'est le cas de `reverse` qui inverse l'ordre des éléments.

```
1 >>> values.reverse()
2 >>> values
3 ['f', 'e', 'c', 'd', 'b', 'a']
```

`sort` permet quant à elle de trier les éléments du plus petit au plus grand.

```
1 >>> values.sort()
2 >>> values
3 ['a', 'b', 'c', 'd', 'e', 'f']
```

Il est possible de passer un booléen comme argument nommé `reverse` pour trier les éléments dans l'autre sens.

```
1 >>> values.sort(reverse=True)
2 >>> values
3 ['f', 'e', 'd', 'c', 'b', 'a']
```

Enfin, on a vu plus haut les problèmes que pouvaient causer les multiples références sur une même liste. Parfois, on veut simplement deux listes contenant les mêmes valeurs mais indépendantes l'une de l'autre, et l'on doit pour cela en réaliser une copie. Les listes possèdent pour cela une méthode `copy`.

```
1 >>> other_values = values.copy()
2 >>> values.append('g')
3 >>> values
4 ['a', 'b', 'c', 'd', 'e', 'f', 'g']
5 >>> other_values
6 ['a', 'b', 'c', 'd', 'e', 'f']
```

Ce même comportement est aussi possible en appelant `list` sur une liste existante, ou en utilisant un *slicing* vide.

```
1 >>> list(values)
2 ['a', 'b', 'c', 'd', 'e', 'f', 'g']
3 >>> values[:]
4 ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```



Attention cependant avec les listes multi-dimensionnelles : `copy` ne réalise une copie que du premier niveau de la liste.

Ainsi, avec le code qui suit, nous aurons encore des références communes entre les deux listes.

```
1 >>> values = [['a', 'b', 'c'], ['d', 'e', 'f']]
2 >>> other_values = values.copy()
3 >>> values[1].append('g')
4 >>> other_values
5 [['a', 'b', 'c'], ['d', 'e', 'f', 'g']]
```

Nous verrons par la suite comment réaliser une copie en profondeur et éviter ce problème.

Mais cela ne concerne bien sûr que les dimensions imbriquées : `values` et `other_values` restent deux listes distinctes.

```
1 >>> values.append(['h', 'i', 'j'])
2 >>> values
3 [['a', 'b', 'c'], ['d', 'e', 'f', 'g'], ['h', 'i', 'j']]
4 >>> other_values
5 [['a', 'b', 'c'], ['d', 'e', 'f', 'g']]
```

IV.1.5.4. Identité

Il existe en Python un opérateur d'identité, l'opérateur `is`. Celui-ci permet de tester si deux valeurs sont un seul et même objet, et non simplement des valeurs égales.

Il permet ainsi de savoir si deux variables pointent vers une même liste ou vers deux listes distinctes.

```
1 >>> values = [1, 2, 3]
2 >>> other_values = values
3 >>> other_values is values
4 True
5 >>> other_values = values.copy()
6 >>> other_values is values
7 False
8 >>> other_values == values
9 True
```

À l'inverse, on trouve l'opérateur `is not` pour tester la non-identité.

```
1 >>> other_values is not values
2 True
```

IV.1.6. None

`None` est une valeur particulière en Python, qui représente l'absence de valeur. On l'a déjà rencontrée sans vraiment y faire attention.

C'est par exemple la valeur renvoyée par les fonctions ou méthodes qui ne renvoient «rien».

```
1 >>> [].clear()
2 >>> print([].clear())
3 None
```

La fonction `print` en elle-même renvoie `None`.

```
1 >>> print(print())
2
3 None
```

Dans certains traitements, il est parfois utile de savoir si l'on a affaire à `None` ou à une autre valeur. Pour vérifier ça, on serait tenté de tester si notre valeur est égale à `None` avec un `==`. Mais `None` est une valeur unique en Python, il n'en existe qu'un (on parle de *singleton*) et on préférera donc utiliser l'opérateur d'identité : `is`.

```
1 >>> None is None
2 True
3 >>> [].clear() is None
4 True
5 >>> abs(-5) is None
6 False
```

On retrouve aussi l'opérateur `is not` pour vérifier qu'une valeur n'est pas `None`.

```
1 >>> abs(-5) is not None
2 True
3 >>> [].clear() is not None
4 False
```

IV.2. Les dictionnaires

Introduction

Les listes permettent de stocker un ensemble d'éléments en associant chaque élément à un index numérique.

Mais cela n'est pas adapté à toutes les données, toutes ne représentent pas une séquence de valeurs.

Comment par exemple représenter un répertoire téléphonique ? Avec une liste de listes où chaque sous-liste serait composée de deux éléments : un nom et un numéro ?

Par exemple on pourrait avoir

```
1 phonebook = [['Alice', '0633432380'], ['Bob', '0663621029'],  
               ['Alex', '0714381809']]
```

Ça fonctionnerait mais c'est loin d'être idéal, il faudrait se souvenir d'utiliser `[0]` pour le nom et `[1]` pour le numéro, et ça demanderait de parcourir toute la liste chaque fois que l'on voudrait chercher un numéro.

Heureusement pour nous, les dictionnaires sont une structure de données bien plus adaptée à ce genre de problématique.

IV.2.1. Des tables d'association

Comme les listes, les dictionnaires sont des conteneurs. C'est-à-dire qu'ils contiennent d'autres valeurs auxquelles on peut accéder avec l'opérateur d'indexation `[]`.

Mais plutôt qu'associer une valeur à un index, ils vont l'associer à une clé quelconque, par exemple une chaîne de caractères. Il s'agit donc d'un ensemble de couples clé-valeur.

Un dictionnaire se définit avec des accolades, entre lesquelles les couples sont séparés par des virgules. Un couple clé-valeur est de la forme **clé: valeur**.

Voilà à quoi pourrait ressembler le répertoire téléphonique donné en introduction :

```
1 phonebook = {'Alice': '0633432380', 'Bob': '0663621029', 'Alex':  
               '0714381809'}
```

C'est déjà plus clair à écrire, mais là où ça devient intéressant c'est pour l'accès aux éléments. On retrouve en effet l'opérateur `[]`, mais on va pouvoir lui préciser une clé de notre dictionnaire plutôt qu'un index.

```
1 >>> phonebook['Alex']  
2 '0714381809'
```

IV. Types de données

Il suffit de connaître le nom pour accéder au numéro, pas besoin de parcourir tout le répertoire. On comprend donc l'analogie avec le dictionnaire, qui permet d'associer des définitions à des mots, et de retrouver la définition à partir du mot.

IV.2.2. Opérations sur les dictionnaires

Les dictionnaires sont des objets modifiables, on retrouve donc l'opérateur d'indexation en lecture et en écriture.

```
1 >>> phonebook['Alice']
2 '0633432380'
3 >>> phonebook['Bob'] = '0712800331'
4 >>> del phonebook['Alex']
5 >>> phonebook
6 {'Alice': '0633432380', 'Bob': '0712800331'}
```

Par contre pas de *slicing* ici, cela n'a pas de sens sur des clés de dictionnaire. Une clé non trouvée dans le dictionnaire provoque une erreur.

```
1 >>> phonebook['Mehdi']
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   KeyError: 'Mehdi'
```

Les dictionnaires sont sensibles à la casse, c'est-à-dire que les lettres majuscules sont traitées différemment des minuscules. 'Alice' et 'alice' sont alors deux clés distinctes.

```
1 >>> phonebook['alice'] = '0729570663'
2 >>> phonebook
3 {'Alice': '0633432380', 'Bob': '0712800331', 'alice': '0729570663'}
```

On retrouve aussi l'opérateur d'appartenance (`in`), qui fonctionne sur les clés et non sur les valeurs.

```
1 >>> 'Bob' in phonebook
2 True
3 >>> '0633432380' in phonebook
4 False
```

Et on peut connaître la taille d'un dictionnaire en appelant la fonction `len`.

```
1 >>> len(phonebook)
2 3
```

Comme tout objet, il est possible de tester l'égalité entre deux dictionnaires avec l'opérateur `==`, et la différence avec `!=`. Deux dictionnaires sont considérés comme égaux s'ils contiennent les

IV. Types de données

mêmes éléments, avec les mêmes valeurs pour les mêmes clés.

```
1 >>> {'a': 1} == {'a': 1}
2 True
3 >>> {'a': 0} == {'b': 0}
4 False
5 >>> {'a': 0} != {'a': 0, 'b': 1}
6 True
```

Cela est vrai quel que soit l'ordre des éléments dans le dictionnaire.

```
1 >>> {'a': 0, 'b': 1} == {'b': 1, 'a': 0}
2 True
3 >>> {'a': 0, 'b': 1} != {'b': 1, 'a': 0, 'c': 2}
4 True
```

IV.2.2.1. Méthodes principales

Une première méthode intéressante est la méthode `get`. Elle agit comme l'opérateur `[]` mais sans produire d'erreur si la clé n'est pas trouvée.

```
1 >>> phonebook.get('Mehdi')
2 >>> print(phonebook.get('Mehdi'))
3 None
```

Comme on le voit, la valeur renvoyée si la clé n'est pas trouvée est `None`. Il est possible de renvoyer une autre valeur en la précisant comme second argument à `get`.

```
1 >>> phonebook.get('Mehdi', 'xxx')
2 'xxx'
```

Ensuite on va surtout trouver des méthodes pour modifier le dictionnaire, comme on en trouvait sur les listes.

La méthode `pop` est d'ailleurs équivalente à celle des listes, elle supprime une clé du dictionnaire et renvoie la valeur associée.

```
1 >>> phonebook.pop('Alice')
2 '0633432380'
3 >>> phonebook.pop('alice')
4 '0729570663'
```

L'appel produit une erreur si la clé n'est pas trouvée, mais il est là encore possible de donner une valeur par défaut en deuxième argument.

IV. Types de données

```
1 >>> phonebook.pop('Mehdi')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   KeyError: 'Mehdi'
5 >>> phonebook.pop('Mehdi', 'xxx')
6 'xxx'
```

La méthode `update` permet d'étendre le dictionnaire avec les données d'un autre dictionnaire.

```
1 >>> phonebook.update({'Julie': '0619096810', 'Mehdi':
2   '0762253973'})
3 >>> phonebook
{'Bob': '0712800331', 'Julie': '0619096810', 'Mehdi': '0762253973'}
```

Si une clé existe déjà dans le dictionnaire actuel, sa valeur est remplacée par la nouvelle qui est reçue.

```
1 >>> phonebook.update({'Julie': '0734593960'})
2 >>> phonebook
3 {'Bob': '0712800331', 'Julie': '0734593960', 'Mehdi': '0762253973'}
```

La méthode `clear` sert à vider complètement un dictionnaire.

```
1 >>> phonebook.clear()
2 >>> phonebook
3 {}
```

`{}` représente donc un dictionnaire vide.

Enfin, `setdefault` est un peu le pendant de `get` mais en écriture : elle va insérer une valeur dans le dictionnaire seulement si la clé n'est pas déjà présente.

La méthode renvoie la valeur associée à cette clé dans le dictionnaire, donc soit celle qui vient d'être ajoutée soit celle qui était déjà présente.

```
1 >>> phonebook.setdefault('Julie', '0619096810')
2 '0619096810'
3 >>> phonebook.setdefault('Julie', '0734593960')
4 '0619096810'
5 >>> phonebook
6 {'Julie': '0619096810'}
```

IV.2.2.2. Conversions

On peut convertir une liste de couples clé/valeur en un dictionnaire, en appelant `dict` comme une fonction.

IV. Types de données

Par exemple avec notre répertoire téléphonique défini en introduction :

```
1 >>> phonebook = [['Alice', '0633432380'], ['Bob', '0663621029'],  
2 ['Alex', '0714381809']]  
3 >>> dict(phonebook)  
4 {'Alice': '0633432380', 'Bob': '0663621029', 'Alex': '0714381809'}
```

L'appel à `dict` sur un dictionnaire existant permet aussi d'en créer une copie.

```
1 >>> phonebook = {'Mehdi': '0762253973'}  
2 >>> mybook = dict(phonebook)  
3 >>> mybook['Julie'] = '0734593960'  
4 >>> mybook  
5 {'Mehdi': '0762253973', 'Julie': '0734593960'}  
6 >>> phonebook  
7 {'Mehdi': '0762253973'}
```

Enfin, une autre utilité de l'appel à `dict` est de pouvoir construire un dictionnaire à partir d'arguments nommés. Les noms des arguments deviennent ainsi les clés du dictionnaire.

```
1 >>> dict(Bob='0712800331', Julie='0734593960', Mehdi='0762253973')  
2 {'Bob': '0712800331', 'Julie': '0734593960', 'Mehdi': '0762253973'}
```

IV.2.3. Données composites

Un autre cas d'utilisation des dictionnaires est celui d'agréger dans un même objet plusieurs valeurs liées les unes aux autres, plutôt que dans des variables différentes. Pensez par exemple à la représentation des monstres dans notre TP : on a un nom d'un côté, un nombre de PV de l'autre et aussi une liste d'attaques.

À la place, on pourrait utiliser un dictionnaire par monstre, en y faisant figurer toutes ses données.

```
1 {  
2     'nom': 'Pythachu',  
3     'PV': 50,  
4     'attaques': ['tonnerre', 'charge'],  
5 }
```

En effet, tous les types de données sont acceptés en tant que valeurs, et toutes les valeurs n'ont pas besoin d'être du même type.

Mais on peut faire encore mieux. En usant de listes et de dictionnaires, on construit facilement des structures arborescentes pour représenter toutes nos données.

IV. Types de données

```
1 monstres = {
2     'Pythachu': {
3         'type': 'foudre',
4         'description': 'Petit rat surchargé.',
5         'attaques': ['tonnerre', 'charge'],
6     },
7     'Pythard': {
8         'type': 'aquatique',
9         'description': "Tétard qui a cru qu'il était tôt.",
10        'attaques': ['jet-de-flotte', 'charge'],
11    },
12    'Ponytha': {
13        'type': 'flamme',
14        'description': 'Cheval enflammé.',
15        'attaques': ['brûlure', 'charge'],
16    },
17 }
18
19 attaques = {
20     'charge': {'degats': 20},
21     'tonnerre': {'degats': 50},
22     'jet-de-flotte': {'degats': 40},
23     'brûlure': {'degats': 40},
24 }
25
26 joueurs = [
27     {
28         'nom': 'Joueur 1',
29         'monstre': 'Pythachu',
30         'PV': 100,
31     },
32     {
33         'nom': 'Joueur 2',
34         'monstre': 'Ponytha',
35         'PV': 120,
36     },
37 ]
```

Ainsi, on représente dans des variables différentes la structure de nos données. Pour avoir d'un côté la définition des monstres et des attaques, et de l'autre les monstres en jeu.

```
1 >>> print(joueurs[0]['nom'], ': ', joueurs[0]['monstre'])
2 Joueur 1 : Pythachu
3 >>> print('Attaques : ',
4         monstres[joueurs[0]['monstre']]['attaques'])
5 Attaques : ['tonnerre', 'charge']
6 >>> print('Dégâts de tonnerre : ', attaques['tonnerre']['degats'])
7 Dégâts de tonnerre : 50
```

IV.2.4. Clés de dictionnaires

Jusqu'ici, je n'ai présenté que des chaînes de caractères comme clés de dictionnaire, par souci de simplicité.

Mais ce ne sont pas les seuls types de clés possibles, les booléens ou les nombres sont aussi des clés valides.

```
1 choices = {
2     True: 'OK',
3     False: 'KO',
4 }
5
6 diviseurs = {
7     4: [1, 2],
8     6: [1, 2, 3],
9     8: [1, 2, 4],
10    9: [1, 3],
11    10: [1, 2, 5],
12 }
```

Qui s'utilisent de la même manière lors de l'indexation.

```
1 >>> choices[True]
2 'OK'
3 >>> choices[False] = 'erreur'
4 >>> diviseurs[8]
5 [1, 2, 4]
6 >>> diviseurs.get(5, [1])
7 [1]
```



Dans le cas de la construction d'un dictionnaire à l'aide d'un appel du type `dict(a=0, b=1)`, les clés seront forcément des chaînes de caractères et doivent correspondre à des noms valides.

Il n'est alors pas possible d'écrire quelque chose comme `dict(9=[1, 3])` puisque 9 n'est pas un nom d'argument valide.

En revanche, tout type de données n'est pas accepté comme clé de dictionnaire. Vous avez vu vous en rendre compte si vous avez essayé d'y placer une liste ou un dictionnaire.

```
1 >>> {[]: 0}
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: unhashable type: 'list'
```

Nous reviendrons plus tard sur cette erreur et ce qu'elle signifie, mais retenez pour le moment que dans les types que nous connaissons seuls les non-modifiables peuvent être utilisés en tant

IV. Types de données

que clés.

Les valeurs modifiables telles que les listes ou les dictionnaires ne peuvent pas être utilisées en tant que clés, car comment retrouverait-on la valeur associée si la clé est mise à jour ?

IV.3. Itérer sur un dictionnaire

Introduction

On a vu que la boucle `for` ne se limitait pas aux listes : elle permet aussi d'itérer sur les chaînes de caractères et les *range* par exemple. Plus généralement, un `for` itère sur un objet dit itérable.

Et ça tombe bien : les dictionnaires sont itérables !

IV.3.1. Dictionnaire et boucle for

Mais itérer sur un dictionnaire, qu'est-ce que ça veut dire ?

En fait un dictionnaire peut être vu comme un ensemble de clés. Itérer sur un dictionnaire revient donc à itérer sur ces clés.

```
1 >>> phonebook = {'Alice': '0633432380', 'Bob': '0663621029',  
2   'Alex': '0714381809'}  
3 >>> for name in phonebook:  
4     ...     print(name)  
5 Alice  
6 Bob  
7 Alex
```

Et à partir d'une clé, il est facilement possible d'accéder à la valeur associée, grâce à l'opérateur `[]`.

```
1 >>> for name in phonebook:  
2     ...     print(name, ': ', phonebook[name])  
3  
4 Alice : 0633432380  
5 Bob : 0663621029  
6 Alex : 0714381809
```



Si vous utilisez une version de Python antérieure à 3.6, il se peut que vous obteniez un ordre différent dans les itérations. En effet, l'ordre des éléments dans un dictionnaire était auparavant aléatoire.

Le fait qu'un dictionnaire soit itérable le rend donc convertible en liste (en appelant `list`) ce qui aura pour effet de renvoyer la liste des clés.

```
1 >>> list(phonebook)
2 ['Alice', 'Bob', 'Alex']
```

IV.3.2. Autres manières d'itérer

On sait itérer sur les clés et récupérer la valeur associée à chaque clé, mais est-ce qu'il n'y a pas plus simple ? En effet, imaginons que nous ne soyons intéressés que par les valeurs du dictionnaire, pourquoi s'encombrer avec les clés ?

Les dictionnaires possèdent pour cela une méthode `values` renvoyant l'ensemble des valeurs du dictionnaire, sans les clés.

```
1 >>> phonebook.values()
2 dict_values(['0633432380', '0663621029', '0714381809'])
```

La méthode renvoie un objet d'un type un peu spécial, `dict_values`. Il s'agit en fait d'une «vue», une sorte de liste qui n'existe pas en tant que telle en mémoire (il n'y a pas de séquence d'éléments) mais qui sait où aller chercher ses éléments.

Il n'y a donc pas de duplication des données, la vue référence juste les valeurs du dictionnaire.

Et cet objet un peu spécial est bien sûr itérable :

```
1 >>> for phone in phonebook.values():
2 ...     print('Numéro de téléphone :', phone)
3 ...
4 Numéro de téléphone : 0633432380
5 Numéro de téléphone : 0663621029
6 Numéro de téléphone : 0714381809
```

De façon symétrique on trouve aussi une méthode `keys` pour renvoyer une vue sur les clés.

```
1 >>> phonebook.keys()
2 dict_keys(['Alice', 'Bob', 'Alex'])
```

Itérer sur cette vue revient donc à itérer directement sur le dictionnaire, mais comme on dit «explicit is better than implicit»¹.

```
1 >>> for name in phonebook.keys():
2 ...     print(name)
3 ...
4 Alice
5 Bob
6 Alex
```

1. Il s'agit d'un «vers» extrait de la [PEP20](#), un «poème» qui décrit la philosophie de Python.

IV. Types de données

Enfin, les dictionnaires disposent d'une troisième vue très utile, la vue `items`. Cette vue renvoie les couples clé/valeur du dictionnaire.

```
1 >>> phonebook.items()
2 dict_items([('Alice', '0633432380'), ('Bob', '0663621029'),
3           ('Alex', '0714381809')])
```

Étant donné qu'il s'agit de couples, on peut itérer sur cette vue en précisant deux variables dans notre `for` : une pour recevoir la clé et une pour la valeur.

```
1 >>> for name, phone in phonebook.items():
2     ...     print(name, ': ', phone)
3     ...
4 Alice : 0633432380
5 Bob : 0663621029
6 Alex : 0714381809
```

IV.3.2.1. Conversions

On l'a vu, on peut convertir un dictionnaire en liste, ce qui nous renvoie la liste de ses clés. Les différentes vues que nous venons de voir sont elles aussi convertibles. Plus généralement, tout objet itérable (que l'on peut parcourir avec un `for`) est convertible en liste via un appel à `list`.

```
1 >>> list(phonebook.keys())
2 ['Alice', 'Bob', 'Alex']
3 >>> list(phonebook.values())
4 ['0633432380', '0663621029', '0714381809']
5 >>> list(phonebook.items())
6 [('Alice', '0633432380'), ('Bob', '0663621029'), ('Alex',
7           '0714381809')]
```

?

Que représentent les parenthèses dans le dernier exemple ? La réponse est dans le prochain chapitre. 🍊

IV.4. Les tuples

Introduction

On vient de voir que la méthode `items` des dictionnaires renvoyait des couples de valeurs. Mais qu'est-ce que c'est au juste qu'un couple, un nouveau type ?

Oui, il s'agit d'un tuple.

IV.4.1. Les tuples

Les tuples (parfois traduits en n-uplets) sont des équivalents non modifiables aux listes. C'est-à-dire des séquences d'un nombre fixe d'éléments : après la définition, on ne peut ni ajouter, ni supprimer, ni remplacer d'élément.

Un tuple est généralement défini par une paire de parenthèses contenant les éléments séparés par des virgules. Comme une liste, un tuple peut contenir des éléments de types différents.

```
1 >>> (1, 2, 3)
2 (1, 2, 3)
3 >>> ('a', 'b', 'c')
4 ('a', 'b', 'c')
5 >>> (42, '!')
6 (42, '!')
```

On notera tout de même que les parenthèses sont facultatives, c'est la virgule qui définit réellement un tuple. Comme pour les opérations arithmétiques, les parenthèses servent en fait à gérer les priorités et mettre en valeur le tuple.

```
1 >>> 1, 2, 3
2 (1, 2, 3)
3 >>> 1, 2, 3 * 3
4 (1, 2, 9)
5 >>> (1, 2, 3) * 3
6 (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Il faut bien penser à cette virgule lorsque l'on cherche à définir un tuple contenant un unique élément. En effet, `(1)` étant une notation équivalente à `1`, il est nécessaire d'en ajouter une pour expliciter le tuple.

```
1 >>> (1)
2 1
3 >>> (1,)
```

IV. Types de données

```
4 (1,)
5 >>> 1,
6 (1,)
```

Par ailleurs, il est possible de définir un tuple vide à l'aide d'une simple paire de parenthèses (il n'y a dans ce cas pas de confusion avec d'autres utilisations possibles des parenthèses).

```
1 >>> ()
2 ()
```

J'utiliserai principalement le terme de tuple, mais il faut savoir qu'on rencontre parfois d'autres noms suivant la taille du tuple. On parle ainsi parfois de couple pour des tuples de 2 éléments, des triplets pour 3, etc.

Par exemple il est courant de dire que la méthode `items` des dictionnaires renvoie des couples clé/valeur.

```
1 >>> phonebook = {'Alice': '0633432380', 'Bob': '0663621029',
2                  'Alex': '0714381809'}
3 >>> for couple in phonebook.items():
4     ...     print(couple)
5     ...
6 ('Alice', '0633432380')
7 ('Bob', '0663621029')
8 ('Alex', '0714381809')
```

IV.4.2. Opérations sur les tuples

Les opérations sont semblables à celles des listes.

Il est possible de convertir un itérable en tuple en appelant le type `tuple` comme une fonction.

```
1 >>> tuple([1, 2, 3])
2 (1, 2, 3)
3 >>> tuple('abcd')
4 ('a', 'b', 'c', 'd')
```

On peut accéder aux éléments d'un tuple (en lecture uniquement) avec l'opérateur d'indexation `[]`, qui gère les index négatifs et les slices.

```
1 >>> values = (4, 5, 6)
2 >>> values[1]
3 5
4 >>> values[-1]
5 6
6 >>> values[:2]
```


IV. Types de données

```
7 (4, 6)
```

`in` permet de vérifier si une valeur est présente dans le tuple.

```
1 >>> 3 in values
2 False
3 >>> 4 in values
4 True
```

On peut concaténer deux tuples avec `+`, et multiplier un tuple par un nombre avec `*`.

```
1 >>> (4, 5, 6) + (7, 8, 9)
2 (4, 5, 6, 7, 8, 9)
3 >>> (4, 5, 6) * 2
4 (4, 5, 6, 4, 5, 6)
```

Les tuples sont ordonnables les uns par rapport aux autres, de la même manière que les listes.

```
1 >>> (1, 2, 3) < (1, 2, 4)
2 True
3 >>> (1, 2, 3) <= (1, 2, 2)
4 False
```

Les fonctions `len`, `min`, `max`, `all`, `any` etc. sont aussi applicables aux tuples.

```
1 >>> len(values)
2 3
3 >>> min(values)
4 4
5 >>> max(values)
6 6
7 >>> all((True, True, False))
8 False
9 >>> any((True, True, False))
10 True
```

Enfin, les tuples sont pourvus de deux méthodes, `index` et `count`, pour respectivement trouver la position d'un élément et compter les occurrences d'un élément.

```
1 >>> values.index(6)
2 2
3 >>> values.count(6)
4 1
```

IV.4.3. Utilisations des tuples

On peut parfois se demander quand utiliser un tuple et quand utiliser une liste. Le tuple étant comparable à une liste non modifiable, il peut donc être utilisé pour toutes les opérations attendant une liste et ne cherchant pas à la modifier. Il est même préférable de l'utiliser si l'on sait qu'il ne sera jamais question de modification, ou si l'on veut empêcher toute modification. Mais il y a aussi des opérations qui ne sont possibles qu'avec les tuples. Étant non modifiables, ils peuvent être utilisés en tant que clés de dictionnaires.

```
1 >>> cells = {(0, 0): 'x', (0, 1): '.', (1, 0): '.', (1, 1): 'x'}
2 >>> cells[(1, 0)]
3 '.'
4 >>> cells[(1, 1)] = ' '
5 >>> cells
6 {(0, 0): 'x', (0, 1): '.', (1, 0): '.', (1, 1): ' '}
```

Les parenthèses sont encore une fois facultatives lors des accès.

```
1 >>> cells[0, 0]
2 'x'
```

Cette structure permet ainsi de représenter d'une grille de données où chaque case est associée à ses coordonnées.



Attention cependant, un tuple qui contient un élément modifiable pourra lui aussi être indirectement altéré.

Par exemple si un tuple contient une liste, rien n'empêche d'ajouter des éléments à cette liste.

```
1 >>> events = ('29/05/2019', ['anniversaire', 'soirée'])
2 >>> events[1].append('rdv coiffeur')
3 >>> events
4 ('29/05/2019', ['anniversaire', 'soirée', 'rdv coiffeur'])
```

Un tel tuple ne pourra donc pas être utilisé comme clé de dictionnaire, parce qu'il contient une liste qui ne peut pas être utilisée comme tel.

```
1 >>> {events: None}
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: unhashable type: 'list'
```

Mais beaucoup d'utilisations des tuples sont tous simplement implicites. C'est en effet une manière de faire des assignations multiples de variables.

IV. Types de données

```
1 >>> a, b = 1, 2
2 >>> a
3 1
4 >>> b
5 2
```

Techniquement, cela revient à écrire `(a, b) = (1, 2)`, ou encore :

```
1 >>> tmp = (1, 2)
2 >>> (a, b) = tmp
```

i

On appelle cette seconde opération (assigner plusieurs variables depuis un tuple) l'*unpacking*, mais nous y reviendrons plus tard.

IV.5. TP

Introduction

L'objectif maintenant va être de reprendre notre code et de placer des dictionnaires aux endroits adéquats. Notamment remplacer les multiples variables associées à un seul monstre par des dictionnaires représentant la hiérarchie de nos données.

IV.5.1. Structurer les données

Un monstre devient alors une structure avec un nom et une liste d'attaques. Et une attaque se compose simplement d'un nombre de dégâts.

Dans le chapitre sur les dictionnaires, je présentais comment les utiliser pour former des données composites, avec un exemple correspondant à notre TP. En repartant de cet exemple on va pouvoir structurer facilement nos monstres et nos attaques.

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }
```

L'avantage de cette structure c'est que les attaques sont pleinement séparées des monstres, ce qui permet de ne pas les répéter quand elles sont partagées entre plusieurs monstres.

Le second avantage c'est qu'on est maintenant en mesure de définir précisément quel monstre peut utiliser quelle attaque, et donc d'avoir une meilleure validation à ce niveau.

Pour commencer le jeu, on demandera toujours aux deux joueurs de choisir leur monstre et

IV. Types de données

d'indiquer les points de vie associés, mais on pourra maintenant vérifier qu'il s'agit d'un monstre que l'on connaît.

```
1 print('Monstres disponibles :')
2 for monster in monsters.values():
3     print('-', monster['name'])
4
5 players = []
6
7 print('Joueur 1, quel monstre choisissez-vous ?')
8 name = input('> ').lower()
9 while name not in monsters:
10     print('Monstre invalide')
11     name = input('> ').lower()
12 pv = int(input('Quel est son nombre de PV ? '))
13 players.append({'id': '1', 'monster': name, 'pv': pv})
```

Le fait d'utiliser des dictionnaires pour représenter nos joueurs nous permet aussi de les stocker dans une liste plutôt que dans deux variables distinctes, et donc d'éviter les répétitions que l'on avait dans nos traitements (puisque l'on va pouvoir appliquer une boucle sur cette liste).

IV.5.2. Solution

On retrouve maintenant la solution à ce TP, qui se rapproche de plus en plus d'un vrai système de combat.

👁 Contenu masqué n°8

À l'exécution on a bien quelque chose d'assez clair sur le déroulement du jeu.

```
1 Monstres disponibles :
2 - Pythachu
3 - Pythard
4 - Ponytha
5 Joueur 1 quel monstre choisissez-vous ?
6 > Pythachu
7 Quel est son nombre de PV ? 100
8 Joueur 2 quel monstre choisissez-vous ?
9 > Ponytha
10 Quel est son nombre de PV ? 120
11
12 Pythachu affronte Ponytha
13
14 Joueur 1 quelle attaque utilisez-vous ?
15 - Tonnerre -50 PV
16 - Charge -20 PV
17 > tonnerre
```

IV. Types de données

```
18 Pythachu attaque Ponytha qui perd 50 PV, il lui en reste 70
19 Joueur 2 quelle attaque utilisez-vous ?
20 - Brûlure -40 PV
21 - Charge -20 PV
22 > charge
23 Ponytha attaque Pythachu qui perd 20 PV, il lui en reste 80
24 Joueur 1 quelle attaque utilisez-vous ?
25 - Tonnerre -50 PV
26 - Charge -20 PV
27 > charge
28 Pythachu attaque Ponytha qui perd 20 PV, il lui en reste 50
29 Joueur 2 quelle attaque utilisez-vous ?
30 - Brûlure -40 PV
31 - Charge -20 PV
32 > brulure
33 Attaque invalide
34 > brûlure
35 Ponytha attaque Pythachu qui perd 40 PV, il lui en reste 40
36 Joueur 1 quelle attaque utilisez-vous ?
37 - Tonnerre -50 PV
38 - Charge -20 PV
39 > tonnerre
40 Pythachu attaque Ponytha qui perd 50 PV, il lui en reste 0
41 Le joueur 1 remporte le combat avec Pythachu
```

Contenu masqué

Contenu masqué n°8

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
```

```

18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }
22
23 print('Monstres disponibles :')
24 for monster in monsters.values():
25     print('-', monster['name'])
26
27 players = []
28
29 # Boucle pour créer 2 joueurs sans se répéter
30 for i in range(2):
31     player_id = i + 1
32     print('Joueur', player_id, 'quel monstre choisirez-vous ?')
33
34     name = input('> ').lower()
35     while name not in monsters:
36         print('Monstre invalide')
37         name = input('> ').lower()
38
39     pv = int(input('Quel est son nombre de PV ? '))
40     players.append({'id': player_id, 'monster': monsters[name],
41                    'pv': pv})
42
43 print()
44 print(players[0]['monster']['name'], 'affronte',
45        players[1]['monster']['name'])
46 print()
47
48 # Représente les tours de jeu, liste de couples (joueur, opposant)
49 turns = [
50     (players[0], players[1]),
51     (players[1], players[0]),
52 ]
53
54 while players[0]['pv'] > 0 and players[1]['pv'] > 0:
55     # On effectue les deux tours de jeu
56     for player, opponent in turns:
57         # Le joueur ne peut jouer que s'il n'est pas KO
58         if player['pv'] > 0:
59             print('Joueur', player['id'],
60                   'quelle attaque utilisez-vous ?')
61             for name in player['monster']['attacks']:
62                 print('-', name.capitalize(),
63                       '-attacks[name][\'damages\'], \'PV\')
64
65             att_name = input('> ').lower()
66             while att_name not in attacks:
67                 print('Attaque invalide')

```

```
64         att_name = input('> ').lower()
65         attack = attacks[att_name]
66
67         opponent['pv'] -= attack['damages']
68
69         print(
70             player['monster']['name'],
71             'attaque',
72             opponent['monster']['name'],
73             'qui perd',
74             attack['damages'],
75             'PV, il lui en reste',
76             opponent['pv'],
77         )
78
79     if players[0]['pv'] > players[1]['pv']:
80         winner = players[0]
81     else:
82         winner = players[1]
83
84     print('Le joueur', winner['id'], 'remporte le combat avec',
          winner['monster']['name'])
```

[Retourner au texte.](#)

Cinquième partie

Les fonctions

Introduction

Nous avons déjà rencontré des fonctions au long de ce tutoriel : `print`, `len` ou `round` en sont des exemples.

Dans cette partie il va maintenant être question d'écrire nos propres fonctions, afin de pouvoir les appeler et d'obtenir les comportements voulus suivant les arguments qui leur seront passés.

V.1. Des fonctions pour factoriser

Introduction

Mais au juste pourquoi vouloir écrire des fonctions, à quoi ça sert ?

V.1.1. Don't Repeat Yourself

L'idée première, c'est d'éviter de se répéter, de dupliquer du code. En effet, il y a plusieurs portions du code de notre TP qui pourraient être mises en commun et qui nous ferait gagner en clarté.

Un code dupliqué, c'est un code plus difficile à maintenir. Déjà, il est plus long à lire et il faut garder plus d'éléments de contexte en tête pour le comprendre. Mais ça alourdit aussi les étapes de réécriture et/ou de correction.

Quand un code est présent à un seul endroit, il n'y a que cet endroit à réécrire pour l'adapter. Quand il est dupliqué à dix emplacements différents, il devient plus difficile d'aller tous les corriger. Et une erreur peut facilement s'y glisser, si l'on oublie l'un de ces emplacements.

En factorisant le code, on isole les portions logiques que l'on peut ainsi plus facilement retrouver. On divise ainsi un programme en plusieurs petites portions de code qu'il est aisé de relire indépendamment les unes des autres. On verra aussi par la suite qu'un code divisé en fonctions est plus facile à tester.

V.1.2. Factoriser

La question va maintenant être de savoir comment identifier et réunir les portions logiques pour éviter les répétitions.

Mais souvent, on ne sera pas face à deux codes strictement identiques, ils seront simplement similaires. L'idée sera alors de travailler à les rendre identiques afin de les factoriser en une unique fonction. Pour cela, il faudra identifier les paramètres variables qui agissent sur le code et le font se différencier. Une fois ces paramètres isolés et placés dans des variables avant la portion concernée, le code deviendra factorisable.

Prenons l'exemple suivant, qui affiche les tables de multiplication de 3 et de 5.

```
1 for i in range(1, 11):
2     print(3, 'x', i, '=', 3*i)
3
4 for i in range(1, 11):
5     print(5, 'x', i, '=', 5*i)
```

On a deux boucles très semblables qui ne sont pourtant pas identiques. Un paramètre diffère entre les deux, le nombre par lequel on multiplie. Si l'on isole ce nombre dans une variable, on

V. Les fonctions

constate que nos deux boucles deviennent identiques.

```
1 n = 3
2 for i in range(1, 11):
3     print(n, 'x', i, '=', n*i)
4
5 n = 5
6 for i in range(1, 11):
7     print(n, 'x', i, '=', n*i)
```

On pourrait alors écrire une fonction pour réaliser ces opérations, paramétrée selon la valeur de `n`.

C'est donc en ça que consiste le travail de factorisation : œuvrer pour que des codes similaires mais différents puisse utiliser une fonction commune.

V.1.2.1. Identifier les portions logiques dans un code plus complet

Pour s'exercer sur un cas d'usage réel, on peut reprendre le code du dernier TP que je réinsère ci-dessous. Notre travail va alors être d'identifier les différentes sections qui composent notre programme et qui pourront donc être séparées en fonctions.

☞ Contenu masqué n°9

À sa lecture on distingue ainsi plusieurs blocs avec des logiques bien distinctes :

- Lignes 1 à 21, la définition de nos données.
- Lignes 23 à 25, l'affichage des monstres existants.
- Lignes 27 à 44, la sélection des joueurs, et plus particulièrement :
 - Lignes 34 à 40, la sélection d'un joueur.
 - Lignes 34 à 37, la validation des monstres.
 - Lignes 42 à 44, l'affichage des monstres en jeu.
- Lignes 46 à 50, la définition des tours de jeu.
- Lignes 52 à 77, la boucle jeu et le déroulement des combats, notamment :
 - Lignes 57 à 77, le déroulement du combat pour un joueur.
 - Lignes 57 à 65, le choix d'une attaque.
 - Lignes 67 à 77, l'application d'une attaque (avec affichage).
- Lignes 79 à 84, la désignation du vainqueur.

L'écriture des fonctions correspondant à ces différentes logiques sera l'objet du prochain TP.

V.1.3. Définir une fonction (bloc `def`)

On définit une fonction à l'aide du mot-clé `def` qui introduit un bloc. On le fait suivre du nom de la fonction, d'une paire de parenthèses et d'un signe `:`. Toutes les lignes indentées qui suivent appartiendront au corps de la fonction.

```
1 def toto():
2     print('Hello')
```

```
3 print('World!')
```

Ainsi, les lignes ne sont pas directement exécutées. Le bloc de code précédent ne fait que créer une nouvelle fonction connue sous le nom de `toto`. Son contenu sera exécuté lors de l'appel à la fonction.

Le nom d'une fonction est similaire à celui d'une variable, et doit donc se soumettre aux mêmes règles : composé uniquement de lettres, de chiffres et d'*underscores* (`_`), et ne pouvant pas commencer par un chiffre.

V.1.4. Appel de fonction

On appelle notre fonction comme toute autre fonction (sans arguments pour le moment), en faisant suivre son nom d'une paire de parenthèses.

```
1 >>> toto()  
2 Hello  
3 World!
```

Le code contenu dans le bloc de notre fonction est alors exécuté, c'est pourquoi nous voyons s'afficher les messages passés à `print`.

Tout le code de la fonction sera à nouveau exécuté à chaque nouvel appel, chaque appel étant indépendant des autres.

```
1 >>> toto()  
2 Hello  
3 World!  
4 >>> toto()  
5 Hello  
6 World!
```

Ce sont les parenthèses qui demandent à Python d'appeler la fonction et d'en exécuter le contenu. Sans elles, l'interpréteur ne ferait qu'évaluer l'expression `toto` pour nous indiquer qu'elle correspond à une fonction.

```
1 >>> toto  
2 <function toto at 0x7fea64fcf1f0>
```



Ne vous souciez pas de cette valeur `0x7fea64fcf1f0` qui apparaît derrière et qui différera sûrement chez vous, il ne s'agit que de l'emplacement en mémoire de la fonction.

Contenu masqué

Contenu masqué n°9

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }
22
23 print('Monstres disponibles :')
24 for monster in monsters.values():
25     print('-', monster['name'])
26
27 players = []
28
29 # Boucle pour créer 2 joueurs sans se répéter
30 for i in range(2):
31     player_id = i + 1
32     print('Joueur', player_id, 'quel monstre choisissez-vous ?')
33
34     name = input('> ').lower()
35     while name not in monsters:
36         print('Monstre invalide')
37         name = input('> ').lower()
38
39     pv = int(input('Quel est son nombre de PV ? '))
40     players.append({'id': player_id, 'monster': monsters[name],
41                    'pv': pv})
42
43 print()
44 print(players[0]['monster']['name'], 'affronte',
45        players[1]['monster']['name'])
```

```

44 print()
45
46 # Représente les tours de jeu, liste de couples (joueur, opposant)
47 turns = [
48     (players[0], players[1]),
49     (players[1], players[0]),
50 ]
51
52 while players[0]['pv'] > 0 and players[1]['pv'] > 0:
53     # On effectue les deux tours de jeu
54     for player, opponent in turns:
55         # Le joueur ne peut jouer que s'il n'est pas KO
56         if player['pv'] > 0:
57             print('Joueur', player['id'],
58                   'quelle attaque utilisez-vous ?')
59             for name in player['monster']['attacks']:
60                 print('-', name.capitalize(),
61                       '-attacks[name][\'damages\'], \'PV\')
62
63             att_name = input('> ').lower()
64             while att_name not in attacks:
65                 print('Attaque invalide')
66                 att_name = input('> ').lower()
67             attack = attacks[att_name]
68
69             opponent['pv'] -= attack['damages']
70
71             print(
72                 player['monster']['name'],
73                 'attaque',
74                 opponent['monster']['name'],
75                 'qui perd',
76                 attack['damages'],
77                 'PV, il lui en reste',
78                 opponent['pv'],
79             )
80
81 if players[0]['pv'] > players[1]['pv']:
82     winner = players[0]
83 else:
84     winner = players[1]
85
86 print('Le joueur', winner['id'], 'remporte le combat avec',
87       winner['monster']['name'])

```

[Retourner au texte.](#)

V.2. Fonctions paramétrées

V.2.1. Paramètres de fonction

On sait définir et appeler une fonction, mais on obtient toujours la même chose à chaque appel. Il serait bien de pouvoir faire varier le comportement d'une fonction suivant les valeurs de certaines expressions, et c'est là qu'interviennent les paramètres.

Le paramètre est une variable définie dans la fonction qui recevra une valeur lors de chaque appel. Cette valeur pourra être de tout type, suivant ce qui est fourni en argument.

Les noms des paramètres sont inscrits lors de la définition de la fonction, entre les parenthèses qui suivent son nom.

```
1 def table_multiplication(n):
2     for i in range(1, 11):
3         print(n, 'x', i, '=', n*i)
```

Encore une fois, les paramètres sont des variables et donc suivent les mêmes règles de nomenclature. S'il y a plusieurs paramètres, ils doivent être séparés par des virgules.

```
1 def hello(firstname, lastname):
2     print('Hello', firstname, lastname, '!')
```

Lors de l'appel de la fonction, on utilisera les arguments pour donner leurs valeurs aux paramètres. On précise ainsi les valeurs dans les parenthèses qui suivent le nom de la fonction.

```
1 >>> table_multiplication(3)
2 3 x 1 = 3
3 3 x 2 = 6
4 3 x 3 = 9
5 3 x 4 = 12
6 3 x 5 = 15
7 3 x 6 = 18
8 3 x 7 = 21
9 3 x 8 = 24
10 3 x 9 = 27
11 3 x 10 = 30
12 >>> table_multiplication(5)
13 5 x 1 = 5
14 5 x 2 = 10
15 5 x 3 = 15
16 5 x 4 = 20
```


V. Les fonctions

```
17 5 × 5 = 25
18 5 × 6 = 30
19 5 × 7 = 35
20 5 × 8 = 40
21 5 × 9 = 45
22 5 × 10 = 50
```

Le comportement est le même pour les fonctions à plusieurs paramètres, les valeurs leur sont attribuées dans l'ordre des arguments : le premier paramètre prend la valeur du premier argument, etc.

Chaque argument correspond ainsi à un paramètre (et inversement).

```
1 >>> hello('Père', 'Noël')
2 Hello Père Noël !
3 >>> hello('Blanche', 'Neige')
4 Hello Blanche Neige !
```

Une erreur survient s'il n'y a pas assez d'arguments pour compléter tous les paramètres.

```
1 >>> hello()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: hello() missing 2 required positional arguments:
      'firstname' and 'lastname'
5 >>> hello('Asterix')
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: hello() missing 1 required positional argument:
      'lastname'
```

Au contraire, une autre erreur est levée s'il y a trop d'arguments par rapport au nombre de paramètres.

```
1 >>> hello('Homer', 'Jay', 'Simpson')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: hello() takes 2 positional arguments but 3 were given
```

Derrière ces exemples simples, il est bien sûr possible d'avoir de vrais comportements qui dépendent des valeurs de nos paramètres.

```
1 def print_div(a, b):
2     if b == 0:
3         print('Division impossible')
4     else:
5         print(a / b)
```

```
1 >>> print_div(5, 2)
2 2.5
3 >>> print_div(1, 0)
4 Division impossible
```

V.2.2. Espace de noms

Chaque fonction comporte son propre espace de noms (ou scope), c'est-à-dire une entité qui contient toutes les définitions de variables. Ainsi, une variable définie à l'intérieur d'une fonction n'existe que dans celle-ci.

```
1 >>> def f():
2 ...     a = 5
3 ...     print(a)
4 ...
5 >>> a
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   NameError: name 'a' is not defined
```

Que ce soit avant ou après l'appel de la fonction, la variable `a` n'existe pas dans l'espace de noms principal (ou global).

```
1 >>> f()
2 5
3 >>> a
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   NameError: name 'a' is not defined
```

Il en est de même pour les paramètres qui sont au final des variables comme les autres au sein de la fonction.

```
1 >>> def f(x):
2 ...     print(x)
3 ...
4 >>> f(5)
5 5
6 >>> x
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   NameError: name 'x' is not defined
```

Il est en revanche possible pour une fonction d'accéder aux variables définies à l'extérieur de celle-ci.

V. Les fonctions

```
1 >>> value = 42
2 >>>
3 >>> def f():
4 ...     print(value)
5 ...
6 >>> f()
7 42
```

Ce qui implique que le comportement de la fonction change si la valeur de la variable est modifiée.

```
1 >>> value = 13
2 >>> f()
3 13
```

Mais les espaces de noms extérieur et intérieur à la fonction sont bien deux scopes distincts. Deux variables de même nom peuvent exister dans des scopes différents sans qu'elles n'interfèrent entre elles.

```
1 >>> x = 0
2 >>>
3 >>> def f():
4 ...     x = 1
5 ...     print(x)
6 ...
7 >>> x
8 0
9 >>> f()
10 1
11 >>> x
12 0
```

Ce qui implique qu'il n'est pas possible de redéfinir une variable extérieure à la fonction (du moins pas de cette manière) car Python croira toujours que l'on cherche à définir une variable locale.

Cela peut poser problème si l'on essaie dans une fonction d'accéder à une variable avant de la redéfinir. En effet, Python ne saura pas si l'on souhaite récupérer une variable extérieure portant ce nom (puisque'elle n'aura pas encore été définie dans le scope local) ou en définir une nouvelle. Il lèvera donc une erreur pour éviter toute ambiguïté.

```
1 >>> def f():
2 ...     print(x)
3 ...     x = 1
4 ...
5 >>> f()
6 Traceback (most recent call last):
```

```
7 File "<stdin>", line 1, in <module>
8 File "<stdin>", line 2, in f
9 UnboundLocalError: local variable 'x' referenced before assignment
```

V.2.3. Arguments positionnels et nommés

Nous n'avons vu pour le moment que les arguments positionnels. C'est-à-dire dont l'association avec les paramètres se fait par la position de l'argument (n-ième argument pour le n-ième paramètre).

Mais il est aussi possible de spécifier des arguments nommés, où la correspondance avec le paramètre se fait par le nom.

```
1 >>> def f(a, b):
2 ...     print(a, b)
3 ...
4 >>> f(a=1, b=2)
5 1 2
```

Dans ce contexte, il est possible d'inverser l'ordre des paramètres (puisque'il n'importe pas pour les identifier).

```
1 >>> f(b=2, a=1)
2 1 2
```

Il est possible de passer à la fois des arguments positionnels et nommés, mais les positionnels devront toujours se trouver avant (puisque c'est leur position qui détermine le paramètre).

```
1 >>> f(1, b=2)
2 1 2
3 >>> f(b=2, 1)
4 File "<stdin>", line 1
5 SyntaxError: positional argument follows keyword argument
```

Un paramètre ne peut toujours correspondre qu'à un seul argument, Python lèvera donc une erreur s'il reçoit deux arguments (un positionnel et un nommé) pour un même paramètre.

```
1 >>> f(1, 2, b=3)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: f() got multiple values for argument 'b'
```

Ou si un argument nommé est répété.

```
1 >>> f(1, b=2, b=3)
2     File "<stdin>", line 1
3 SyntaxError: keyword argument repeated
```

De la même manière, il n'est pas possible de préciser un argument positionnel pour le second paramètre sans en préciser pour le premier (puisque le premier argument positionnel est forcément destiné au premier paramètre).

Ainsi, dans l'appel suivant Python considèrera qu'il reçoit l'argument positionnel 2 pour le paramètre `a` (le premier) et donc lèvera aussi une erreur.

```
1 >>> f(2, a=1)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: f() got multiple values for argument 'a'
```

V.3. Retours de fonctions

V.3.1. Renvoyer une valeur avec return

Pour l'instant nos fonctions s'occupent d'afficher des valeurs mais ne renvoient rien (ou plutôt renvoient `None`).

```
1 def addition(a, b):  
2     print(a + b)
```

C'est-à-dire que `addition(1, 2)` est une expression qui s'évalue à `None`, malgré le texte affiché par la fonction.

```
1 >>> x = addition(1, 2)  
2 3  
3 >>> print(x)  
4 None
```

On ne peut donc rien faire de ce résultat qui a été affiché par la fonction. Afin d'extraire le résultat, il va nous falloir le renvoyer depuis notre fonction, ce qui se fait avec le mot-clé `return`.

`return` est suivi d'une expression vers laquelle sera évalué l'appel de la fonction.

```
1 def addition(a, b):  
2     return a + b
```

On remarque que maintenant, l'appel à la fonction n'affiche plus rien (il n'y a plus de `print`).

```
1 >>> x = addition(1, 2)
```

En revanche, on récupère bien le résultat calculé dans la variable `x`.

```
1 >>> print(x)  
2 3
```

`x = addition(1, 2)` est grossièrement équivalent à `x = 1 + 2`, l'expression `addition(1, 2)` valant `1 + 2`.

Étant une expression à part entière, il est possible de l'utiliser comme valeur dans d'autres expressions :

```
1 >>> addition(addition(1, 1), addition(addition(1, 1), 1))
2 5
```

V.3.2. Plusieurs return dans une fonction

Une fonction n'est pas limitée à un seul `return` et il est ainsi possible d'en avoir plusieurs pour contrôler le flux d'exécution.

L'exécution de la fonction s'arrêtera au premier `return` rencontré, renvoyant la valeur associée à l'expression de ce `return`.

On pourrait par exemple imaginer une fonction `division(a, b)` renvoyant la division de `a` par `b` et gérant le cas de la division par zéro en renvoyant zéro.

```
1 def division(a, b):
2     if b == 0:
3         return 0
4     return a / b
```

Dans les cas où `b` vaut zéro, on rentrera donc dans le bloc de la première condition et le `return` sera exécuté. On se retrouve donc à sortir de la fonction sans exécuter la suite, c'est pourquoi aucune erreur n'est ensuite levée.

```
1 >>> division(1, 2)
2 0.5
3 >>> division(2, 0)
4 0
```

Si aucun `return` n'est rencontré lors de l'exécution de la fonction, c'est la valeur `None` qui sera automatiquement renvoyée.

```
1 def secret_addition(a, b):
2     if a + b == 42:
3         return 42
```

```
1 >>> secret_addition(12, 30)
2 42
3 >>> secret_addition(12, 33)
4 >>> print(secret_addition(12, 33))
5 None
```

Pour rappel, la valeur `None` n'est par défaut pas affichée par l'interpréteur interactif, d'où l'appel à `print` pour la mettre en évidence.

V.3.3. Renvoyer plusieurs valeurs

Comme on vient de le voir, la fonction s'arrête au premier `return` rencontré. Une fonction renvoie donc toujours une et une seule valeur, celle de l'expression située derrière ce premier `return`.

Mais il existe une astuce pour faire comme si on renvoyait plusieurs valeurs en une fois : en utilisant un tuple contenant ces valeurs. C'est le cas de la fonction `divmod` de Python, renvoyant à la fois la division entière et le modulo.

```
1 >>> divmod(13, 4)
2 (3, 1)
```

On pourrait recoder cette fonction comme cela.

```
1 def divmod(a, b):
2     return (a // b, a % b)
```

Les parenthèses autour des tuples étant facultatives, il est courant de les omettre pour les `return`, ce qui donne vraiment l'impression de renvoyer plusieurs valeurs.

```
1 def divmod(a, b):
2     return a // b, a % b
```

V.3.3.1. Unpacking

Mais une construction très intéressante en Python à ce propos est l'*unpacking*, qui permet de déstructurer un tuple. Il s'agit en fait d'utiliser un tuple de variables comme membre de gauche lors d'une assignation, pour assigner les éléments du tuple de droite aux variables de gauche.

```
1 >>> (a, b) = (3, 4)
2 >>> a
3 3
4 >>> b
5 4
```

Encore une fois, les parenthèses sont facultatives, on a donc quelque chose qui ressemble à une affectation multiple.

```
1 >>> a, b = 3, 4
```

Et bien sûr, cela fonctionne avec toute expression s'évaluant comme un tuple, par exemple un appel à `divmod`.

V. Les fonctions

```
1 >>> d, m = divmod(13, 4)
2 >>> d
3 3
4 >>> m
5 1
```

Parfois, certains éléments du tuple ne nous intéressent pas lors de l'unpacking, une convention dans ces cas-là est d'utiliser la variable `_` pour affecter les résultats inintéressants.

```
1 def compute(x):
2     return x, x*2, x*3, x*4
```

```
1 >>> _, a, _, b = compute(2)
2 >>> a
3 4
4 >>> b
5 8
```

On notera que l'*unpacking* est aussi possible pour des tuples d'un seul élément.

```
1 >>> values = (42,)
2 >>> a, = values
3 >>> a
4 42
```

Enfin, une propriété amusante de la construction/déconstruction de tuples est qu'elle permet facilement d'échanger les valeurs de deux variables. En effet, il suffit de construire un tuple avec les valeurs des deux variables puis de le déconstruire vers ces deux mêmes variables en les inversant.

```
1 >>> a = 3
2 >>> b = 5
3 >>> a, b = b, a
4 >>> a
5 5
6 >>> b
7 3
```

V.4. Paramètres et types mutables

V.4.1. Rappel sur les types mutables

On a vu que certains types étaient modifiables (mutables) et d'autres non. Les types mutables que nous avons étudiés sont les listes et les dictionnaires.

Cela signifie qu'une fois ces objets instanciés, il est possible d'en modifier la valeur. Ce qui n'est pas la même chose que réassigner une variable car cela affecte toutes les références vers l'objet.

```
1 >>> a = b = {}
2 >>> a[0] = True
3 >>> a
4 {0: True}
5 >>> b
6 {0: True}
```

Cela n'est pas possible avec un nombre, une chaîne de caractères ou un tuple, qui ne peuvent pas être modifiés en tant que tels. Et la réassignation d'une variable la fait pointer vers une nouvelle valeur, sans affecter les autres références à l'ancienne valeur.

```
1 >>> a = b = 3
2 >>> a = 5
3 >>> a
4 5
5 >>> b
6 3
```

Ainsi, toute référence vers un objet mutable va permettre d'en modifier le contenu, ce sera donc le cas aussi pour ces objets passés en arguments à des fonctions. Il faudra alors être très attentif sur ceux-ci.

V.4.2. Paramètres mutables

Ça peut donc être perturbant au premier abord, puisque la modification d'un paramètre altère la valeur passée en argument.

```
1 def append_zero(values):
2     values.append(0)
3     return values
```

```
1 >>> l = [1, 2, 3]
2 >>> append_zero(l)
3 [1, 2, 3, 0]
4 >>> l
5 [1, 2, 3, 0]
```

Ce n'est pas le cas avec une redéfinition qui crée une nouvelle instance (et donc oublie la référence précédente).

```
1 def append_zero(values):
2     values = values + [0]
3     return values
```

```
1 >>> l = [1, 2, 3]
2 >>> append_zero(l)
3 [1, 2, 3, 0]
4 >>> l
5 [1, 2, 3]
```

Attention cependant à l'opérateur += des listes qui opère une modification sur la liste existante plutôt qu'une réaffectation sur une nouvelle liste.

```
1 def append_zero(values):
2     values += [0]
3     return values
```

```
1 >>> l = [1, 2, 3]
2 >>> append_zero(l)
3 [1, 2, 3, 0]
4 >>> l
5 [1, 2, 3, 0]
```

V.4.2.1. Effets de bord

Modifier ainsi les valeurs passées en paramètres provoque ce que l'on appelle des effets de bord, c'est-à-dire que l'exécution de la fonction affecte un état extérieur, elle n'est pas reproductible dans les mêmes conditions.

On dit aussi qu'elle n'est pas «pure» (contrairement à une fonction purement mathématique qui ne ferait que calculer un nouveau résultat à partir des paramètres).

Parfois, ces effets de bord sont désirables, mais ils ne le sont pas toujours. Dans les cas où on veut les éviter, on privilégiera alors des types immutables (tuples par exemple) ou l'on créera des copies des objets reçus en paramètres.

V. Les fonctions

```
1 def append_zero(values):  
2     values = list(values) # on crée une copie  
3     values.append(0)  
4     return values
```

V.5. Fonctions de tests

V.5.1. Un monde rempli de bugs

Dans un monde idéal, on écrirait le code d'un programme du premier coup et celui-ci fonctionnerait sans aucun bug. Malheureusement ce monde n'est pas le nôtre, ici les bugs sont légion.

Regardez le code suivant, qui se veut être un équivalent à la fonction `sum` de Python.

```
1 def my_sum(numbers):
2     result = numbers[0]
3     size = len(numbers) - 1
4     for i in range(1, size):
5         result += numbers[i]
```

En regardant le code rapidement on se dit que ça doit répondre au problème. Et pourtant plusieurs bugs se sont glissés dans le code de la fonction qui font qu'elle ne pourra pas renvoyer le bon résultat.

Pour vérifier ça on va tester notre fonction, c'est-à-dire l'appeler avec différents arguments et vérifier son comportement et sa valeur de retour.

On pourrait tester une fois pour toutes les cas qui nous passent par la tête, considérer la fonction comme bonne si elle valide tout et ne plus y toucher, mais c'est une technique qui risquerait de laisser passer beaucoup de bugs. En effet, un code est amené à évoluer. Et si nous touchons au code de notre fonction (ou d'une autre fonction qu'elle appellerait), il faudrait nous assurer que son comportement est toujours le bon, qu'il n'y a pas eu de régressions.

Pour cela, on préfère avoir une suite de tests que l'on ré-appliquera à chaque nouvelle modification, afin de vérifier que nous n'avons rien cassé (que le comportement est toujours celui attendu). Il faudra donc écrire les scénarios de tests les plus précis et complets possibles pour qu'ils couvrent bien tout ce que doit réaliser la fonction.

Des bugs peuvent se glisser à toutes les phases du développement, et il est donc préférable de ne pas attendre la fin du développement d'une fonctionnalité pour la tester. Tester tôt permet en effet d'éliminer plus tôt les bugs rencontrés, et de ne pas les enfouir sous d'autres couches qui les rendront plus difficilement détectables.

Certains modèles vont encore plus loin et préconisent l'écriture des tests avant même de réaliser les fonctionnalités cibles (on parle de *test-driven development*). Cela permet d'être clair sur le comportement attendu et d'avancer itérativement en écrivant les tests puis les fonctionnalités, jusqu'à ce que notre fonction remplisse tous les cas de tests attendus.

Nous allons maintenant voir comment écrire simplement nos scénarios de tests en Python.

V.5.2. Fonctions de tests

V.5.2.1. Assertions

Il existe en Python un mot-clé, `assert`, qui permet de tester une expression booléenne. Si cette expression s'évalue à `True`, il ne se passe rien.

```
1 >>> assert 1 == 1
```

Mais si l'expression s'évalue à `False`, alors une erreur de type `AssertionError` est levée.

```
1 >>> assert 1 == 2
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   AssertionError
```



Attention cependant, les assertions ne doivent avoir un rôle que lors du développement. Elles peuvent en effet être désactivées (et donc n'avoir aucun effet même si évaluées à `False`) en production (notamment si les optimisations de l'interpréteur sont activées). Aucun problème pour des tests puisqu'ils seront exécutés dans un environnement de développement ou de tests.



Vous pouvez d'ailleurs essayer en lançant un script contenant des assertions avec `python -O script.py`, celles-ci n'ont alors plus aucun effet.

V.5.2.2. Tests unitaires

Le but maintenant va être de réaliser des assertions sur des appels à notre fonction. On veut que pour une entrée donnée on obtienne le retour attendu.

Par exemple `assert my_sum([1, 2, 3]) == 6`.

Afin d'avoir quelque chose de facilement reproductible, on va placer notre assertion dans une fonction `test_my_sum` qu'il nous suffira de réexécuter pour lancer la suite de tests. On va en profiter pour ajouter quelques autres assertions.

```
1 def test_my_sum():
2     assert my_sum([1, 2, 3]) == 6
3     assert my_sum([-1, 0, 1]) == 0
4     assert my_sum([42]) == 42
```

Puis on l'exécute.

V. Les fonctions

```
1 >>> test_my_sum()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in test_my_sum
5   AssertionError
```

Voilà déjà une première erreur, sur la première assertion (*line 2*). Et en effet, si on regarde de plus près, on voit que la fonction ne renvoie rien.

```
1 >>> my_sum([1, 2, 3])
```

On corrige donc en ajoutant un `return result` en fin de fonction, et on relance les tests.

```
1 def my_sum(numbers):
2     result = numbers[0]
3     size = len(numbers) - 1
4     for i in range(1, size):
5         result += numbers[i]
6     return result
```

```
1 >>> test_my_sum()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in test_my_sum
5   AssertionError
```

Toujours une erreur sur la même assertion, quel est le soucis cette fois ? Nous verrons plus loin quelques outils d'aide au débogage, on va pour le moment regarder «manuellement».

```
1 >>> my_sum([1, 2, 3])
2 3
3 >>> my_sum([1, 2, 3, 4])
4 6
5 >>> my_sum([1, 2, 3, 4, 5])
6 10
7 >>> my_sum([11, 2, 3, 4, 5])
8 20
```

Il semble bien que c'est le dernier élément de la liste qui est ignoré.

On peut ajouter un `print(i)` dans la boucle de notre fonction pour nous en assurer.

Quel est le soucis ? On a oublié que `range(a, b)` itérail sur les entiers `i` tels que `a <= i < b` et non `a <= i <= b`. On a donc calculé `size = len(numbers) - 1` comme index du dernier élément alors qu'il aurait fallu l'index après le dernier, simplement `size = len(numbers)`. Ce genre d'erreur est très courant et porte un nom, c'est une *off-by-one error*, une erreur de décalage de 1.

V. Les fonctions

On corrige le code de notre fonction, et on relance.

```
1 def my_sum(numbers):
2     result = numbers[0]
3     size = len(numbers)
4     for i in range(1, size):
5         result += numbers[i]
6     return result
```

```
1 >>> test_my_sum()
```

Cette fois-ci, il ne se passe rien, c'est donc que toutes les assertions sont bonnes et que les tests sont validés.

Est-ce que pour autant notre fonction est bonne ? Cela dépend justement des tests.

Quand on teste, il est important d'identifier quels cas peuvent potentiellement être problématiques. Ici on a testé avec des listes de nombres entiers positifs, des négatifs, zéro, c'est très bien. Mais on n'a pas testé de nombres flottants, on n'a pas testé de tuples. On n'a pas non plus testé le cas d'une liste vide.

Pour ne pas trop surcharger notre fonction `test_my_sum` de cas en tous genres, on va la découper en plusieurs petites fonctions pour séparer les cas bien précis. Il sera ainsi plus facile d'identifier quel genre de problème fait buguer notre fonction.

Par commodité on gardera pour le moment une fonction `test_my_sum` générale pour appeler toutes nos autres fonctions et avoir ainsi un unique point d'entrée. On verra par la suite qu'il est possible d'avoir beaucoup mieux avec les bons outils de tests.

```
1 def test_my_sum_int():
2     assert my_sum([1, 2, 3]) == 6
3     assert my_sum([-1, 0, 1]) == 0
4     assert my_sum([42]) == 42
5
6 def test_my_sum_float():
7     assert my_sum([1.0, 2.0, 3.0]) == 6.0
8     assert my_sum([0.1, 0.2, 0.3]) == 0.6
9
10 def test_my_sum_tuple():
11     assert my_sum((1, 2, 3)) == 6
12
13 def test_my_sum_empty():
14     assert my_sum([]) == 0
15     assert my_sum(()) == 0
16
17 def test_my_sum():
18     test_my_sum_int()
19     test_my_sum_float()
20     test_my_sum_tuple()
21     test_my_sum_empty()
```


V. Les fonctions

C'est l'heure du test !

```
1 >>> test_my_sum()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 3, in test_my_sum
5   File "<stdin>", line 3, in test_my_sum_float
6 AssertionError
```

Ah, la deuxième assertion (*line 3*) des tests sur les flottants ne fonctionne pas.

```
1 >>> my_sum([0.1, 0.2, 0.3])
2 0.60000000000000001
```

Et oui, souvenez-vous, l'arithmétique sur les flottants n'est pas la même chose que l'arithmétique sur les nombres décimaux. Ici c'est notre test qui est faux, il s'attend à obtenir `0.6` alors que `0.1 + 0.2 + 0.3 == 0.60000000000000001`.

Il existe des fonctions pour tester l'égalité entre flottants avec un seuil de tolérance, nous découvrirons ça dans un prochain chapitre. Pour le moment, on va simplement comparer notre résultat avec celui d'une addition entre flottants.

```
1 def test_my_sum_float():
2     assert my_sum([1.0, 2.0, 3.0]) == 1.0 + 2.0 + 3.0
3     assert my_sum([0.1, 0.2, 0.3]) == 0.1 + 0.2 + 0.3
```

Mais c'est aussi quelque chose à quoi il faudra faire attention, les problèmes peuvent aussi bien se situer dans la fonction à tester que dans les tests eux-mêmes.

On relance une nouvelle fois nos tests.

```
1 >>> test_my_sum()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 5, in test_my_sum
5   File "<stdin>", line 2, in test_my_sum_empty
6   File "<stdin>", line 2, in my_sum
7 IndexError: list index out of range
```

Maintenant, c'est le test sur la liste vide qui plante. Mais on n'obtient pas une `AssertionError`, c'est une `IndexError` qui est levée. Parce que ce n'est pas l'assertion qui a échoué, une erreur est survenue avant.

Si on regarde à la ligne indiquée dans la fonction `my_sum`, on voit `result = numbers[0]`. En effet, sur une liste vide il n'y a pas de premier élément (index 0), d'où l'erreur `IndexError`. Comme correction, on pourrait apporter une pré-condition en début de fonction pour traiter explicitement le cas de la liste vide en renvoyant directement zéro. Ainsi, la suite de la fonction ne serait pas exécutée et on éviterait de rencontrer l'erreur.

```
1 def my_sum(numbers):
2     if not numbers: # une liste vide s'évalue à False
3         return 0
4     result = numbers[0]
5     size = len(numbers)
6     for i in range(1, size):
7         result += numbers[i]
8     return result
```

Et maintenant, ça marche.

```
1 >>> test_my_sum()
```

Cette fois-ci, nous couvrons l'ensemble des cas que nous souhaitons vérifier. Nous ne testons pas la fonction sur une chaîne de caractères ou d'autres types incohérents car nous savons qu'elle n'est pas prévue pour fonctionner dans ces conditions.

i

Bien sûr, la fonction `my_sum` est inutilement compliquée, elle n'était là que dans un but d'exercice pour montrer comment apparaissaient les erreurs et quelles stratégies on pouvait adopter pour les corriger. En voici une autre version bien plus lisible et elle aussi dépourvue de bugs.

```
1 def my_sum(numbers):
2     result = 0
3     for number in numbers:
4         result += number
5     return result
```

```
1 >>> test_my_sum()
```

V.6. TP : Intégrons des fonctions à notre application

V.6.1. Découpage en fonctions

On le sait, et je le répète depuis le début : le code actuel de notre TP est très répétitif. Le but ici va donc être de le factoriser pour enfin gagner en lisibilité.

Pour cela on va se donner les différents objectifs suivants :

- Diviser et grouper les différentes commandes de saisie et de validation en fonctions.
- Ajouter une fonction pour initialiser un nouveau joueur.
- Ajouter une fonction pour réaliser un tour de jeu (joueur courant contre adversaire).
- Ajouter une fonction dédiée à l'application d'une attaque.
- Ajouter une fonction pour identifier le gagnant.
- Paramétrer ces fonctions selon les besoins.

Cette liste d'objectifs est bien sûr donnée à titre indicatif, n'hésitez pas à ajouter d'autres fonctions si vous les trouvez utiles.

V.6.1.1. Solution

Voici la solution que je propose pour ce TP. Elle repose sur plusieurs fonctions, notamment `get_choice_input(choices, error_message)` qui permet de demander une saisie à l'utilisateur et de la vérifier en fonction des choix prévus, et `game_turn(player, opponent)` qui exécute un tour de jeu (sélection et application d'une attaque).

👁 Contenu masqué n°10

V.6.2. Tests

On va maintenant ajouter quelques tests à notre jeu, pour vérifier le bon comportement de certaines fonctions.

Malheureusement, beaucoup de nos fonctions attendent des saisies de l'utilisateur, et nous ne sommes pas en mesure de les tester automatiquement pour le moment.

Nos tests vont donc se résumer aux fonctions qui n'interagissent pas avec l'utilisateur : dans ma solution il s'agit des fonctions `apply_attack` et `get_winner`. Pour les autres fonctions, il faudra pour l'instant se contenter de tests manuels en exécutant le code du TP.

Pour la fonction `apply_attack`, nous voulons nous assurer que les dégâts correspondant à l'attaque sont bien soustraits aux points de vie du joueur adverse. Nous souhaitons aussi vérifier que les points de vie ne descendent jamais en dessous de zéro.

Pour ce qui est de `get_winner`, on cherche à contrôler que c'est le joueur avec le plus de points de vie qui est identifié comme gagnant. Le cas de l'égalité entre joueurs ne nous intéresse pas

vraiment, car il ne peut pas se produire en jeu, mais on peut toujours le vérifier pour s'assurer que la fonction est cohérente (renvoie toujours le deuxième joueur par exemple).

V.6.2.1. Solution

Voilà donc les deux fonctions de tests que l'on peut ajouter et exécuter dans notre TP pour vérifier le comportement de nos fonctions.

👁 Contenu masqué n°11

Contenu masqué

Contenu masqué n°10

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }
22
23
24 def get_choice_input(choices, error_message):
25     entry = input('> ').lower()
26     while entry not in choices:
27         print(error_message)
28         entry = input('> ').lower()
29     return choices[entry]
30
31
32 def get_player(player_id):
```

```

33     print('Joueur', player_id, 'quel monstre choisissez-vous ?')
34     monster = get_choice_input(monsters, 'Monstre invalide')
35     pv = int(input('Quel est son nombre de PV ? '))
36     return {'id': player_id, 'monster': monster, 'pv': pv}
37
38
39 def get_players():
40     print('Monstres disponibles :')
41     for monster in monsters.values():
42         print('-', monster['name'])
43     return get_player(1), get_player(2)
44
45
46 def apply_attack(attack, opponent):
47     opponent['pv'] -= attack['damages']
48     if opponent['pv'] < 0:
49         opponent['pv'] = 0
50
51
52 def game_turn(player, opponent):
53     # Si le joueur est KO, il n'attaque pas
54     if player['pv'] <= 0:
55         return
56
57     print('Joueur', player['id'], 'quelle attaque utilisez-vous ?')
58     for name in player['monster']['attacks']:
59         print('-', name.capitalize(), -attacks[name]['damages'],
60               'PV')
61
62     attack = get_choice_input(attacks, 'Attaque invalide')
63     apply_attack(attack, opponent)
64
65     print(
66         player['monster']['name'],
67         'attaque',
68         opponent['monster']['name'],
69         'qui perd',
70         attack['damages'],
71         'PV, il lui en reste',
72         opponent['pv'],
73     )
74
75 def get_winner(player1, player2):
76     if player1['pv'] > player2['pv']:
77         return player1
78     else:
79         return player2
80
81

```

```
82 player1, player2 = get_players()
83
84 print()
85 print(player1['monster']['name'], 'affronte',
86        player2['monster']['name'])
87
88 while player1['pv'] > 0 and player2['pv'] > 0:
89     game_turn(player1, player2)
90     game_turn(player2, player1)
91
92 winner = get_winner(player1, player2)
93 print('Le joueur', winner['id'], 'remporte le combat avec',
94       winner['monster']['name'])
```

[Retourner au texte.](#)

Contenu masqué n°11

```
1 def test_apply_attack():
2     player = {'id': 0, 'monster': monsters['pythachu'], 'pv': 100}
3
4     apply_attack(attacks['brûlure'], player)
5     assert player['pv'] == 60
6
7     apply_attack(attacks['tonnerre'], player)
8     assert player['pv'] == 10
9
10    apply_attack(attacks['charge'], player)
11    assert player['pv'] == 0
12
13
14 def test_get_winner():
15     player1 = {'id': 0, 'monster': monsters['pythachu'], 'pv': 100}
16     player2 = {'id': 0, 'monster': monsters['pythard'], 'pv': 0}
17     assert get_winner(player1, player2) == player1
18     assert get_winner(player2, player1) == player1
19
20     player2['pv'] = 120
21     assert get_winner(player1, player2) == player2
22     assert get_winner(player2, player1) == player2
23
24     player1['pv'] = player2['pv'] = 0
25     assert get_winner(player1, player2) == player2
26     assert get_winner(player2, player1) == player1
```

[Retourner au texte.](#)

Sixième partie

Entrées / sorties

Introduction

On a maintenant les outils pour faire un programme, mais un programme qui ne sait pas faire grand chose. Niveau interactions, on se limite aux fonctions `input` et `print`, qui sont assez limitées.

Dans cette partie nous allons voir comment gérer de façon plus poussée les événements extérieurs.

VI.1. Découper son code en modules

VI.1.1. Factoriser le code

À force de factoriser notre code, on peut facilement se retrouver avec un script Python contenant de nombreuses fonctions. Des fonctions qui ne sont pas toujours liées les unes aux autres car agissent sur des concepts différents.

Pour aller plus loin dans la factorisation, il faudrait alors regrouper nos fonctions selon les liens qu'elles entretiennent, pour former des unités logiques. Par exemple les fonctions liées à l'affichage d'un côté et celles concernant les calculs de l'autre.

Ces unités logiques portent un nom en Python, on les appelle des modules.

VI.1.2. Les modules

Les modules forment un espace de noms et permettent ainsi de regrouper les définitions de fonctions et variables, en les liant à une même entité.

Ils prennent la forme de fichiers Python (un nom et une extension `.py`) et doivent suivre une nomenclature particulière (la même que pour les noms de variables ou de fonction) : uniquement composés de lettres, de chiffres et d'*underscores* (`_`), et ne commençant pas par un chiffre.

Ainsi, un fichier `foo.py` correspondra à un module `foo`.

```
1 def addition(a, b):  
2     return a + b  
3  
4 def soustraction(a, b):  
5     return a - b
```

Listing 17 – `foo.py`

Pour charger le code d'un module (le code du fichier associé) et avoir accès à ses définitions, il est nécessaire de l'importer. On utilise pour cela le mot-clé `import` de Python suivi du nom du module (`foo` dans notre exemple).

```
1 >>> import foo
```

Rien ne se passe. En fait, le code de notre fichier `foo.py` a bien été exécuté, mais comme il ne fait que définir des fonctions c'est invisible pour nous.

Avec le fichier `bar.py` suivant :

```
1 print('Je suis le module bar')
```

Listing 18 – bar.py

On constate bien que son code est exécuté à l'import.

```
1 >>> import bar
2 Je suis le module bar
```

Mais revenons-en à notre premier module, `foo`. C'est bien beau de l'avoir importé, mais on aimerait pouvoir en exécuter les fonctions. Si vous avez tenté d'appeler `addition` ou `soustraction` vous avez remarqué que les fonctions n'existaient pas et obtenu une erreur `NameError`. C'est parce que les fonctions existent mais dans l'espace de noms (*namespace*) du module `foo`. Il faut alors les préfixer de `foo.` pour y accéder : `foo.addition` ou `foo.soustraction`. L'opérateur `.` signifiant «accède au nom contenu dans».

```
1 >>> foo.addition(3, 5)
2 8
3 >>> foo.soustraction(12, 42)
4 -30
```

`foo` en lui-même est un objet d'un nouveau type représentant le module importé.

```
1 >>> foo
2 <module 'foo' from '/.../foo.py'>
```

VI.1.2.1. La fonction `help`

Une fonction de Python est très utile pour se documenter sur un module, il s'agit de la fonction `help`. On appelle la fonction depuis l'interpréteur interactif en lui donnant l'objet-module en argument (il faut donc l'avoir importé au préalable).

```
1 >>> help(foo)
```

Le terminal affiche alors un écran d'aide détaillant le contenu du module. Appuyez sur la touche **Q** pour quitter cet écran et revenir à l'interpréteur.

```
1 Help on module foo:
2
3 NAME
4     foo
5
6 FUNCTIONS
7     addition(a, b)
8
9     soustraction(a, b)
10
```

```
11 FILE
12     /.../foo.py
13
14 (END)
```

C'est très succinct pour le moment, nous verrons par la suite comment étayer tout cela en ajoutant de la documentation à notre module.

Notez que la fonction `help` n'est pas utile uniquement pour les modules, elle permet aussi de se documenter sur une fonction ou un type.

```
1 >>> help(abs)
2
3 >>> help(int)
```

Vous pouvez faire défiler l'écran à l'aide des flèches haut/bas ou page-up/page-down, ainsi que la touche espace pour naviguer de page en page.

La fonction s'utilise aussi avec une chaîne de caractères en argument pour rechercher de l'aide sur un sujet précis. Par exemple `help('keywords')` pour obtenir la liste des mots-clés de Python.

Enfin, on peut utiliser la fonction sans argument pour entrer dans une interface d'aide où chaque ligne entrée sera exécutée comme un nouvel appel à la fonction `help`.

```
1 >>> help()
2
3 Welcome to Python 3.9's help utility!
4
5 [...]
6
7 help>
```



Actuellement nos modules ne sont accessibles que si les fichiers Python sont disposés dans le même répertoire. Nous verrons dans un prochain chapitre comment permettre une arborescence plus complexe à l'aide des paquets.

VI.1.3. Imports

Il y a différentes manières d'importer un module, et nous allons ici voir en quoi elles consistent.

Déjà, on a vu le simple `import foo` qui crée un nouvel objet `foo` représentant notre module et donc contenant les fonctions du module `foo.addition` et `foo.soustraction`.

Il est possible lors de l'import de choisir un autre nom que `foo` pour l'objet créé (par exemple pour opter pour un nom plus court) à l'aide du mot-clé `as` suivi du nom souhaité.

```

1 >>> import foo as oof
2 >>> foo
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 NameError: name 'foo' is not defined
6 >>> oof
7 <module 'foo' from '/tmp/foo.py'>
8 >>> oof.addition(1, 2)
9 3

```

Une autre syntaxe permet d'importer directement les objets que l'on veut depuis le module, sans créer d'objet pour le module, il s'agit de `from ... import ...`.

```

1 >>> from foo import addition
2 >>> addition
3 <function addition at 0x7feb439c34c0>
4 >>> addition(3, 5)
5 8
6 >>> soustraction
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 NameError: name 'soustraction' is not defined
10 >>> foo
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 NameError: name 'foo' is not defined

```

Comme on le voit, cette syntaxe permet de rendre accessible la fonction `addition` directement, et uniquement elle.

Il est aussi possible de préciser plusieurs objets à importer en les séparant par des virgules.

```

1 >>> from foo import addition, soustraction
2 >>> addition(3, 5)
3 8
4 >>> soustraction(8, 2)
5 6

```

Enfin, il peut arriver que vous rencontriez des `from foo import *`. Cela permet d'importer tous les noms présents dans le module `foo` (ici `addition` et `soustraction`) et de les rendre directement accessibles comme s'ils étaient tous précisés explicitement.

C'est une syntaxe pratique pour des tests rapides dans l'interpréteur mais qui est peu recommandable généralement, parce qu'elle pollue inutilement l'espace de noms courant avec tout le contenu du module (et peut effacer des objets s'il y a un conflit entre les noms). Comme on dit, l'explicite est préférable à l'implicite.

VI.1.4. Bibliothèque standard

Python dispose par défaut de nombreux modules déjà prêts à être utilisés. Ils sont regroupés dans ce qu'on appelle la bibliothèque standard (ou *stdlib* pour *standard library*), c'est-à-dire les modules disponibles directement après l'installation de Python.

Ces modules apportent des fonctions concernant des domaines particuliers qui ne sont pas incluses dans l'espace de noms global pour ne pas le surcharger. Ainsi on a par exemple un module `math` pour toutes les fonctions mathématiques usuelles (`sqrt`, `exp`, `cos`, `sin`) ainsi que les constantes (`pi`, `e`).

On importe donc le module comme on le faisait précédemment.

```
1 >>> import math
```

Le module a beau ne pas se trouver dans le répertoire d'exécution, Python arrive à le trouver car il se situe dans un des répertoires d'installation.



Attention d'ailleurs à la priorité des répertoires lors de la recherche d'un module : si nous avons un fichier `math.py` dans le répertoire d'exécution, c'est lui qui serait importé lors d'un `import math` plutôt que celui de la bibliothèque standard. Veillez donc toujours à ne pas utiliser de nom existant pour vos propres modules.

Comme annoncé, nous retrouvons dans ce module différentes constantes mathématiques. Il s'agit de nombres flottants, avec donc la précision qui est la leur.

```
1 >>> math.pi
2 3.141592653589793
3 >>> math.e
4 2.718281828459045
5 >>> math.inf
6 inf
```

Cette dernière représente l'infini, un nombre flottant supérieur à tout autre.

Question fonctions il ne sera pas possible de tout énumérer mais en voici quelques exemples.

```
1 >>> math.sqrt(2) # Racine carrée
2 1.4142135623730951
3 >>> math.floor(1.5) # Arrondi à l'inférieur
4 1
5 >>> math.ceil(1.5) # Arrondi au supérieur
6 2
7 >>> math.cos(math.pi) # Cosinus (argument en radians)
8 -1.0
9 >>> math.sin(0) # Sinus (argument en radians)
10 0.0
11 >>> math.radians(180) # Conversion degrés -> radians
12 3.141592653589793
```

```

13 >>> math.degrees(math.pi) # Conversion radians -> degrés
14 180.0
15 >>> math.exp(1) # Exponentielle
16 2.718281828459045
17 >>> math.log(math.e) # Logarithme
18 1.0
19 >>> math.gcd(12, 8) # Calcul de PGCD
20 4

```

Encore une fois, pensez à `help(math)` si vous voulez un aperçu complet, ou à consulter [la documentation](#) .

Je voudrais enfin attirer votre attention sur la fonction `isclose`. Cette fonction permet de comparer deux nombres flottants avec une certaine marge d'erreur.

Pour rappel, il y a une certaine imprécision dans le stockage des flottants, et l'opérateur `==` est donc déconseillé. `isclose` prend simplement les deux nombres en paramètres et renvoie un booléen indiquant s'ils sont «égaux» (disons très proches) ou non.

```

1 >>> 0.1 + 0.2 == 0.3
2 False
3 >>> math.isclose(0.1 + 0.2, 0.3)
4 True
5 >>> math.isclose(0.2, 0.3)
6 False

```

VI.1.5. Modules de tests

Revenons-en à notre dernier TP. Il serait intéressant dans un premier temps de séparer les tests du reste du code. Ils n'ont en effet pas de raison particulière d'être placés là.

Dans un fichiers `tests.py`, on va donc placer toutes les fonctions `test_*`. Mais ce module `tests` n'aura par défaut pas accès aux fonctions à tester, il va donc nous falloir les importer. Au début du module `tests`, on placera donc les lignes d'import suivantes.

```

1 from game import ...
2 from game import ...

```

Aussi, vous vous souvenez de notre fonction pour réunir tous les tests ? Elle n'a maintenant plus lieu d'être, étant donné que nous sommes dans un module à part nous savons que son code ne sera pas exécuté par erreur.

On peut donc placer les appels des fonctions de tests à la toute fin de notre module. Enfin pas tout à fait, on va inclure nos appels dans un bloc conditionnel `if __name__ == '__main__':`.

```

1 if __name__ == '__main__':
2     test_...()
3     test_...()

```

Cette ligne obscure permet de savoir si on exécute directement le module ou si on l'importe. En effet, la variable spéciale `__name__` contient le nom du module. Dans le cas où le module est exécuté directement par Python (`python tests.py`), ce nom vaudra `'__main__'` (il s'agira sinon de `'tests'` lors d'un import).

Par cette ligne, nous nous assurons donc que les fonctions de tests ne seront pas exécutées lors d'un import. Ce n'est pas très important pour un module de tests qui n'a pas vocation à être importé, mais ça reste un outil pratique pour qu'un script soit importable. C'est donc toujours une bonne habitude à prendre.

```
1 print('A')
2
3 if __name__ == '__main__':
4     print('B')
```

Listing 19 – foo.by

```
1 $ python foo.by
2 A
3 B
```

```
1 >>> import foo
2 A
```

Nous allons d'ailleurs aussi modifier le code de notre TP pour ajouter une telle condition `if __name__ == '__main__':` et y placer le code qui ne figure dans aucune fonction. Ça nous évitera d'avoir le code du jeu qui s'exécute lors d'un `import game` depuis les tests.

© Contenu masqué n°12

Contenu masqué

Contenu masqué n°12

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
}
```

```

10     'ponythia': {
11         'name': 'Ponytha',
12         'attacks': ['brûlure', 'charge'],
13     },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }
22
23
24 def get_choice_input(choices, error_message):
25     entry = input('> ').lower()
26     while entry not in choices:
27         print(error_message)
28         entry = input('> ').lower()
29     return choices[entry]
30
31
32 def get_player(player_id):
33     print('Joueur', player_id, 'quel monstre choisissez-vous ?')
34     monster = get_choice_input(monsters, 'Monstre invalide')
35     pv = int(input('Quel est son nombre de PV ? '))
36     return {'id': player_id, 'monster': monster, 'pv': pv}
37
38
39 def get_players():
40     print('Monstres disponibles :')
41     for monster in monsters.values():
42         print('-', monster['name'])
43     return get_player(1), get_player(2)
44
45
46 def apply_attack(attack, opponent):
47     opponent['pv'] -= attack['damages']
48     if opponent['pv'] < 0:
49         opponent['pv'] = 0
50
51
52 def game_turn(player, opponent):
53     # Si le joueur est KO, il n'attaque pas
54     if player['pv'] <= 0:
55         return
56
57     print('Joueur', player['id'], 'quelle attaque utilisez-vous ?')
58     for name in player['monster']['attacks']:

```



```

59         print('-', name.capitalize(), -attacks[name]['damages'],
60               'PV')
61
62     attack = get_choice_input(attacks, 'Attaque invalide')
63     apply_attack(attack, opponent)
64
65     print(
66         player['monster']['name'],
67         'attaque',
68         opponent['monster']['name'],
69         'qui perd',
70         attack['damages'],
71         'PV, il lui en reste',
72         opponent['pv'],
73     )
74
75 def get_winner(player1, player2):
76     if player1['pv'] > player2['pv']:
77         return player1
78     else:
79         return player2
80
81
82 if __name__ == '__main__':
83     player1, player2 = get_players()
84
85     print()
86     print(player1['monster']['name'], 'affronte',
87           player2['monster']['name'])
88     print()
89
90     while player1['pv'] > 0 and player2['pv'] > 0:
91         game_turn(player1, player2)
92         game_turn(player2, player1)
93
94     winner = get_winner(player1, player2)
95     print('Le joueur', winner['id'], 'remporte le combat avec',
96           winner['monster']['name'])

```

[Retourner au texte.](#)

VI.2. Lire un fichier en Python

Introduction

Avec les modules nous savons déjà lire les fichiers Python, mais seulement eux et pour un traitement bien particulier.

Ici, nous voulons plutôt apprendre à gérer les fichiers présents sur l'ordinateur, comme des documents textes.

VI.2.1. Fichiers et dossiers sur l'ordinateur

Pour rappel, un ordinateur organise ses données en fichiers. Il existe des fichiers de tous types : des images, des fichiers de code, des musiques, etc. Un fichier représente un document bien précis sur l'ordinateur.

Chaque fichier se situe dans un dossier (ou répertoire). On peut voir les dossiers comme des classeurs où seraient rangés les fichiers.

Ces dossiers forment une structure hiérarchique sur l'ordinateur : un dossier peut contenir d'autres dossiers, comme des intercalaires dans un classeur, ou des classeurs sur une étagère.

Un fichier appartient alors à un dossier, qui lui-même appartient à un dossier parent, etc. jusqu'à atteindre la racine du système de fichiers.

Pour retrouver un fichier, il est alors courant d'utiliser son chemin. Il s'agit de la hiérarchie de dossiers à parcourir puis du nom du fichier en question. Ce chemin est unique. C'est ce chemin que nous utiliserons dans nos programmes pour accéder aux fichiers.

Sous Windows, un chemin sera généralement de la forme `C:\chemin\vers\mon\fichier.txt` où `C:\` représente la racine du système de fichiers.

Sous Linux on verra plutôt `/chemin/vers/mon/fichier.txt` (où `/` est la racine).

On dit que ce chemin est le chemin absolu vers le fichier, car il débute par la racine du système, qui permet donc de le retrouver depuis n'importe où.

Mais il est aussi possible de préciser le chemin d'un fichier à partir d'un autre répertoire, on parle alors de chemin relatif.

Par exemple, depuis le répertoire `C:\chemin\vers` (ou `/chemin/vers`), le chemin relatif de notre fichier est `mon\fichier.txt` (ou `mon/fichier.txt`). Il s'agit du chemin restant à parcourir pour trouver le fichier.

En programmation, nous exécuterons toujours notre code depuis un répertoire particulier, que l'on appellera répertoire courant (généralement le dossier dans lequel sont stockés les fichiers de code). Nous pourrions ainsi référencer nos fichiers par leur chemin absolu, ou par leur chemin relatif par rapport à ce répertoire.



Il est aussi possible dans un chemin relatif d'accéder à un fichier d'un répertoire parent, à l'aide de la syntaxe `..`.

Par exemple depuis le répertoire `C:\chemin\vers\toto` (`/chemin/vers/toto`), on



peut accéder à notre fichier `fichier.txt` via le chemin relatif `..\mon\fichier.txt` (`../mon/fichier.txt`).

VI.2.2. Problématique : sauvegarder l'état de notre jeu

Avec notre jeu, nous sommes pour l'instant obligé de faire toute la partie en une fois. Bon, il est assez simpliste et ne consiste que dans un combat.

Mais imaginons que nous le développons pour avoir un système de tournoi, ou développer un RPG autour, alors il serait pratique de pouvoir mettre le jeu en pause. Pour cela, il va falloir d'une manière ou d'une autre enregistrer l'état actuel du jeu afin de le reprendre plus tard.

Et la manière la plus simple de procéder, c'est d'utiliser un fichier : l'état du jeu sera sauvegardé dans le fichier à la fermeture, et rechargé depuis le même fichier au lancement. Nous allons donc dans un premier temps voir comment nous pouvons gérer nos fichiers, et dans un second nous nous intéresserons au format des données.

VI.2.3. Fonction open

Nous allons commencer simplement avec un fichier texte. Commencez par créer un fichier `hello.txt` dans votre répertoire courant, contenant simplement la phrase `Hello World!`. Vous pouvez utiliser votre éditeur de code pour créer ce fichier.



Sous Windows, l'extension des fichiers n'est pas affichée par défaut. Assurez-vous donc que votre fichier se nomme bien `hello.txt` (extension comprise) pour que la suite puisse fonctionner correctement.

Depuis Python, nous utiliserons ensuite la fonction `open` pour ouvrir le fichier, avec comme argument le chemin vers notre fichier. Ici, comme notre fichier se trouve dans le répertoire courant, il nous suffira de faire `open('hello.txt')`.

Pour un fichier dans le répertoire parent, nous aurions par exemple écrit `open('../hello.txt')`, ou `open('subdirectory/hello.txt')` pour un répertoire enfant (`open('..\hello.txt')` ou `open('subdirectory\hello.txt')` sous Windows).

```
1 >>> open('hello.txt')
2 <_io.TextIOWrapper name='hello.txt' mode='r' encoding='UTF-8'>
```

On voit que l'appel nous renvoie un objet un peu étrange mais l'essentiel est là : nous avons ouvert un fichier `hello.txt` encodé en `UTF-8` et en mode `r`.

Qu'est-ce que ce mode ?

Il faut savoir que plusieurs opérations sont possibles pour les fichiers, de lecture et d'écriture. Les différentes opérations impliquent des besoins différents et le système d'exploitation requiert donc un mode lors de l'ouverture du fichier.

Ici, `r` signifie que nous ouvrons le fichier en lecture seule (*read*), nous verrons par la suite quels autres modes d'ouverture existent.

La fonction `open` prend un deuxième argument optionnel pour spécifier ce mode. Il vaut `'r'` par défaut, d'où le comportement que nous observons.

```

1 >>> open('hello.txt', 'r')
2 <_io.TextIOWrapper name='hello.txt' mode='r' encoding='UTF-8'>

```

Ça c'est pour les cas où ça se passe bien. Il se peut aussi que l'ouverture échoue : si le fichier est introuvable ou que les droits sont insuffisants par exemple (pas la permission d'accéder au fichier appartenant à un autre utilisateur). Dans ce cas, une erreur sera levée par la fonction `open` et le fichier ne sera pas ouvert.

```

1 >>> open('notfound.txt')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 FileNotFoundError: [Errno 2] No such file or directory:
   'notfound.txt'
5 >>> open('cantread.txt')
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 PermissionError: [Errno 13] Permission denied: 'cantread.txt'

```

Dans le cas où vous rencontriez ces erreurs pour un fichier qui devrait être bon, assurez-vous donc toujours que vous êtes dans le bon répertoire et que l'utilisateur a les droits suffisants pour lire le fichier.

VI.2.4. Fichiers

VI.2.4.1. Lire le contenu d'un fichier

Avoir ouvert un fichier, c'est bien, mais ce qui nous intéresse ici est son contenu. Nous allons pour cela nous intéresser à l'objet renvoyé par `open`.

Il s'agit d'un objet de type `TextIOWrapper`, c'est ainsi que Python identifie un fichier textuel. Cet objet possède différentes méthodes, et notamment la méthode `read`. Utilisée sans argument, elle renvoie le contenu complet du fichier sous forme d'une chaîne de caractères.

```

1 >>> f = open('hello.txt')
2 >>> f.read()
3 'Hello World!\n'

```

On remarque ici que mon fichier se termine par un saut de ligne, cela fait partie du contenu du fichier.



Sous Windows, il est possible que votre fichier se termine par `\r\n`, qui est la représentation d'un passage à la ligne sur ce système.

Mais l'objet que nous avons en Python n'est pas à proprement parler un fichier, c'est une entité qui enrobe les opérations possibles sur le fichier, on parle de *wrapper*. Et celui-ci ne représente qu'un curseur qui avance dans le fichier présent sur le système. Ainsi, l'état d'un fichier évolue au fur et à mesure qu'on le parcourt.

À l'ouverture, le curseur se trouvait naturellement au début du fichier. Mais une fois le contenu lu, celui-ci s'est déplacé—comme sur une bande d'enregistrement qui défilerait—et se trouve maintenant à la fin. Ne vous étonnez donc pas si vous tentez un nouveau `read` sur le même fichier et obtenez une chaîne vide.

```
1 >>> f.read()
2 ''
```

L'explication est que la fonction lit le contenu à partir de là où se trouve le curseur dans le fichier, et en l'occurrence il n'y a plus rien à lire.

Une seule lecture suffit généralement à traiter le contenu du fichier, mais il peut arriver dans certains cas que l'on veuille revenir en arrière. Il existe pour cela la méthode `seek` prenant une position dans le fichier pour y déplacer le curseur. `0` correspond au début du fichier.

```
1 >>> f.seek(0)
2 0
3 >>> f.read()
4 'Hello World!\n'
```

Mais une autre position dans le fichier serait aussi valide.

```
1 >>> f.seek(6)
2 6
3 >>> f.read()
4 'World!\n'
```

VI.2.4.2. Fermer un fichier

Un tel curseur sur un fichier représente une ressource au niveau du système d'exploitation, et les ressources sont limitées. Le nombre de fichiers qu'un programme peut ouvrir va dépendre de la machine et du système, il est par exemple de 1024 chez moi. C'est-à-dire que chaque programme ne peut ouvrir plus de 1024 fichiers simultanément.

Vous me direz que nous en sommes encore loin mais toujours est-il qu'il n'est pas utile de gaspiller ces ressources. Ainsi, nous prendrons l'habitude de libérer notre ressource dès que nous aurons terminé de travailler avec elle.

Cela se fait par exemple avec un appel à la méthode `close` sur le fichier.

```
1 >>> f.close()
```

La méthode ne renvoie rien, tout s'est bien passé, la ressource est maintenant libérée sur le système.

Si nous essayons à nouveau de réaliser une opération sur notre fichier (`read`, `seek`), nous obtiendrons une erreur comme quoi le fichier est fermé. Python n'a en effet plus de référence vers le fichier et il faudrait l'ouvrir à nouveau (avec un appel à `open`) pour retravailler dessus.

```

1 >>> f.read()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ValueError: I/O operation on closed file.

```

VI.2.4.3. Bloc `with`

Néanmoins, l'appel explicite à `close` n'est pas la manière à privilégier pour libérer la ressource. Prenons par exemple la fonction suivante, pour récupérer le contenu d'un fichier sous forme d'un nombre entier (`int`).

```

1 def get_file_number(filename):
2     f = open(filename)
3     content = f.read()
4     value = int(content)
5     f.close()
6     return value

```

À l'usage, sur un fichier `number.txt` contenant le texte `42`, elle fonctionne très bien.

```

1 >>> get_file_number('number.txt')
2 42

```

Mais si on tente de l'exécuter avec notre fichier `hello.txt` (qui ne contient pas un nombre) on obtient logiquement une erreur.

```

1 >>> get_file_number('hello.txt')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 4, in get_file_number
5   ValueError: invalid literal for int() with base 10: 'Hello
   World!\n'

```

L'erreur survient à la ligne 4 de notre fonction, `value = int(content)`. À cet instant, l'exécution de la fonction s'arrête pour remonter l'erreur survenue.

La ligne suivante, `f.close()` n'a donc pas pu être exécutée, et ne le sera pas. C'est tout de même problématique.

Il y a des mécanismes pour traiter les erreurs et gérer des cas comme celui-ci (voir chapitres suivants), mais le plus simple est encore de ne pas avoir à faire l'appel à `close` nous-même.

Pour cela il existe en Python ce qu'on appelle des gestionnaires de contexte qui permettent de facilement traiter les ressources. Ils prennent la forme d'un bloc `with`, suivi par l'expression récupérant la ressource (ici l'appel à `open`). Le mot-clé `as` permet ensuite de récupérer cette ressource dans une variable.

```

1 with open('hello.txt') as f:
2     print(f.read())

```

Le code précédent est ainsi équivalent à :

```

1 f = open('hello.txt')
2 print(f.read())
3 f.close()

```

À l'exception que le `close` sera réalisé dans tous les cas, même si le `read` échoue par exemple. Le code de notre fonction `get_file_number` deviendrait donc :

```

1 def get_file_number(filename):
2     with open(filename) as f:
3         content = f.read()
4         return int(content)

```

Et on observe le même comportement que précédemment à l'utilisation.

```

1 >>> get_file_number('number.txt')
2 42
3 >>> get_file_number('hello.txt')
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "<stdin>", line 4, in get_file_number
7   ValueError: invalid literal for int() with base 10: 'Hello
   World!\n'

```

L'erreurs survient toujours, mais cette fois-ci la ressource a correctement été libérée, le mécanisme est géré par Python.



Quand vous manipulez des fichiers, utilisez donc toujours un bloc `with` pour éviter les soucis.

VI.3. Itérer sur un fichier

VI.3.1. Méthodes des fichiers

Avec `read` nous savons lire le contenu complet d'un fichier dans une chaîne de caractères. Mais ce n'est pas toujours le plus pratique et il est souvent préférable de pouvoir traiter un fichier par morceaux. En plus, ça évite de devoir stocker la totalité du fichier en mémoire si ça n'est pas nécessaire (heureusement que les lecteurs vidéo ne chargent pas tout le contenu d'un film dans une chaîne de caractères).

Pour ce chapitre, j'utiliserai le fichier `corbeau.txt` avec le contenu suivant :

👁 Contenu masqué n°13

Une première manière de découper est d'utiliser l'argument optionnel de `read` qui permet de préciser une taille. La longueur du texte renvoyé sera ainsi toujours inférieure ou égale à cette taille (inférieur s'il n'y a plus rien d'autre à lire par exemple), et le curseur avancé d'autant dans le fichier.

```
1 >>> with open('corbeau.txt') as f:
2 ...     f.read(100)
3 ...     f.read(100)
4 ...     f.read(100)
5 ...     f.read(100)
6 ...     f.read(100)
7 ...     f.read(100)
8 ...     f.read(100)
9 ...     f.read(100)
10 ...
11 "Maître Corbeau, sur un arbre perché,\nTenait en son bec un
    fromage.\nMaître Renard, par l'odeur alléché"
12 'é,\nLui tint à peu près ce langage :\nEt bonjour, Monsieur du
    Corbeau.\nQue vous êtes joli ! que vous m'
13 'e semblez beau !\nSans mentir, si votre ramage\nSe rapporte à
    votre plumage,\nVous êtes le Phénix des h'
14 'ôtes de ces bois.\nÀ ces mots, le Corbeau ne se sent pas de joie
    ;\nEt pour montrer sa belle voix,\nIl '
15 "ouvre un large bec, laisse tomber sa proie.\nLe Renard s'en
    saisit, et dit : Mon bon Monsieur,\nAppren"
16 "ez que tout flatteur\nVit aux dépens de celui qui
    l'écoute.\nCette leçon vaut bien un fromage, sans do"
17 "ute.\nLe Corbeau honteux et confus\nJura, mais un peu tard, qu'on
    ne l'y prendrait plus.\n"
```

Il serait possible, à l'aide d'une boucle, de parcourir le fichier en entier.


```

1 with open('corbeau.txt') as f:
2     chunk = f.read(100)
3     while chunk:
4         print(chunk)
5         chunk = f.read(100)

```

C'est très bien quand on souhaite découper en morceaux de taille fixe (ou tronçons, *chunks*), mais ça se prête assez mal à un fichier texte. Une lecture ligne par ligne nous serait plus utile. Et c'est le but de la méthode `readline`. Celle-ci s'occupe de repérer où sont les retours à la ligne et ainsi de ne renvoyer qu'une ligne à la fois, en gardant ce qui suit pour un prochain appel.

```

1 >>> with open('corbeau.txt') as f:
2     ...     line = f.readline()
3     ...     while line:
4     ...         line
5     ...         line = f.readline()
6     ...
7 'Maître Corbeau, sur un arbre perché,\n'
8 'Tenait en son bec un fromage.\n'
9 'Maître Renard, par l'odeur alléché,\n'
10 'Lui tint à peu près ce langage :\n'
11 'Et bonjour, Monsieur du Corbeau.\n'
12 'Que vous êtes joli ! que vous me semblez beau !\n'
13 'Sans mentir, si votre ramage\n'
14 'Se rapporte à votre plumage,\n'
15 'Vous êtes le Phénix des hôtes de ces bois.\n'
16 'À ces mots, le Corbeau ne se sent pas de joie ;\n'
17 'Et pour montrer sa belle voix,\n'
18 'Il ouvre un large bec, laisse tomber sa proie.\n'
19 'Le Renard s'en saisit, et dit : Mon bon Monsieur,\n'
20 'Apprenez que tout flatteur\n'
21 'Vit aux dépens de celui qui l'écoute.\n'
22 'Cette leçon vaut bien un fromage, sans doute.\n'
23 'Le Corbeau honteux et confus\n'
24 'Jura, mais un peu tard, qu'on ne l'y prendrait plus.\n'

```

On constate tout de même que le retour à la ligne est considéré comme faisant partie de la ligne.

On trouve aussi la méthode `readlines` pour lire toutes les lignes d'un coup et les renvoyer dans un tableau. Mais on retombe sur le problème initial : cela demande à stocker le fichier en mémoire dans sa totalité.

```

1 >>> with open('corbeau.txt') as f:
2     ...     f.readlines()
3     ...
4 ['Maître Corbeau, sur un arbre perché,\n',

```

```

5  'Tenait en son bec un fromage.\n',
6  "Maître Renard, par l'odeur alléché,\n",
7  'Lui tint à peu près ce langage :\n',
8  'Et bonjour, Monsieur du Corbeau.\n',
9  'Que vous êtes joli ! que vous me semblez beau !\n',
10 'Sans mentir, si votre ramage\n',
11 'Se rapporte à votre plumage,\n',
12 'Vous êtes le Phénix des hôtes de ces bois.\n',
13 'À ces mots, le Corbeau ne se sent pas de joie ;\n',
14 'Et pour montrer sa belle voix,\n',
15 'Il ouvre un large bec, laisse tomber sa proie.\n',
16 "Le Renard s'en saisit, et dit : Mon bon Monsieur,\n",
17 'Apprenez que tout flatteur\n',
18 "Vit aux dépens de celui qui l'écoute.\n",
19 'Cette leçon vaut bien un fromage, sans doute.\n',
20 'Le Corbeau honteux et confus\n',
21 "Jura, mais un peu tard, qu'on ne l'y prendrait plus.\n"]

```

VI.3.2. Les fichiers sont itérables

La solution avec `readlines` n'est pas satisfaisante si nous voulons traiter le fichier pas à pas, et celle avec `readline` est un peu compliquée : on constate que la boucle `while` ne se prête pas à ce problème puisqu'on est obligé de répéter l'opération `line = f.readline()`.

Mais pour rappel, les listes ne sont pas les seuls objets itérables. Outre les autres exemples que l'on a déjà vus, il est aussi possible d'itérer sur des fichiers. Et cela correspond évidemment à une itération ligne par ligne sur le fichier.

```

1  >>> with open('corbeau.txt') as f:
2  ...     for line in f:
3  ...         line
4  ...
5  'Maître Corbeau, sur un arbre perché,\n'
6  'Tenait en son bec un fromage.\n'
7  "Maître Renard, par l'odeur alléché,\n"
8  'Lui tint à peu près ce langage :\n'
9  'Et bonjour, Monsieur du Corbeau.\n'
10 'Que vous êtes joli ! que vous me semblez beau !\n'
11 'Sans mentir, si votre ramage\n'
12 'Se rapporte à votre plumage,\n'
13 'Vous êtes le Phénix des hôtes de ces bois.\n'
14 'À ces mots, le Corbeau ne se sent pas de joie ;\n'
15 'Et pour montrer sa belle voix,\n'
16 'Il ouvre un large bec, laisse tomber sa proie.\n'
17 "Le Renard s'en saisit, et dit : Mon bon Monsieur,\n"
18 'Apprenez que tout flatteur\n'
19 "Vit aux dépens de celui qui l'écoute.\n"
20 'Cette leçon vaut bien un fromage, sans doute.\n'

```

```
21 'Le Corbeau honteux et confus\n'  
22 "Jura, mais un peu tard, qu'on ne l'y prendrait plus.\n"
```

On fera difficilement plus simple que cette solution.

Partant de là, il est aussi facile de traiter notre fichier comme s'il ne s'agissait que d'un ensemble de lignes, avec une fonction comme la suivante.

```
1 def print_text(lines):  
2     i = 1 # Compteur de ligne  
3     for line in lines:  
4         line = line.rstrip('\n') # On retire le saut de ligne  
5         print(i, ': ', line)  
6         i += 1  
7  
8 with open('corbeau.txt') as f:  
9     print_text(f)
```

Cette fonction s'abstrait complètement du type réel de l'objet et fonctionnerait très bien avec une liste de chaînes de caractères en argument.

```
1 >>> print_text(['abc', 'def', 'ghi'])  
2 1 : abc  
3 2 : def  
4 3 : ghi
```

Contenu masqué

Contenu masqué n°13

```
1 Maître Corbeau, sur un arbre perché,  
2 Tenait en son bec un fromage.  
3 Maître Renard, par l'odeur alléché,  
4 Lui tint à peu près ce langage :  
5 Et bonjour, Monsieur du Corbeau.  
6 Que vous êtes joli ! que vous me semblez beau !  
7 Sans mentir, si votre ramage  
8 Se rapporte à votre plumage,  
9 Vous êtes le Phénix des hôtes de ces bois.  
10 À ces mots, le Corbeau ne se sent pas de joie ;  
11 Et pour montrer sa belle voix,  
12 Il ouvre un large bec, laisse tomber sa proie.  
13 Le Renard s'en saisit, et dit : Mon bon Monsieur,  
14 Apprenez que tout flatteur  
15 Vit aux dépens de celui qui l'écoute.
```

```
16 Cette leçon vaut bien un fromage, sans doute.  
17 Le Corbeau honteux et confus  
18 Jura, mais un peu tard, qu'on ne l'y prendrait plus.
```

Listing 20 – corbeau.txt

[Retourner au texte.](#)

VI.4. Écrire dans un fichier

VI.4.1. Écriture

Nous avons vu que la fonction `open` prenait un argument optionnel pour spécifier le mode d'ouverture du fichier, et n'avons pour le moment utilisé que le mode lecture (`'r'`). Vous vous en doutez, il va ici être question d'un nouveau mode afin de pouvoir écrire dans nos fichiers. Il n'y a pas un unique mode d'écriture, car plusieurs options sont possibles, mais nous allons commencer avec le mode `'w'` (pour *write*).

i

Dans les exemples qui suivront je n'utiliserai pas de bloc `with` pour simplifier les opérations dans l'interpréteur interactif. Il s'agit là d'une exception, gardez en tête de toujours utiliser un bloc `with` par défaut dans vos codes.

Commençons par ouvrir notre fichier `hello.txt`.

```
1 >>> f = open('hello.txt', 'w')
2 >>> f
3 <_io.TextIOWrapper name='hello.txt' mode='w' encoding='UTF-8'>
```

Comme le mode l'indique, il ne nous est pas possible de lire le contenu du fichier, l'opération produirait une erreur.

```
1 >>> f.read()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 io.UnsupportedOperation: not readable
```

Mais il nous est alors possible d'écrire dans le fichier, à l'aide de la méthode `write`.

```
1 >>> f.write('Salut')
2 5
```

La méthode prend naturellement une chaîne de caractères en argument et renvoie le nombre de caractères écrits, c'est-à-dire la longueur de la chaîne dans notre cas.

Si vous regardez alors le contenu de votre fichier depuis un éditeur de texte, il se peut que vous le voyiez vide.

En fait, les fichiers fonctionnent avec une mémoire tampon pour éviter les écritures trop nombreuses sur le disque dur. Cette mémoire est généralement vidée (et donc le contenu du fichier écrit sur le disque) à la fermeture du fichier, lors d'un retour à la ligne ou par une demande

explicite.

Ce dernier cas correspond à la méthode `flush` qui permet donc de valider toutes les opérations d'écriture en cours.

```
1 >>> f.flush()
```

Si vous inspectez à nouveau le contenu du fichier, le contenu devrait cette fois-ci apparaître. Il n'est généralement pas utile de faire appel à `flush`, car celui-ci arrivera bien assez tôt (comme dans les cas expliqués plus haut). Mais à titre d'exemple, vous saurez que la méthode existe et quel effet elle a.

```
1 salut
```

Listing 21 – hello.txt

Nous pouvons maintenant fermer notre fichier (en l'absence de `with`) : `f.close()`.

Pour rappel, notre fichier contenait précédemment le texte «Hello World!», celui-ci a été entièrement effacé lorsque nous avons ouvert le fichier en mode `'w'`. C'est le comportement de Python avec ce mode.

Un autre comportement du mode d'écriture est de créer le fichier de destination si celui-ci n'existe pas.

```
1 with open('newfile.txt', 'w') as f:
2     f.write('I am a new file')
```

Ce code ne provoque pas d'erreur et crée un nouveau fichier `newfile.txt` contenant le texte «I am a new file».

VI.4.1.1. Écrire plusieurs lignes dans un fichier

Vous avez peut-être effectué plusieurs appels successifs à `write` en espérant écrire plusieurs lignes dans un fichier. Mais ça ne fonctionne pas comme ça, vous avez juste obtenu des lettres à la suite.

```
1 >>> with open('alphabet.txt', 'w') as f:
2     ...     f.write('abc')
3     ...     f.write('def')
4     ...     f.write('ghi')
5     ...
6 3
7 3
8 3
9 >>> with open('alphabet.txt', 'r') as f:
10    ...     f.read()
11    ...
12 'abcdefghi'
```

En fait, si vous vous souvenez de la lecture des fichiers, les lignes étaient chaque fois terminées

d'un caractère pour marquer le saut de ligne, `'\n'`. C'est aussi ce caractère que nous devons utiliser pour passer des lignes dans notre fichier.

```

1 >>> with open('alphabet.txt', 'w') as f:
2     ...     f.write('abc\n')
3     ...     f.write('def\n')
4     ...     f.write('ghi\n')
5     ...
6 4
7 4
8 4
9 >>> with open('alphabet.txt', 'r') as f:
10    ...     f.readlines()
11    ...
12 ['abc\n', 'def\n', 'ghi\n']

```

Bien sûr, cela fonctionnerait de la même manière avec un seul appel à `write`, celui-ci n'étant pas lié au nombre de lignes que l'on veut écrire.

```

1 with open('alphabet.txt', 'w') as f:
2     f.write('abc\ndef\nghi\n')

```

Mais on pourra trouver plusieurs appels à `write` si l'on dispose par exemple d'une liste d'éléments à écrire, auquel cas on procèdera avec une boucle `for`.

```

1 lines = ['abc\n', 'def\n', 'ghi\n']
2
3 with open('alphabet.txt', 'w') as f:
4     for line in lines:
5         f.write(line)

```

Notez que les fichiers possèdent déjà une méthode `writelines` pour répondre à ce problème, qui est donc l'inverse de `readlines` (`writelines` prend en argument le même type de valeur que ce que renvoie `readlines`).

```

1 >>> with open('alphabet.txt', 'w') as f:
2     ...     f.writelines(['abc\n', 'def\n', 'ghi\n'])
3     ...
4 >>> with open('alphabet.txt', 'r') as f:
5     ...     f.readlines()
6     ...
7 ['abc\n', 'def\n', 'ghi\n']

```

VI.4.1.2. La fonction `print`

Enfin, sachez qu'il est aussi possible d'utiliser la fonction `print` pour écrire dans des fichiers. Par défaut cette fonction écrit son résultat sur le terminal (qui est vu comme un fichier par le système), mais il est possible de choisir une autre sortie (un autre fichier) avec l'argument nommé `file`.

```
1 >>> with open('hello.txt', 'w') as f:
2 ...     print('Hello', 'World!', file=f)
3 ...
4 >>> with open('hello.txt', 'r') as f:
5 ...     f.read()
6 ...
7 'Hello World!\n'
```

La fonction procède de la même manière que sur le terminal et espace donc les arguments, puis ajoute un saut de ligne à la fin.

Cela permet aussi facilement d'écrire vers un fichier des objets autres que des chaînes de caractères, ce qui n'est pas possible avec des appels à `write` (à moins de convertir préalablement les valeurs).

```
1 >>> with open('types', 'w') as f:
2 ...     print(42, {'a': True}, [1.5], file=f)
3 ...
4 >>> with open('types', 'r') as f:
5 ...     f.read()
6 ...
7 "42 {'a': True} [1.5]\n"
```

Ce saut de ligne ajouté à la fin est le comportement par défaut de `print` mais il est possible de le changer à l'aide de l'argument nommé `end`, qui prend une chaîne de caractères comme marqueur de fin de ligne.

```
1 >>> print('hello', 'world', end='!\n')
2 hello world!
3 >>> print('hello', 'world', end='!')
4 hello world!>>>
```



Le résultat sans `\n` peut paraître surprenant. Les `>>` sont en fait l'invite de commande de Python : comme il n'y a pas eu de saut de ligne, il apparaît à la suite.

Dans un fichier cela donnerait les résultats que l'on pouvait avoir précédemment avec `write`.

```
1 >>> with open('hello.txt', 'w') as f:
2 ...     print('Hello', file=f, end=' ')
```



```

3 ...     print('World!', file=f)
4 ...
5 >>> with open('hello.txt', 'r') as f:
6 ...     f.read()
7 ...
8 'Hello World!\n'

```

Cela est bien sûr compatible avec l'argument `sep` pour préciser le séparateur de valeurs.

```

1 >>> with open('hello.txt', 'w') as f:
2 ...     print('Hello', 'World', file=f, sep=' - ', end='!\n')
3 ...
4 >>> with open('hello.txt', 'r') as f:
5 ...     f.read()
6 ...
7 'Hello - World!\n'

```

Enfin la fonction `print` prend aussi un argument optionnel `flush` recevant un booléen, et qui permet donc un appel automatique à la méthode `flush` si `True` lui est passé. Ce n'est encore une fois utile que dans de rares cas, et pour des écritures qui ne seraient pas déjà terminées d'un saut de ligne.

VI.4.2. Autres modes des fichiers

On l'a vu, le mode `'w'` a pour effet de supprimer le contenu du fichier pour partir sur un contenu vierge.

```

1 with open('hello.txt', 'w') as f:
2     f.write('salut')

```

Ce n'est pas toujours le comportement voulu, et c'est pourquoi il existe différents modes d'ouverture.

VI.4.2.1. Insérer à la fin du fichier

On a ainsi un mode `'a'` (pour *append*, ajouter) qui permet d'insérer du texte à la fin du fichier. C'est-à-dire que tout le contenu déjà présent sera conservé, les modifications apportées seront simplement ajoutées au fichier.

Voyez par exemple avec notre fichier `hello.txt` contenant pour le moment `salut`.

```

1 >>> with open('hello.txt', 'a') as f:
2 ...     f.write(' tout le monde')
3 ...
4 14
5 >>> with open('hello.txt', 'r') as f:

```

```
6 ...     f.read()
7 ...
8 'salut tout le monde'
```

C'est un mode qui peut être particulièrement utile pour des outils de journalisation, car cela évite les conflits entre de multiples écritures.

VI.4.2.2. Créer un fichier

On l'a vu, le mode `'w'` crée le fichier s'il n'existe pas. Il existe un mode plus strict, `'x'` (pour *eXclusif*), spécialement dédié à la création de fichier : ce mode échouera si le fichier existe déjà.

```
1 >>> with open('hello.txt', 'x') as f:
2 ...     pass
3 ...
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   FileNotFoundError: [Errno 17] File exists: 'hello.txt'
```

Mais dans le cas d'un fichier inexistant, il aura le même effet que le mode `'w'`. C'est un mode qui permet par exemple d'éviter que deux programmes concurrents n'écrasent un fichier en croyant le créer.

?

Quelle est donc cette instruction `pass` ?

C'est une instruction Python qui permet juste de ne rien faire, elle permet de conclure un bloc indenté (quand Python attend quelque chose) sans rien faire de particulier, juste *passer*.

Elle n'est pas équivalente à `...`, qui est une expression et possède donc une valeur (Ellipsis).

```
1 >>> with open('newfile.txt', 'x') as f:
2 ...     f.write('New file')
3 ...
4 8
```

VI.4.2.3. Lire et écrire à la fois

Nous avons vu que nous pouvions ouvrir un fichier pour le lire ou pour y écrire, mais il est aussi possible d'y faire les deux à la fois. Cela se fait avec le mode `'r+'`, dédié à la mise à jour (*update*).

```

1 >>> with open('hello.txt', 'r+') as f:
2     ...     f.read()
3     ...     f.write('!!!')
4     ...
5 'salut tout le monde'
6 3
7 >>> with open('hello.txt', 'r') as f:
8     ...     f.read()
9     ...
10 'salut tout le monde!!!'

```

Mais attention à ne pas vous emmêler avec les lectures/écritures et la mémoire tampon, sachant qu'il n'y a qu'un unique curseur dans le fichier. Il est ainsi possible d'écraser des portions du fichier qui n'ont pas encore été lues, c'est pourquoi il faut être vigilant lors de l'utilisation de ce mode.

```

1 >>> with open('hello.txt', 'r+') as f:
2     ...     f.write('>>>')
3     ...     f.read()
4     ...
5 3
6 'ut tout le monde!!!'
7 >>> with open('hello.txt', 'r') as f:
8     ...     f.read()
9     ...
10 '>>>ut tout le monde!!!'

```

Pensez donc aux méthodes `seek` et `flush` qui pourraient vous être utiles pour vous déplacer dans le fichier et vider le tampon.

De façon similaire on trouve aussi des modes de mise à jour en ajout ('`a+`'), en troncature ('`w+`') et en création ('`x+`'). Le premier aura pour effet de placer le curseur à la fin du fichier, et le second d'effacer le contenu actuel du fichier.

Il ne s'agit ici que de modes pour opérer sur les fichiers en mode texte, nous verrons par la suite comment traiter les fichiers binaires.

VI.5. Chaînes de formatage

VI.5.1. Opérations de formatage

VI.5.1.1. Méthode `format`

La méthode `write` des fichiers textes ne comprend que les chaînes de caractères. Nous avons vu qu'il était possible avec `print` d'écrire d'autres types de données, mais les options de formatage sont relativement limitées.

Par exemple il ne nous est pas possible de choisir combien de chiffres après la virgule on veut afficher pour les décimaux (souvenez-vous des erreurs d'arrondis qui donnent parfois des résultats inattendus) ni gérer facilement les espacements.

Tout cela relève du formatage, et il existe une méthode dédiée : la méthode `format` des chaînes de caractères. C'est une méthode qui permet de générer une nouvelle chaîne à partir d'un patron (la chaîne d'origine) et d'arguments.

Pour cela, une syntaxe particulière est utilisée dans le chaîne faisant office de patron pour définir où seront insérés les arguments (on parle de *placeholders*). Cette syntaxe, c'est `{}`.

Lors d'un appel à `format` sur une chaîne de caractères, les `{}` seront repérés dans la chaîne et remplacés par les arguments.

```
1 >>> '{}'.format(10)
2 '10'
3 >>> '{}'.format(1.5)
4 '1.5'
5 >>> '{}'.format('abc')
6 'abc'
```

Chaque `{}` correspond à la valeur suivante dans la liste des arguments.

```
1 >>> '{}-{}'.format(10, 'abc')
2 '10-abc'
3 >>> '{} dit à {} : {}'.format('Alice', 'Bob', 'Salut')
4 'Alice dit à Bob : Salut'
```

Mais ces accolades ne sont pas destinées à rester éternellement vides, on peut y préciser différents types de choses. Déjà, cela peut servir à spécifier l'argument que l'on souhaite utiliser : il peut arriver qu'on veuille afficher le deuxième argument avant le premier par exemple. On entre donc simplement le numéro de l'argument positionnel entre les accolades pour y faire référence (0 étant le premier argument, 1 le deuxième, etc.).

```

1 >>> '{0}-{1}'.format(10, 'abc')
2 '10-abc'
3 >>> '{1}-{0}'.format(10, 'abc')
4 'abc-10'

```

La méthode `format` ne se limite pas aux arguments positionnels mais accepte aussi les arguments nommés, ce qui permet de gagner en clarté. Pour utiliser un argument nommé, il faudra nécessairement préciser son nom entre les accolades.

```

1 >>> '{number}-{name}'.format(number=10, name='abc')
2 '10-abc'
3 >>> '{speaker} dit à {listener} : {sentence}'.format(listener='Alice', speaker='Bob', sentence='Salut')
4 'Bob dit à Alice : Salut'

```

Il est possible de mixer arguments positionnels et nommés, mais attention à ne pas perdre en lisibilité.

```

1 >>> '{0}-{name}'.format(10, name='abc')
2 '10-abc'

```

On peut aussi accéder directement aux attributs ou éléments de l'argument positionnel ou nommé, en utilisant le point pour les attributs et les crochets pour les éléments.

```

1 >>> '{0.real}-{items[1]}'.format(1+2j, items=['a', 'b', 'c'])
2 '1.0-b'

```

Voilà pour le placement des arguments mais ce n'est pas tout : le principal intérêt de cette méthode `format` est de pouvoir... formater les valeurs, leur donner le format que l'on souhaite.

VI.5.1.2. Options de formatage

Il existe pour cela un mini-langage dédié aux options de formatage. Ces options se placeront toujours derrière un signe `:` entre les accolades.

Par exemple, il est possible en utilisant un nombre comme option d'aligner le texte sur un certain nombre de caractères. On l'appelle la largeur de champ.

```

1 >>> '{:10}'.format('abc')
2 'abc          '

```

Par défaut le texte sera aligné à gauche (espaces ajoutées à droite). Il est possible d'être explicite là-dessus en faisant précéder le nombre d'un `<`.

Mais on peut aussi utiliser `>` ou `^` pour l'aligner à droite ou le centrer.

```

1 >>> '{:<10}'.format('abc')
2 'abc          '
3 >>> '{:>10}'.format('abc')
4 '          abc'
5 >>> '{:^10}'.format('abc')
6 '    abc    '

```

Pour le formatage des nombres, on peut préciser l'option (espace) qui a pour effet d'ajouter une espace avant les nombres positifs, de façon à les aligner avec les négatifs (qui commencent par un caractère `-`).

De même on peut utiliser l'option `+` pour afficher explicitement le `+` des nombres positifs.

```

1 >>> '{: }'.format(5)
2 ' 5'
3 >>> '{: }'.format(-5)
4 '-5'
5 >>> '{: +}'.format(5)
6 '+5'
7 >>> '{: +}'.format(-5)
8 '-5'

```

Pour les nombres entiers, on peut utiliser les caractères `x`, `o` ou `b` comme options pour choisir la base dans laquelle le nombre sera écrit. Avec `x`, le nombre sera écrit en hexadécimal, en octal avec `o` et en binaire avec `b`.

```

1 >>> '{:x}'.format(42)
2 '2a'
3 >>> '{:o}'.format(42)
4 '52'
5 >>> '{:b}'.format(42)
6 '101010'

```

On peut ajouter un `#` avant ce caractère pour insérer un préfixe indiquant la base utilisée.

```

1 >>> '{:#x}'.format(42)
2 '0x2a'
3 >>> '{:#b}'.format(42)
4 '0b101010'

```

La largeur de champ est aussi utilisable pour les nombres, ils seront par défaut alignés à droite. On peut préfixer cette largeur de champ d'un `0` pour compléter le nombre avec des zéros plutôt qu'avec des espaces.

```

1 >>> '{:5}'.format(123)
2 ' 123'
3 >>> '{:05}'.format(123)
4 '00123'

```

Pour ce qui est des nombres flottants, on peut utiliser l'option `.` suivie d'un nombre pour indiquer la précision. Ce nombre correspond au nombre maximum de chiffres que l'on veut afficher, cela compte les chiffres avant et après la virgule (sauf les zéros initiaux)

```

1 >>> '{}'.format(0.1+0.2)
2 '0.30000000000000004'
3 >>> '{:.1}'.format(0.1+0.2)
4 '0.3'
5 >>> '{}'.format(1/3)
6 '0.3333333333333333'
7 >>> '{:.5}'.format(1/3)
8 '0.33333'
9 >>> '{:.5}'.format(4/3)
10 '1.3333'
11 >>> '{:.5}'.format(1/30)
12 '0.03333'

```

Il existe aussi une option `%` pour afficher un nombre flottant sous la forme d'un pourcentage. On peut ajouter une précision (avec un point) à ce pourcentage, qui cette fois-ci précise le nombre de chiffres après la virgule uniquement.

```

1 >>> '{:%}'.format(1/2)
2 '50.000000%'
3 >>> '{:.1%}'.format(1/3)
4 '33.3%'

```

Étant donné que les accolades ont un effet bien particulier au sein des chaînes de formatage, il est nécessaire de les échapper pour les ajouter en tant que caractères. Il faut pour cela les doubler. `{{` correspondra au caractère `{` dans une chaîne de formatage, et `}}` au caractère `}`.

```

1 >>> '{} {}'.format(1, 2)
2 '1 {} 2'

```

Ces options de formatage ne sont pas exhaustives, et vous les trouverez plus en détails dans la [documentation détaillée](#) [↗](#).

Vous pouvez aussi obtenir plus d'aide sur le formatage à l'aide de l'appel `help('FORMATTING')` depuis l'interpréteur interactif.

VI.5.1.3. Operateur %

Une autre méthode de formatage plus ancienne existe aussi en Python, elle utilise l'opérateur %.

On applique donc cet opérateur à une chaîne (à gauche) en lui donnant un tuple d'arguments (à droite). Comme précédemment, la chaîne suit un certain format pour définir où seront insérés les arguments.

Ici, le format est celui utilisé par la fonction `printf` en C, où l'on identifie les valeurs par leur type : %s pour une chaîne de caractère, %d pour un nombre entier ou encore %f pour un flottant. Les arguments seront toujours pris successivement dans le tuple qui les fournit (comme les arguments positionnels avec {}).

En pratique, on a donc quelque chose de la sorte :

```
1 >>> '%s dit à %s: tu me dois %d€' % ('Bob', 'Alice', 20)
2 'Bob dit à Alice: tu me dois 20€'
```

On trouve aussi la possibilité de préciser des options telle que la largeur de champ ou la précision, en les insérant entre le % et le caractère représentant le type.

```
1 >>> '%10s répond: il ne me reste que %.2f€' % ('Alice', 18.5)
2 '      Alice répond: il ne me reste que 18.50€'
```

Et il existe encore d'autres possibilités (types et options). Mais cette syntaxe est de moins en moins utilisée en Python, c'est pourquoi je n'en parlerai pas plus longuement. Vous trouverez cependant plus d'informations à son sujet [dans la documentation](#) .

Sachez que tout ce qu'il est possible de faire sur les chaînes de caractères avec % est aussi réalisable avec la méthode `format`. Et nous allons maintenant voir une forme encore plus simple d'utilisation.

VI.5.2. f-strings

Les *f-strings* ou chaînes de formatage sont une nouveauté apportée par Python 3.6 qui simplifie la création de chaînes de caractères dynamiques (se construisant à partir d'autres valeurs). Elles se caractérisent par un préfixe `f` placé avant les guillemets délimitant la chaîne.

```
1 >>> f'abc'
2 'abc'
```

Il ne s'agit pas d'un type particulier, on le voit car notre expression a juste renvoyé la chaîne `'abc'`. Le préfixe signale simplement qu'il peut y avoir des choses à interpréter à l'intérieur de notre chaîne.

Et ces choses, elles sont similaires à ce que l'on faisait avec `str.format`. Dans les chaînes de formatage, on va pouvoir trouver des séquences entre accolades pour signaler où l'on souhaite insérer des valeurs.

Ainsi, `'{} + {} = {}'.format(3, 5, 3 + 5)` deviendra `f' {3} + {5} = {3 + 5}'`. Plus court et plus clair.

VI. Entrées / sorties

Ici il n'est donc pas question de préciser des positions entre les accolades, mais des expressions. Il est ainsi possible de capturer des variables pour les utiliser dans la chaîne.

```
1 >>> a = 3
2 >>> b = 5
3 >>> f'{a} + {b} = {a+b}'
4 '3 + 5 = 8'
```

Bien sûr, tous les types de données y sont utilisables.

```
1 >>> name = 'Max'
2 >>> f'Salut {name} !'
3 'Salut Max !'
```

Et tous types d'expressions sont valides à l'intérieur de ces accolades. Il faut juste faire attention à ne pas s'emmêler les pinceaux avec les guillemets : on ne peut pas placer d'apostrophe dans une chaîne délimitée par des apostrophes par exemple.

```
1 >>> fruits = {'a': 'abricot', 'b': 'banane'}
2 >>> f"{fruits['b']}, {len(fruits)}"
3 'banane, 2'
```

Entre accolades, on peut aussi placer un `:` et y ajouter toutes les options de formatage disponibles avec `str.format`.

```
1 >>> f'{a} + {b} = {a+b:+'}'
2 '3 + 5 = +8'
3 >>> f'Salut {name:10} !'
4 'Salut Max          !'
```

VI.6. Gérer les exceptions (try/except)

Introduction

Quand on code un programme il peut arriver que tout ne se passe pas comme prévu, que des exceptions surviennent qui interrompent le déroulé normal du programme.

Ce chapitre a pour but de vous présenter le fonctionnement des exceptions et la manière de les gérer.

VI.6.1. Tout ne se passe pas comme prévu

On a déjà rencontré des exceptions, ce sont les erreurs qui se produisent quand une opération échoue (conversion impossible, élément inexistant dans un dictionnaire, ouverture d'un fichier introuvable, etc.). L'erreur survient alors sous la forme d'une exception avec un type particulier (`ValueError`, `TypeError`, `KeyError`, etc.).

Le souci c'est que cela coupe l'exécution de la fonction et du programme (hors interpréteur interactif).

Imaginons que nous souhaitions au chargement de notre jeu regarder si une sauvegarde existe. On essaierait alors d'ouvrir le fichier de sauvegarde, et s'il n'existe pas on obtiendrait une exception.

```
1 with open('game.sav') as save:
2     state = load_game(save.read())
3
4 print('Jeu en cours...')
```

Listing 22 – game.py

À l'exécution :

```
1 Traceback (most recent call last):
2   File "<stdin>", line 1, in <module>
3   FileNotFoundError: [Errno 2] No such file or directory: 'game.sav'
```

Ainsi, le programme s'arrête à l'exception, ce qui est plutôt embêtant. Notre jeu devrait être en mesure de démarrer sans sauvegarde existante, de traiter l'erreur et de continuer.

Pour autant une exception peut être un comportement attendu, d'autant plus si elle provient d'une valeur entrée par l'utilisateur. Dans une calculatrice, on ne veut pas que le programme plante si l'utilisateur demande une division par zéro. De même dans un annuaire si un nom n'est pas trouvé.

```

1 def calculatrice(a, op, b):
2     if op == '+':
3         return a + b
4     if op == '-':
5         return a - b
6     if op == '*':
7         return a * b
8     if op == '/':
9         return a / b
10    print('Calcul impossible')

```

```

1 >>> calculatrice(3, '+', 0)
2 3
3 >>> calculatrice(3, '/', 0)
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "<stdin>", line 9, in calculatrice
7 ZeroDivisionError: division by zero

```

Comment alors peut-on gérer ces erreurs pour éviter cela ?

VI.6.2. Éviter l'exception

Une solution pour éviter les erreurs est d'empêcher qu'elles se produisent. Ainsi, avant d'exécuter une action, on va tester différents cas d'erreurs pour les écarter. C'est la stratégie dite *LBYL* (*Look before you leap*, soit *réfléchis avant d'agir*).

Par exemple pour une calculatrice dans le cadre d'une division, on testerait si le quotient n'est pas nul avant de réaliser l'opération.

```

1 def calculatrice(a, op, b):
2     if op == '+':
3         return a + b
4     if op == '-':
5         return a - b
6     if op == '*':
7         return a * b
8     if op == '/' and b != 0:
9         return a / b
10    print('Calcul impossible')

```

```

1 >>> calculatrice(3, '/', 2)
2 1.5
3 >>> calculatrice(3, '/', 0)
4 Calcul impossible

```

Pour notre problématique de sauvegarde, il faudrait donc être en mesure de tester si un fichier existe. Une telle fonctionnalité est disponible dans le module `pathlib`. Ce module propose un type `Path` représentant un chemin sur le système de fichiers, et bénéficiant naturellement d'une méthode `exists` renvoyant un booléen pour tester si le chemin existe ou non.

```
1 >>> from pathlib import Path
2 >>> p = Path('hello.txt')
3 >>> p.exists()
4 True
5 >>> p = Path('game.sav')
6 >>> p.exists()
7 False
```

Ainsi, on peut remplacer notre code de chargement par :

```
1 from pathlib import Path
2
3 if Path('game.sav').exists():
4     with open('game.sav') as save:
5         state = load_game(save.read())
6 else:
7     state = None
```

On notera que les objets `Path` possèdent aussi une méthode `open` équivalente à la fonction du même nom : `Path(foo).open()` revient à écrire `open(foo)`. On peut alors améliorer notre code précédent pour éviter les répétitions.

```
1 save_path = Path('game.sav')
2
3 if save_path.exists():
4     with save_path.open() as save:
5         state = load_game(save.read())
6 else:
7     state = None
```

VI.6.2.1. Limites

La stratégie *LBYL* est cependant limitée. Déjà, il est difficile d'envisager tous les cas d'erreurs : on pourrait obtenir une exception parce que le fichier est un répertoire, parce que les permissions ne sont pas suffisantes pour le lire, etc.

Mais considérons que l'on arrive à anticiper toutes les erreurs possibles, il resterait un problème. Quand on demande au système si un fichier existe, il le vérifie à l'instant t ; mais quand on l'ouvre nous sommes à l'instant $t+1$.

Pendant ce très court laps de temps le fichier a pu être supprimé, déplacé, ses permissions modifiées, et donc on n'échapperait pas à l'exception.

Il va alors nous falloir adopter une autre stratégie, dite *EAFP* (*Easier to ask for forgiveness than permission, il est plus simple de demander pardon que demander la permission*). C'est-à-dire

laisser l'exception se produire et la traiter ensuite, comme nous allons le voir tout de suite. Pour autant, la stratégie *LBYL* n'est pas à jeter, il reste des cas où elle est parfaitement adaptée, quand les conditions ne sont pas amenées à changer entre les pré-conditions et l'opération. C'est le cas par exemple du test pour le quotient nul dans la division, s'il est non-nul à l'instant t , il sera toujours à $t+1$.

VI.6.3. Traiter l'exception

Pour gérer les exceptions on va utiliser un nouveau type de bloc, ou plutôt un couple de blocs, introduits par les mots-clés `try` et `except` (littéralement «essaie» et «à l'exception de»).

Ces deux mots-clés vont de paire pour intercepter les erreurs.

Dans le bloc `try` on place le code qui peut échouer, et le bloc `except` sera exécuté si et seulement si une exception survient. Il aura pour effet d'attraper cette exception et donc éviter que le programme ne plante, en proposant un traitement adapté.

```
1 >>> try:
2 ...     result = 1 / 0
3 ... except:
4 ...     print('Division par zéro')
5 ...
6 Division par zéro
```

Ici notre traitement est simplement d'afficher un message, mais il est possible de faire ce que l'on veut dans le bloc `except`, comme renvoyer une valeur particulière.

```
1 def division(a, b):
2     try:
3         return a / b
4     except:
5         return float('nan')
```

?

Quel est ce `float('nan')` ?

NaN, pour *Not a Number* (*Pas un Nombre*), est une valeur particulière de la norme des nombres flottants évoquant un résultat qui ne serait pas un nombre.

On y accède en Python via la variable `nan` du module `math`, ou avec un simple `float('nan')`.

```
1 >>> division(3, 5)
2 0.6
3 >>> division(4, 2)
4 2.0
5 >>> division(10, 0)
6 nan
```

L'exécution du programme reprend normalement à l'issue du `except`. On ne le voit pas dans

VI. Entrées / sorties

l'exemple car on y utilise un `return`, mais la suite de la fonction est bien exécutée.

```
1 def division(a, b):
2     try:
3         result = a / b
4     except:
5         result = float('nan')
6     print('Résultat :', result)
7     return result
```

Si l'exécution s'arrêtait juste après le `except`, nous ne passerions pas dans le `print` et le `return`.

```
1 >>> division(1, 2)
2 Résultat : 0.5
3 0.5
4 >>> division(1, 0)
5 Résultat : nan
6 nan
```

Aussi, nous utilisons `except` sans lui préciser aucun argument, il attrapera donc toute exception qui surviendrait, quel qu'en soit son type.

```
1 >>> division('x', 'y')
2 Résultat : nan
3 nan
```

Pourtant ce n'est pas toujours souhaitable. Par exemple dans le cas présent il s'agit d'une erreur de type et donc d'un mauvais usage de la fonction, on pourrait préférer ne pas traiter cette exception et la laisser survenir.

Ainsi, on pourra préciser derrière `except` le type de l'exception que l'on veut attraper, dans notre cas `ZeroDivisionError`.

```
1 def division(a, b):
2     try:
3         return a / b
4     except ZeroDivisionError:
5         return float('nan')
```

Notre fonction interceptera maintenant les erreurs de division par zéro, et uniquement celles-ci.

```
1 >>> division(1, 2)
2 0.5
3 >>> division(1, 0)
4 nan
```

```

5 >>> division(1, 'x')
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "<stdin>", line 3, in division
9   TypeError: unsupported operand type(s) for /: 'int' and 'str'

```

VI.6.3.1. Attraper plusieurs exceptions

On peut placer plusieurs blocs `except` à la suite d'un `try` pour traiter des exceptions différentes. Sur le même principe que les `if / elif / else`, un seul de ces blocs sera exécuté, le premier qui correspond à l'exception survenue.

Changeons d'exemple et passons à un cas plus réel de lecture de fichier. Imaginons que l'on souhaite simplement lire un score dans un fichier. Il nous faut alors une fonction prenant un chemin de fichier en paramètre et renvoyant son contenu sous forme de nombre.

Plusieurs exceptions peuvent survenir comme on l'a vu : le fichier peut ne pas exister ou ne pas avoir les bonnes permissions (erreurs `OSError`), peut contenir une valeur invalide (`ValueError`) et d'autres encore.

```

1 def get_score(path):
2     try:
3         with open(path) as f:
4             return int(f.read())
5     except OSError:
6         print("Impossible d'ouvrir le fichier")
7     except ValueError:
8         print('Score invalide')

```

Maintenant voilà ce que l'on obtient avec un fichier `score.txt` contenant `42` et un fichier `hello.txt` quelconque.

```

1 >>> get_score('score.txt')
2 42
3 >>> get_score('hello.txt')
4 Score invalide
5 >>> get_score('not_found.txt')
6 Impossible d'ouvrir le fichier

```

Bien sûr, les blocs `except` ne peuvent attraper que les exceptions qui surviendraient pendant l'exécution du `try`. Toute exception survenue avant leur échapperait.

```

1 def get_score(path):
2     with open(path) as f:
3         try:
4             return int(f.read())
5         except OSError:

```

```

6         print("Impossible d'ouvrir le fichier")
7     except ValueError:
8         print('Score invalide')

```

Dans l'exemple précédent, la conversion du contenu du fichier en nombre a toujours lieu dans le `try` donc l'erreur sur `hello.txt` sera bien traitée. Mais l'ouverture du fichier se situe en dehors, nous ne gérons donc pas l'erreur `OSError` sur `not_found.txt`.

```

1 >>> get_score('score.txt')
2 42
3 >>> get_score('hello.txt')
4 Score invalide
5 >>> get_score('not_found.txt')
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "<stdin>", line 2, in get_score
9   FileNotFoundError: [Errno 2] No such file or directory:
    'not_found.txt'

```



On voit que l'erreur qui survient est une `FileNotFoundError` et non une `OSError`. Il faut savoir qu'il existe une hiérarchie des exceptions que nous étudierons plus tard, et que `FileNotFoundError` est une erreur qui descend de `OSError`.

Aussi, l'exécution d'un bloc `try` s'arrête à la première erreur rencontrée. Cela signifie que tout son contenu n'est pas nécessairement exécuté, donc certaines variables définies dans le `try` n'existent peut-être pas.

Essayez la fonction suivante pour vous rendre compte des problèmes que cela peut poser.

```

1 def get_score(path):
2     try:
3         with open(path) as f:
4             content = f.read()
5             score = int(content)
6     except OSError:
7         print("Impossible d'ouvrir le fichier")
8     except ValueError:
9         print('Score invalide')
10    return score

```

La fonction de cet exemple gère mal les exceptions : `score` ne sera jamais définie si une erreur est survenue, et donc le `return` échouera car accèdera à une variable inexistante.

Quant aux variables `f` et `content` on ne sait pas si elles existent car cela dépend de l'endroit précis où est survenu l'erreur. Pour une erreur à l'ouverture du fichier `content` ne sera pas définie, mais s'il s'agit d'une erreur lors de la conversion alors `content` contiendra sa bonne valeur.

Pour s'assurer que ces variables existent, il nous faut alors les définir dans tous les cas. Soit en

le faisant avant le `try` (puisque c'est du code qui sera toujours exécuté), soit en répétant la définition dans chaque clause `except`.

```

1 def get_score(path):
2     score = None
3
4     try:
5         with open(path) as f:
6             content = f.read()
7             score = int(content)
8     except OSError:
9         print("Impossible d'ouvrir le fichier")
10    except ValueError:
11        print('Score invalide')
12
13    return score

```

VI.6.3.2. Remontée d'erreurs

On peut voir l'exécution d'un programme informatique comme le parcours d'un arbre, de branche en branche, de façon à passer par toutes les feuilles. Les embranchements étant faits de conditions, de boucles et d'appels de fonctions. Notamment d'appels de fonctions.

À chaque instant du programme, l'instruction en cours d'exécution représente une curseur le long d'une branche : l'appel d'une fonction fait aller ce curseur plus loin dans l'arbre tandis qu'un retour le fait revenir sur ses pas.

Ainsi, il existe toujours un chemin depuis la racine du programme (le tronc) jusque la position actuelle du curseur.

Ce chemin représente la pile d'appels courante (*stacktrace*), les fonctions qu'il a fallu parcourir pour arriver jusqu'à ce point du programme. Toute exception est liée à la position courante dans le programme, au contexte qui l'a fait surgir, et donc à un certain état de la pile d'appels.

Cette pile liée à l'exception, on la voit d'ailleurs apparaître dans le terminal quand on n'attrape pas l'exception.

```

1 def division(a, b):
2     return a / b
3
4 def inverse(x):
5     return division(1, x)
6
7 def main():
8     for i in range(10):
9         print(i, inverse(i))
10
11 main()

```

Listing 23 – error.py

```

1 % python error.py
2 Traceback (most recent call last):
3   File "error.py", line 11, in <module>
4     main()
5   File "error.py", line 9, in main
6     print(i, inverse(i))
7   File "error.py", line 5, in inverse
8     return division(1, x)
9   File "error.py", line 2, in division
10    return a / b
11 ZeroDivisionError: division by zero

```

De haut en bas, on voit que l'appel à `main` ligne 11 a provoqué un appel à `inverse` ligne 9, qui induit lui-même un appel à `division` ligne 5, à l'intérieur de laquelle se produit l'erreur (ligne 2).

Quand une exception n'est pas attrapée, elle remonte pas à pas la pile d'appels, et continue sa route jusqu'à couper le programme lui-même.

Car oui, il n'existe pas un seul endroit où l'exception peut être attrapée, elle peut l'être tout le long du programme. On pourrait choisir de placer un `try / except` dans la fonction `division`, mais aussi dans `inverse` ou dans `main`. Choisir de le mettre dans la boucle ou à l'extérieur, chaque solution ayant un comportement différent.

Par exemple, attraper l'exception à l'extérieur de la boucle aura pour effet de s'arrêter à la première erreur, puisque la boucle sera coupée à la première itération (`i = 0`).

```

7 def main():
8     try:
9         for i in range(10):
10            print(i, inverse(i))
11    except ZeroDivisionError:
12        pass

```

Listing 24 – error.py

```

1 % python error.py

```

Alors qu'attraper l'exception à l'intérieur de la boucle permettra de ne couper que l'itération courante puis de passer à la suivante.

```

7 def main():
8     for i in range(10):
9         try:
10            print(i, inverse(i))
11        except ZeroDivisionError:
12            pass

```

Listing 25 – error.py

```

1 % python error.py
2 1 1.0
3 2 0.5
4 3 0.3333333333333333
5 4 0.25
6 5 0.2
7 6 0.16666666666666666
8 7 0.14285714285714285
9 8 0.125
10 9 0.11111111111111111

```

Mais dans cet exemple, les appels à `inverse(0)` ou `division(1, 0)` continuent d'échouer : on pourrait choisir de traiter l'erreur dans ces fonctions pour renvoyer *NaN*.

```

1 def division(a, b):
2     try:
3         return a / b
4     except ZeroDivisionError:
5         return float('nan')

```

Listing 26 – error.py

```

1 % python error.py
2 0 nan
3 1 1.0
4 2 0.5
5 3 0.3333333333333333
6 4 0.25
7 5 0.2
8 6 0.16666666666666666
9 7 0.14285714285714285
10 8 0.125
11 9 0.11111111111111111

```

Il convient alors chaque fois de réfléchir au comportement que l'on veut adopter et de placer judicieusement les blocs `try` / `except` en fonction de cela, pour n'être ni trop large, ni trop fin.

VI.7. Formater les données

Introduction

On a vu comment ouvrir des fichiers et y écrire du texte, mais toutes les données que nous manipulons ne sont pas du texte. Bien sûr il est possible de les convertir, c'est d'ailleurs ce que fait la fonction `print` sur ce qu'elle reçoit, mais comment conserver une structure des données ?

Par exemple pour notre sauvegarde il va nous falloir enregistrer l'état du jeu, tout ce qui différencie la partie en cours d'une autre : les noms des monstres et leurs points de vie. Il s'agit donc de données de types différents (chaînes de caractères, nombres) qu'il va nous falloir représenter, en utilisant pour cela un format de données adéquat. Le format est une notion un peu abstraite qui explique de quelle manière les données doivent être traitées, comment elles peuvent être reconstruites à partir de leur représentation.

Tous les fichiers que l'on utilise représentent leurs données selon un certain format, et tous les formats ne permettent pas de stocker la même chose, ils ont chacun leurs particularités. On ne représente pas une image de la même manière qu'une musique par exemple.

On appelle sérialisation l'opération qui permet de transformer en texte des données structurées, de façon à pouvoir reconstruire ces données ensuite. À l'inverse, cette reconstruction s'appelle une désérialisation. On parle aussi de *parsing* pour qualifier l'analyse syntaxique du texte et l'extraction des données.

VI.7.1. Format JSON

Un premier format de données assez courant en informatique est le JSON (pour *JavaScript Object Notation*) qui comme son nom l'indique provient du Javascript. Il s'est ainsi répandu dans le monde du web pour devenir un format de prédilection pour les échanges entre applications. C'est un format textuel, c'est-à-dire qu'il est lisible à l'œil sous forme de texte (contrairement à un format binaire), bien que parfois difficile à écrire à la main.

Voici à quoi ressemble un document JSON :

```
1 {  
2   "id": "001",  
3   "name": "Pythachu",  
4   "type": "foudre",  
5   "attaques": ["tonnerre", "charge"],  
6   "base_pv": 50  
7 }
```

Listing 27 – pythachu.json

On le voit, c'est un format qui ressemble beaucoup au Python. Il est cependant plus restreint. Un document JSON ne peut comporter des valeurs que de 7 types :

- `null`, équivalent au `None` de Python.
- `boolean`, un booléen donc, `true` ou `false`.
- `int`, un nombre entier (42).
- `float`, un nombre flottant (1.5, 1e10).
- `str`, une chaîne de caractère, toujours entre double-guillemets ("hello world").
- `array`, un tableau de valeurs, équivalent à une liste Python ([8, "foo"]).
- `object`, l'équivalent plus restreint d'un dictionnaire Python : seules les chaînes de caractères peuvent être utilisées en clés, les types des valeurs sont libres (`{"key": [3, 5]}`).

VI.7.1.1. Module `json`

Ce format est exploitable en Python avec le module `json` de la bibliothèque standard. Le module fournit principalement 4 fonctions : `load`, `loads`, `dump` et `dumps`. Retenez ces noms de fonctions, ils sont courants en Python et communs à beaucoup de modules de sérialisation.

VI.7.1.1.1. Lecture

`load` est une fonction qui prend en argument un fichier (un objet-fichier ouvert en lecture au préalable) et traite son contenu afin de le renvoyer sous la forme d'un objet Python. Par exemple, avec le document `pythachu.json` présenté plus haut, nous aurions ceci.

```
1 >>> import json
2 >>> with open('pythachu.json') as f:
3 ...     json.load(f)
4 ...
5 {'id': '001', 'name': 'Pythachu', 'type': 'foudre', 'attaques':
   ['tonnerre', 'charge'], 'base_pv': 50}
```

La fonction nous a renvoyé la représentation en Python de notre objet.

`loads` est similaire à `load` mais reçoit en argument une chaîne de caractères plutôt qu'un fichier (*loads* pour *load string*). Elle traite donc le contenu directement depuis la chaîne.

```
1 >>> json.loads('{"name": "Pythard", "base_pv": null}')
2 {'name': 'Pythard', 'base_pv': None}
```

Ces fonctions lèveront une exception si l'entrée n'est pas dans un format correct.

```
1 >>> json.loads('{42: "foo"}')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   [...]
5 json.decoder.JSONDecodeError: Expecting property name enclosed in
   double quotes: line 1 column 2 (char 1)
```

VI.7.1.1.2. Écriture

`dump` et `dumps` sont les fonctions de sérialisation, elles permettent de passer d'un objet Python à sa représentation JSON.

`dumps` reçoit en argument un objet Python et renvoie sa sérialisation sous forme d'une chaîne de caractères.

```
1 >>> json.dumps([1, 2, 3, 'foo'])
2 '[1, 2, 3, "foo"]'
```

`dump` reçoit un objet Python et un fichier (ouvert en écriture), la sérialisation de l'objet sera écrite dans le fichier donné.

```
1 with open('output.json', 'w') as f:
2     json.dump({'key': 'value'}, f)
```

```
1 {"key": "value"}
```

Listing 28 – output.json

Ces deux fonctions prennent aussi un argument nommé `indent` qui permet de préciser l'indentation du document de sortie. Avec `json.dump({'key': 'value'}, f, indent=2)`, nous aurions obtenu le résultat suivant.

```
1 {
2     "key": "value"
3 }
```

Listing 29 – output.json

L'objet passé en argument se doit d'être composé de types convertibles en JSON, une exception sera levée dans le cas contraire.

```
1 >>> json.dumps(1+5j)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4     [...]
5 TypeError: Object of type complex is not JSON serializable
```

VI.7.1.2. Avantages et inconvénients

Les avantages de ce format sont qu'il est très répandu et assez lisible, il est donc adapté pour une communication basique entre programmes (notamment des API web) ou pour sauvegarder des données simples (dans les types supportés par le format).

C'est en revanche un format avec une syntaxe assez stricte, qui ne conviendrait pas à une écriture humaine, évitez-le donc pour un fichier de configuration. Il est assez verbeux et ne se

prête pas forcément à des échanges «intenses» entre programmes. De plus il ne permet pas de représenter tous les objets Python, ce qui peut-être limitant dans certains cas.

VI.7.2. Format XML

Le XML (pour *eXtensible Markup Language*) est un format assez ancien toujours couramment utilisé (SVG, XHTML, docx, ODT).

C'est un langage dit de balisage, formé de différentes balises imbriquées. Il se présente comme suit.

```
1 <monster id="001">
2   <name>Pythachu</name>
3   <type>foudre</type>
4   <attaques>
5     <attaque>tonnerre</attaque>
6     <attaque>charge</attaque>
7   </attaques>
8   <base_pv>50</base_pv>
9 </monster>
```

Listing 30 – pythachu.xml

On le voit donc, une balise XML s'ouvre par un `<balise>` et se ferme avec un `</balise>`, on peut placer à l'intérieur d'autres balises (qui forment donc la hiérarchie du document) ou du texte.

Il est aussi possible de spécifier des attributs aux balises lors de leur ouverture, comme des métadonnées, avec la syntaxe `<balise attribut="valeur">`.

Un document XML ne comprend que le texte et pas d'autres types de valeurs, il vous faudra donc opérer les conversions manuellement lors du traitement du document.

VI.7.2.1. Module `xml`

L'analyse d'un document XML n'est pas aussi simple que celle d'un JSON. Il n'y a pas un unique module pour le faire, et pas de fonction `load` / `dump`, juste des fonctions pour opérer sur le document et aller extraire des informations à un endroit précis.

Il existe plusieurs modules Python dédiés à l'analyse des documents XML, tous regroupés dans le [module `xml`](#) [↗](#). Nous ne nous intéresserons ici qu'au [module `xml.etree`](#) [↗](#).

Pour commencer, on va importer le module `xml.etree.ElementTree` qu'il est courant de simplement appeler `ET`.

```
1 import xml.etree.ElementTree as ET
```

VI.7.2.1.1. Lire un fichier XML

Ensuite, on va ouvrir un document XML à l'aide de la fonction `parse` de ce module. La fonction accepte un chemin de fichier en argument, ou directement un objet-fichier.

```

1 >>> tree = ET.parse('pythachu.xml')
2 >>> tree
3 <xml.etree.ElementTree.ElementTree object at 0x7f6ff11b5f70>

```

i

Il est coutume d'appeler `tree` (*arbre*) un document XML, par rapport à sa structure arborescente.

Une fois ce document chargé, on peut en récupérer l'élément principal (le nœud racine) à l'aide de la méthode `getroot`.

```

1 >>> root = tree.getroot()
2 >>> root
3 <Element 'monster' at 0x7f6ff0faaef0>

```

`root` est un objet de type `Element`. Il possède entre autres un attribut `tag` qui référence le nom de la balise, et un attribut `attrib` qui contient le dictionnaire d'attributs de la balise.

```

1 >>> root.tag
2 'monster'
3 >>> root.attrib
4 {'id': '001'}

```

Notez qu'il existe aussi la fonction `fromstring` pour charger un élément à partir d'une chaîne de caractères.

```

1 >>> ET.fromstring('<foo>bar</foo>')
2 <Element 'foo' at 0x7f6ff10e1900>

```

Cette fonction lève une erreur `ParseError` si la chaîne ne représente pas un document XML valide (il en est de même avec la fonction `parse`).

```

1 >>> ET.fromstring('<foo>bar</foo>')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "/usr/lib/python3.9/xml/etree/ElementTree.py", line 1348,
      in XML
5     return parser.close()
6 xml.etree.ElementTree.ParseError: unclosed token: line 1, column 8

```

Les éléments XML sont des objets Python itérables. Itérer dessus revient à parcourir les balises filles.


```

1 >>> for elem in root:
2     ...     print(elem)
3     ...
4 <Element 'name' at 0x7f6ff0faaf40>
5 <Element 'type' at 0x7f6ff0faaf90>
6 <Element 'attaques' at 0x7f6ff0fad040>
7 <Element 'base_pv' at 0x7f6ff0fad130>

```

Les éléments possèdent aussi une méthode `find` pour directement trouver une balise fille en fonction de son nom.

```

1 >>> root.find('name')
2 <Element 'name' at 0x7f6ff0faaf40>
3 >>> root.find('attaques')
4 <Element 'attaques' at 0x7f6ff0fad040>

```

Quand il existe plusieurs éléments du même nom, la méthode `findall` permet de tous les trouver, elle renvoie une liste d'éléments.

```

1 >>> root.find('attaques').findall('attaque')
2 [<Element 'attaque' at 0x7f6ff0fad090>, <Element 'attaque' at
   0x7f6ff0fad0e0>]

```

Et l'on peut accéder au contenu textuel des éléments à l'aide de leur attribut `text`.

```

1 >>> root.find('name').text
2 'Pythachu'
3 >>> root.find('base_pv').text
4 '50'
5 >>> for attack in root.find('attaques').findall('attaque'):
6     ...     print(attack.text)
7     ...
8 tonnerre
9 charge

```

VI.7.2.1.2. Construire un fichier XML

Il est aussi possible de construire un document XML de toute pièce à l'aide d'`etree`.

On peut pour cela commencer par créer un élément racine en instanciant un objet `Element`, en fournissant le nom de la balise comme argument.

```

1 >>> root = ET.Element('foo')
2 >>> root
3 <Element 'foo' at 0x7f2496c4c8b0>

```

On peut ensuite facilement ajouter des éléments à un élément parent avec la fonction `SubElement`.

```
1 >>> ET.SubElement(root, 'bar')
2 <Element 'bar' at 0x7f2496c4c8b0>
3 >>> ET.SubElement(root, 'baz')
4 <Element 'baz' at 0x7f2496c44f40>
```

Et l'on peut parfaitement ajouter des sous-éléments à un sous-élément, etc.

```
1 >>> sub = ET.SubElement(root, 'list')
2 >>> ET.SubElement(sub, 'item')
3 <Element 'item' at 0x7f2496c619a0>
4 >>> ET.SubElement(sub, 'item')
5 <Element 'item' at 0x7f2496b2af90>
```

On peut aussi manipuler directement le dictionnaire d'attributs des éléments pour en ajouter ou en modifier.

```
1 >>> root.attrib['name'] = 'Doc'
2 >>> root.attrib
3 {'name': 'Doc'}
```

De même que l'on peut redéfinir l'attribut `text` pour ajouter du texte à une balise.

```
1 root.find('bar').text = 'bonjour'
```

Enfin, le module `ET` possède une fonction `dump` pour transformer en chaîne de caractères l'élément que l'on vient de créer.

```
1 >>> ET.dump(root)
2 <foo name="Doc"><bar>bonjour</bar><baz /><list><item /><item
  /></list></foo>
```



Notez que les balises telles que `<baz />` sont des balises auto-fermantes. `<baz/>` est équivalent à `<baz></baz>`, c'est simplement une balise qui ne contient ni enfants ni texte.

Il est aussi possible de créer un document (`ElementTree`) et d'appeler sa méthode `write` pour écrire le document dans un fichier.

```
1 >>> ET.ElementTree(root).write('doc.xml')
```

```

1 <foo name="Doc"><bar>bonjour</bar><baz /><list><item /><item
  /></list></foo>

```

Listing 31 – doc.xml

Il y a beaucoup à dire sur le format XML et tout ne pourra pas être décrit ici. Sachez que c'est un format assez complet, qui comporte des mécanismes de validation (schémas XML), d'espaces de noms (*namespaces*), un sous-langage de requêtage (XPath) et tout un écosystème avec des outils de transformation comme XSLT.

Tous ces termes peuvent vous amener à des ressources complémentaires sur le format XML.

Il est aussi à noter que plusieurs types de parseurs existent pour analyser des documents XML. L'approche de construction d'un document tel que nous l'avons fait ici ([DOM](#) [↗](#) n'est pas la seule.

Il existe par exemple l'[approche SAX](#) [↗](#) qui consiste à ne pas construire le document mais à le parcourir et à appeler des fonctions définies par l'utilisateur pour chaque ouverture/fermeture de balise, ce qui permet de ne pas occuper de place en mémoire. Voyez par exemple [cet article](#) [↗](#) qui utilise la fonction `iterparse` d'`etree` pour analyser un document (l'article nécessite de comprendre [les générateurs](#) [↗](#)).

Enfin, sachez qu'il existe en Python une bibliothèque externe, [lxml](#) [↗](#), qui simplifie l'usage des documents XML.

VI.7.2.2. Avantages et inconvénients

Son ancienneté et les technologies autour (XMLSchema, XSLT, XPath) sont les forces de ce format plutôt décrié pour sa verbosité et sa relative illisibilité.

Un autre avantage se situe au niveau des diverses technologies de *parsing*, notamment le SAX plutôt adapté aux gros documents et à la réception de données au fil de l'eau.

Mais le gros point noir d'un point de vue Python est clairement relatif à ces technologies, il est difficile de savoir par où commencer et de manipuler un document XML, là où JSON est très simple d'utilisation.

VI.7.3. Format INI

Le format INI (*Initialization*) est un format dédié à l'écriture de fichiers de configuration simples.

Il permet de décrire différents paramètres de configuration (sous forme de couples clé-valeur) et de les regrouper en sections.

```

1 [game]
2 save=game.dat
3
4 [window]
5 title=Mon super jeu
6 width=800
7 height=600

```

Listing 32 – config.ini

Ainsi une section est définie par un `[nom_de_la_section]` et réunit en son sein toutes les définitions suivantes (de la forme `cle=valeur`).

Toutes les valeurs sont considérées comme des chaînes de caractères et peuvent donc nécessiter une conversion manuelle au cas par cas (on voudra par exemple convertir les valeurs `width` et `height` vers des nombres).

VI.7.3.1. Module `configparser`

Python propose une implémentation du format INI via [son module `configparser`](#) .

VI.7.3.1.1. Lecture

Afin de lire un document INI, il faut au préalable instancier un objet `ConfigParser`.

```
1 >>> from configparser import ConfigParser
2 >>> config = ConfigParser()
```

Cet objet possède une méthode `read` qui prend un chemin de fichier en argument et complète la configuration à partir du contenu de ce fichier.

```
1 >>> config.read('config.ini')
2 ['config.ini']
```

On peut ensuite accéder aux différentes sections de la configuration à l'aide de la méthode `sections` et utiliser l'objet `config` comme un dictionnaire.

```
1 >>> config.sections()
2 ['game', 'window']
3 >>> config['game']
4 <Section: game>
5 >>> config['window']
6 <Section: window>
```

Les sections elles aussi sont des objets semblables à des dictionnaires que l'on peut donc manipuler pour accéder aux différentes valeurs.

```
1 >>> config['game']['save']
2 'game.dat'
3 >>> config['window']['height']
4 '600'
5 >>> int(config['window']['height'])
6 600
7 >>> dict(config['window'])
8 {'title': 'Mon super jeu', 'width': '800', 'height': '600'}
```

En cas de fichier invalide, la méthode `read` lèvera une exception `configparser.ParsingError`.

```
1 >>> config.read('invalid.ini')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "/usr/lib/python3.10/configparser.py", line 698, in read
5     self._read(fp, filename)
6   File "/usr/lib/python3.10/configparser.py", line 1117, in _read
7     raise e
8 configparser.ParsingError: Source contains parsing errors:
9   'invalid.ini'
10     [line 6]: 'width\n'
```

VI.7.3.1.2. Écriture

En écriture, un objet `ConfigParser` se comporte là aussi comme un dictionnaire.

```
1 >>> config = ConfigParser()
2 >>> config['game'] = {'save': 'new.dat'}
3 >>> config['window'] = {}
4 >>> config['window']['width'] = '200'
```



Attention, toutes les valeurs renseignées dans la configuration doivent être des chaînes de caractères, sans quoi vous obtiendrez une erreur `TypeError`.

Et l'objet possède une méthode `write` pour écrire le contenu de la configuration dans un fichier précédemment ouvert en écriture.

```
1 >>> with open('new.ini', 'w') as configfile:
2 ...     config.write(configfile)
3 ...
```

```
1 [game]
2 save = new.dat
3
4 [window]
5 width = 200
```

Listing 33 – new.ini

VI.7.3.2. Avantages et inconvénients

Le format INI est un format plat (il n'y a pas de structures arborescentes), ce qui est à la fois un avantage et un inconvénient : cela permet de garder des fichiers de configuration simples puisque les constructions complexes n'y sont pas permises.

Ce format a aussi l'avantage d'être clair pour comprendre en un coup d'œil la configuration d'un programme, il est aussi assez répandu.

Son principal inconvénient est de n'autoriser que les chaînes de caractères et donc de forcer les conversions manuelles pour chacune des valeurs.

VI.7.4. Format CSV

Le format CSV (*Comma-Separated Values*) est un format textuel utilisé pour représenter des données tabulaires, comme un tableur. Chaque ligne du fichier correspondra à une ligne du tableau, et les lignes sont divisées en colonnes selon un séparateur (généralement `,` ou `;`).

Voici un exemple de document CSV :

```
1 [game]
2 save = new.dat
3
4 [window]
5 width = 200
```

Listing 34 – attaques.csv

Une première ligne (l'en-tête) identifie les noms des colonnes, elle est facultative, mais il faudra en tenir compte lors de l'analyse du fichier.

Comme en XML, toutes les données du document sont considérées comme du texte, les nombres devront donc être convertis manuellement.

VI.7.4.1. Module `csv`

Le module `csv` [↗](#) de la bibliothèque standard offre ce qu'il faut pour traiter un document CSV.

VI.7.4.1.1. Lecture

Le module fournit une fonction `reader` qui permet de lire un document CSV depuis un fichier. Elle reçoit donc le fichier en argument¹ et renvoie un itérable contenant les lignes du CSV, ces lignes prenant la forme de listes de valeurs.

```
1 >>> import csv
2 >>> with open('attaques.csv') as f:
3 ...     reader = csv.reader(f)
4 ...     for row in reader:
```

1. En réalité tout itérable sur des lignes (chaînes de caractères) est accepté en entrée, un fichier correspond à cette définition.

```

5 ...         print(row)
6 ...
7 ['nom', 'type', 'degats']
8 ['charge', 'normal', '20']
9 ['tonnerre', 'foudre', '50']
10 ['jet-de-flotte', 'aquatique', '40']
11 ['brûlure', 'flamme', '40']

```

Comme on le voit notre en-tête est considérée comme une ligne à part entière. Il serait néanmoins possible de l'isoler en utilisant par exemple la fonction `next` de Python (je reviendrai plus tard sur cette fonction).

```

1 >>> with open('attaques.csv') as f:
2 ...     reader = csv.reader(f)
3 ...     header = next(reader)
4 ...     print('en-tête:', header)
5 ...     for row in reader:
6 ...         print(row)
7 ...
8 en-tête: ['nom', 'type', 'degats']
9 ['charge', 'normal', '20']
10 ['tonnerre', 'foudre', '50']
11 ['jet-de-flotte', 'aquatique', '40']
12 ['brûlure', 'flamme', '40']

```

Mais encore mieux, le module offre aussi l'utilitaire `DictReader`. Celui-ci s'utilise de la même manière que `reader`, mais il consomme directement l'en-tête et produit les lignes sous forme de dictionnaires plutôt que de listes (utilisant les valeurs de l'en-tête comme clés).

```

1 >>> with open('attaques.csv') as f:
2 ...     reader = csv.DictReader(f)
3 ...     for row in reader:
4 ...         print(row)
5 ...
6 {'nom': 'charge', 'type': 'normal', 'degats': '20'}
7 {'nom': 'tonnerre', 'type': 'foudre', 'degats': '50'}
8 {'nom': 'jet-de-flotte', 'type': 'aquatique', 'degats': '40'}
9 {'nom': 'brûlure', 'type': 'flamme', 'degats': '40'}

```

VI.7.4.1.2. Écriture

On trouve de manière similaire une fonction `writer` recevant un fichier (ouvert en écriture) pour y écrire des données tabulaires au format CSV. Cette fonction renvoie un objet possédant une méthode `writerow` qui sera appelée pour l'écriture de chaque ligne.

```

1 >>> with open('monstres.csv', 'w') as f:
2     ...     writer = csv.writer(f)
3     ...     writer.writerow(['nom', 'type', 'pv']) # en-tête
4     ...     writer.writerow(['pythachu', 'foudre', '100'])
5     ...     writer.writerow(['ponytha', 'flamme', '150'])
6     ...
7 13
8 21
9 20

```

Chaque appel renvoie le nombre d'octets écrits dans le fichier.

Le code précédent produit donc le fichier suivant.

```

1 >>> with open('monstres.csv', 'w') as f:
2     ...     writer = csv.writer(f)
3     ...     writer.writerow(['nom', 'type', 'pv']) # en-tête
4     ...     writer.writerow(['pythachu', 'foudre', '100'])
5     ...     writer.writerow(['ponytha', 'flamme', '150'])
6     ...
7 13
8 21
9 20

```

Listing 35 – monstres.csv

On notera que l'objet possède aussi une méthode `writerows` pour écrire plusieurs lignes en une fois (en prenant en argument une liste de lignes).

De même, le module propose aussi `DictWriter` pour écrire des lignes depuis un dictionnaire. Le `DictWriter` doit être appelé avec en arguments le fichier de sortie mais aussi la ligne d'en-tête, qui servira à extraire les bonnes valeurs des dictionnaires. La ligne d'en-tête en elle-même sera écrite en appelant la méthode `writeheader` de l'objet.

Ainsi, notre code précédent est équivalent à :

```

1 with open('monstres.csv', 'w') as f:
2     writer = csv.DictWriter(f, ['nom', 'type', 'pv'])
3     writer.writeheader()
4     writer.writerow({'nom': 'pythachu', 'type': 'foudre', 'pv':
5         '100'})
6     writer.writerow({'nom': 'ponytha', 'type': 'flamme', 'pv':
7         '150'})

```

VI.7.4.1.3. Dialectes

Une particularité du CSV est de supporter plusieurs dialectes, car différents outils apportent au format leurs propres spécifications. `,` n'est pas toujours le séparateur de colonnes par exemple. Le dialecte définit aussi quels caractères d'échappement utiliser dans différents contextes.

Ainsi, toutes les fonctions que nous avons vu acceptent un argument nommé `dialect` qui

permet de choisir le dialecte à utiliser (il s'agit d'`'excel'` par défaut), ou directement des arguments correspondant aux options à définir (`delimiter`, `quotechar`, `escapechar`, etc.).

VI.7.4.2. Avantages et inconvénients

Le format CSV a l'intérêt d'être interopérable, malgré ses multiples dialectes qui peuvent rendre son utilisation confuse. Il est néanmoins assez lisible et facile d'utilisation.

C'est par contre un format assez limité qui ne permet que de représenter des données tabulaires simples (peu adapté pour formater des données arborescentes) et qui ne permet pas de typer ses valeurs.

VI.7.5. Chaînes de bytes

Pour la suite nous allons quitter les formats textuels et nous intéresser aux formats dits «binaires», qui ne sont donc pas lisibles comme du texte. Et pour cela, nous avons besoin de découvrir un autre type de Python, le type `bytes`.

Ce type représente une chaîne d'octets, les octets étant l'unité de stockage des informations sur un ordinateur, soit des nombres de 8 bits (de 0 à 255 inclus). Un objet `bytes` peut donc être vu comme un tableau de nombres, chaque nombre étant la valeur d'un octet.

On peut d'ailleurs définir un objet `bytes` à partir d'un tel tableau.

```
1 >>> bytes([1, 2, 3])
2 b'\x01\x02\x03'
```

La représentation de notre objet peut sembler perturbante, mais il s'agit bien de notre tableau.

```
1 >>> data = bytes([1, 2, 3])
2 >>> data[0]
3 1
```

Comme les chaînes de caractères, les chaînes d'octets sont immutables.

```
1 >>> data[0] = 10
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: 'bytes' object does not support item assignment
```

Les deux types sont d'ailleurs assez semblables, ils étaient même confondus en Python 2, les deux identifiant des chaînes.

Les caractères ne sont qu'une abstraction pour interpréter des octets comme du texte, et une chaîne de caractères est ainsi une chaîne d'octets munie d'une règle définissant comment interpréter les octets en caractères. Cette règle est appelée un encodage, mais j'y reviendrai ensuite.

Cette similitude entre les deux s'appuie entre autres sur la table ASCII qui établit une correspondance entre certains caractères (notamment les caractères alphanumériques latins «de base»—sans accents—et les chiffres, ainsi que des caractères de contrôle) et des octets, elle sert

encore aujourd'hui de base à de nombreux encodages.

	0	1	2	3	4	5	6	7	0
0	N	U	D	L	,	'	'	'	'
1	S	C	D	C	!	'	1	'	A
2	S	T	D	C	'	'	'	2	'
3	E	T	D	C	'	#	'	3	'
4	E	C	D	C	'	\$	'	4	'
5	E	N	N	A	'	%	'	5	'
6	A	C	S	Y	'	&	'	6	'
7	B	E	E	T	'	'	'	7	'
8	B	S	C	A	'	'	'	8	'
9	H	T	E	M	'	'	'	9	'
A	L	F	S	U	'	'	'	:	'
B	V	T	E	S	'	'	'	;	'
C	F	F	F	S	'	'	'	<	'
D	C	R	G	S	'	'	'	=	'
E	S	C	R	S	'	'	'	.	'
F	S	I	U	S	'	'	'	/	'

TABLE VI.7.2. – Table ASCII

C'est pourquoi, lors de l'affichage, Python essaie généralement de représenter un objet *bytes* comme du texte, en s'appuyant sur la table ASCII.

```
1 >>> bytes([65, 66, 67])
2 b'ABC'
```

65, 66 et 67 sont les valeurs ASCII des caractères **A**, **B** et **C** (ou **0x41**, **0x42** et **0x43** en hexadécimal).

On le voit ainsi, une chaîne d'octets peut simplement se définir comme une chaîne de caractères préfixée d'un **b**.

```
1 >>> b'foobar'
2 b'foobar'
```

Cela ne change rien au fait que la chaîne ainsi créée est toujours considérée comme un tableau de nombres.

```
1 >>> b'foobar'[0]
2 102
```

Bien sûr, seulement les caractères de la table ASCII sont utilisables pour construire une chaîne d'octet, impossible d'y utiliser des caractères spéciaux qui n'ont aucune correspondance.

```
1 >>> b'été'
2 File "<stdin>", line 1
3 SyntaxError: bytes can only contain ASCII literal characters.
```

Et comme on l'a vu plus haut, on peut utiliser la notation `\xNN` pour insérer des octets particuliers, `NN` étant la valeur de l'octet en hexadécimal.

```
1 >>> data = b'\x01\x2A\x61'
2 >>> data[1]
3 42
4 >>> hex(data[1])
5 '0x2a'
6 >>> hex(data[2])
7 '0x61'
```

Les octets pouvant être interprétés comme des caractères sont affichés comme tel par Python pour faciliter la lisibilité.

```
1 >>> data
2 b'\x01*a'
```

Qui dit similitude avec les chaînes de caractères dit aussi opérations similaires. Ainsi il est possible de concaténer des chaînes d'octets et d'y appliquer pratiquement les mêmes méthodes.

```
1 >>> b'abc' + b'def'
2 b'abcdef'
3 >>> b'foo'.replace(b'o', b'e')
4 b'fee'
5 >>> b'a;b;c'.split(b';')
6 [b'a', b'b', b'c']
```

Mais les deux types ne sont pas compatibles entre-eux.

```
1 >>> b'abc' + 'def'
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: can't concat str to bytes
```

VI.7.5.1. Encodages

Il est en revanche possible de convertir l'un vers l'autre. Les chaînes de caractères possèdent une méthode `encode` renvoyant une chaîne d'octets.

```
1 >>> 'foobar'.encode()
2 b'foobar'
```

À l'inverse, les chaînes d'octets ont une méthode `decode` pour les convertir en chaînes de caractères.

```
1 >>> b'foobar'.decode()
2 'foobar'
```

Je n'utilise ici que des caractères de la table ASCII, mais cela fonctionne aussi avec des caractères «spéciaux».

```
1 >>> 'été'.encode()
2 b'\xc3\xa9t\xc3\xa9'
3 >>> b'\xc3\xa9t\xc3\xa9'.decode()
4 'été'
```

Comment cela fonctionne ? Avec la notion d'encodage dont je parlais plus haut. Un encodage c'est une table qui fait la correspondance entre des caractères et des octets, associant un ou plusieurs octets à un caractère. La table ASCII est un encodage (mais avec un ensemble limité de caractères).

En Python, on utilise plus couramment des encodages unicode—qui peuvent représenter tous les caractères existant—et plus particulièrement UTF-8. C'est cet encodage UTF-8 qui a été utilisé par défaut lors des opérations précédentes. En effet, les méthodes `encode` et `decode` peuvent prendre un argument optionnel pour spécifier l'encodage vers lequel encode / depuis lequel décoder.

```
1 >>> 'été'.encode('utf-8')
2 b'\xc3\xa9t\xc3\xa9'
```

On notera que la taille varie entre chaînes de caractères et chaînes d'octets, l'appel à `len` nous renverra 3 dans le premier cas et 5 dans le second. C'est bien parce que l'on compte soit les caractères soit les octets.

```
1 >>> len('été')
2 3
3 >>> len('été'.encode('utf-8'))
4 5
```

D'autres encodages existent et ils ont chacun leurs particularités. Par exemple l'UTF-32 est un encodage unicode qui représente chaque caractère sur 4 octets.

```

1 >>> 'été'.encode('utf-32')
2 b'\xff\xfe\x00\x00\xe9\x00\x00\x00t\x00\x00\x00\xe9\x00\x00\x00'
3 >>> 'abc'.encode('utf-32')
4 b'\xff\xfe\x00\x00a\x00\x00\x00b\x00\x00\x00c\x00\x00\x00'

```

Ou encore l'encodage latin-1 (ou iso-8859-1) un encodage encore parfois utilisé sur certains systèmes en Europe (Windows notamment).

```

1 >>> 'été'.encode('latin-1')
2 b'\xe9t\xe9'

```

Mais latin-1 n'est pas un encodage unicode et ne pourra donc pas représenter tous les caractères.

```

1 >>> '♪'.encode('utf-8')
2 b'\xe2\x99\xab'
3 >>> '♪'.encode('latin-1')
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 UnicodeEncodeError: 'latin-1' codec can't encode character
  '\u266b' in position 0: ordinal not in range(256)

```

Une chaîne ayant été encodée avec un certain encodage doit toujours être décodé avec ce même encodage, cela donnerait sinon lieu à des erreurs ou des incohérences.

```

1 >>> 'été'.encode('utf-8').decode('latin-1')
2 'Ã©tÃ©'
3 >>> 'été'.encode('latin-1').decode('utf-8')
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in
  position 0: invalid continuation byte

```

On notera aussi que l'ascii est reconnu comme un encodage à part entière par les méthodes `encode` et `decode`. Bien sûr, seuls les caractères de la table ASCII sont autorisés dans les chaînes.

```

1 >>> 'abcdef'.encode('ascii')
2 b'abcdef'
3 >>> b'abcdef'.decode('ascii')
4 'abcdef'

```

Les encodages interviennent quand vous traitez des données extérieures au programme, et notamment des fichiers. Ainsi, la fonction `open` dispose d'un paramètre `encoding` pour préciser l'encodage du fichier à ouvrir.

```

1 with open('output.txt', 'w', encoding='latin-1') as f:
2     f.write('été')
```

Gardez donc en tête qu'un fichier texte (ou même n'importe quel texte) est toujours lié à un encodage, et que celui-ci n'est pas toujours UTF-8. Souvent l'encodage sera renseigné comme métadonnée avec le fichier, comme c'est le cas en HTTP avec l'en-tête `Content-Type` qui précise l'encodage des données.

VI.7.5.2. Mode binaire

Mais tous les fichiers ne représentent pas du texte, même sous des encodages particuliers, les images par exemple. Ainsi, on voudrait parfois pouvoir traiter un fichier comme des données brutes, comme des octets.

Cela est possible à l'aide du mode binaire, il s'agit d'un caractère `b` ajouté au mode d'ouverture du fichier. Ce mode aura pour effet que toutes les opérations sur le fichier traiteront des chaînes d'octets et non des chaînes de caractères.

```

1 >>> with open('output.txt', 'rb') as f:
2     ...     f.read()
3     ...
4 b'\xe9t\xe9'
```

Il en est de même en écriture, où les méthodes attendront des chaînes d'octets.

```

1 >>> with open('output.txt', 'wb') as f:
2     ...     f.write(b'\x01\x02\x03')
3     ...
4 3
```

Ce mode nous sera utile pour maintenant aborder un autre format de sérialisation des données, un format binaire.

VI.7.6. Sérialisation binaire

Python possède un format de sérialisation qui lui est propre, disponible via le module `pickle`, capable de gérer à peu près tous les types Python. C'est donc un format très pratique pour enregistrer l'état d'un programme.

Étant un format binaire, je ne vais pas décrire à quoi il ressemble, ça serait juste un tas d'octets illisibles. Sachez juste que le format gère de nombreux objets Python, pour peu que leur type soit connu par le programme qui chargera les données, en inspectant ce qui est contenu dans ces objets.

VI.7.6.1. Module `pickle`

Le module `pickle` [utilise](#) l'interface dont je vous avais parlé plus tôt pour `json`, et propose donc les fonctions `load`, `loads`, `dump` et `dumps`.

VI.7.6.1.1. Écriture

Puisqu'il nous est impossible de partir d'un fichier existant, nous débuterons cette fois par l'écriture. Elle se fait donc à l'aide des fonctions `dump` et `dumps`.

`dump` prend en argument un objet Python et un fichier (ouvert en écriture) vers lequel le sérialiser.

Il est possible d'enchaîner les appels à `dump` pour écrire plusieurs objets dans le fichier.

```

1 with open('game.dat', 'wb') as f:
2     monsters = {
3         '001': {
4             'name': 'Pythachu',
5             'attaques': ['charge', 'tonnerre'],
6         },
7         '002': {
8             'name': 'Pythard',
9             'attaques': ['charge', 'jet-de-flotte'],
10        },
11    }
12    pickle.dump(monsters, f)
13    attacks = [
14        {'name': 'charge', 'type': 'normal', 'damage': 20},
15        {'name': 'tonnerre', 'type': 'foudre', 'damage': 50},
16        {'name': 'jet-de-flotte', 'type': 'aquatique', 'damage':
17            50},
18    ]
19    pickle.dump(attacks, f)

```

La méthode `dumps` prend simplement un objet et renvoie une chaîne d'octets qui sera compatible avec `loads`.

```

1 >>> pickle.dumps([1, 2, 3])
2 b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.'
3 >>> pickle.dumps(2+1j)
4 b'\x80\x04\x95.\x00\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x07complex\x94\x93\x94G@\x00\x00\x00\x00\x00\x00\x00G?\xf0\x00\x00\x00\x00\x00\x00\x86\x94R\x94.'

```

VI.7.6.1.2. Lecture

La méthode `loads` prend donc en argument une chaîne d'octets, reconstruit l'objet Python représenté et le renvoie.

```

1 >>> pickle.loads(b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.')
2 [1, 2, 3]

```

```

3 >>> pickle.loads(b'\x80\x04\x95.\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x07complex\x94\x93\x94G@\x00\x00\x00\x00\x00\x00\x00?\xf0\x00\x00\x00\x00\x00\x00\x86\x94R\x94.')
```

`load` prend elle un fichier et renvoie aussi l'objet qui y est contenu. Comme pour `dump`, il est possible d'appeler `load` plusieurs fois de suite sur un même fichier (pour y lire les différents objets écrits).

```

1 >>> with open('game.dat', 'rb') as f:
2     ...     print('monstres :', pickle.load(f))
3     ...     print('attaques :', pickle.load(f))
4     ...
5 monstres : {'001': {'name': 'Pythachu', 'attaques': ['charge',
6     'tonnerre']}, '002': {'name': 'Pythard', 'attaques':
7     ['charge', 'jet-de-flotte']}}
8 attaques : [{'name': 'charge', 'type': 'normal', 'damage': 20},
9     {'name': 'tonnerre', 'type': 'foudre', 'damage': 50}, {'name':
10    'jet-de-flotte', 'type': 'aquatique', 'damage': 50}]
```

VI.7.6.2. Avantages et inconvénients

Vous l'aurez compris, `pickle` est un format très pratique en Python, puisqu'il permet de tout représenter ou presque. Il n'est en revanche pas interopérable puisque applicable seulement à Python.



Attention aussi, ce format permet l'exécution de code arbitraire ce qui présente donc une grosse faille de sécurité sur des données non sûres, il est donc à bannir pour tout ce qui reçoit des données distantes sans couche supplémentaire de sécurité.

Dans notre cas d'une sauvegarde de l'état d'un programme, c'est un assez bon choix.

VI.7.7. Autres formats

Nous avons fait un tour des modules disponibles dans la bibliothèque standard de Python, mais ce ne sont pas les seuls formats existant. Pour les autres, il faudra en revanche s'appuyer sur des modules tiers, nous verrons par la suite comment en installer.

Voici donc quelques autres formats que vous pourriez croiser et qui se prêtent à diverses utilisations.

VI.7.7.0.1. toml

Le format `toml` est un format simple adapté à des fichiers de configuration, dérivé du format `INI`.

Il permet ainsi de représenter des couples clé/valeur regroupés en sections mais ajoute la gestion des types des valeurs.

Le type d'une valeur sera alors dépendant de la syntaxe utilisée pour la définir, de la même manière que le fait le format JSON. Ainsi dans l'exemple suivant `width` et `height` seront des valeurs de type `int` alors que `save` sera une chaîne de caractères.

```
1 >>> with open('game.dat', 'rb') as f:
2 ...     print('monstres :', pickle.load(f))
3 ...     print('attaques :', pickle.load(f))
4 ...
5 monstres : {'001': {'name': 'Pythachu', 'attaques': ['charge',
6             'tonnerre']}, '002': {'name': 'Pythard', 'attaques':
            ['charge', 'jet-de-flotte']}}
7 attaques : [{'name': 'charge', 'type': 'normal', 'damage': 20},
            {'name': 'tonnerre', 'type': 'foudre', 'damage': 50}, {'name':
            'jet-de-flotte', 'type': 'aquatique', 'damage': 50}]
```

Listing 36 – config.toml

— Page de la bibliothèque `toml` en Python : <https://github.com/uiri/toml> ↗

VI.7.7.0.2. yaml

YAML est un format de données riches, semblable à *JSON* mais plus axé sur la lisibilité. Il permet de décrire de manière claire des données complexes.

```
1 id: 001
2 name: Pythachu
3 type: foudre
4 attaques:
5   - tonnerre
6   - charge
7 base_pv: 50
```

Listing 37 – pythachu.yaml

— Page de la bibliothèque `PyYAML` en Python : <https://pyyaml.org/wiki/PyYAML> ↗

VI.7.7.0.3. msgpack

msgpack est lui aussi un format de données assez semblable à *JSON*, à l'exception près que c'est un format binaire. Il permet donc de manière compacte de représenter nombres, chaînes de caractères, listes et dictionnaires.

C'est un format interopérable qui possède des bibliothèques pour à peu près tous les langages.

— Page du projet `msgpack` : <https://msgpack.org/> ↗

VI.7.7.0.4. Protobuf

Protobuf est un format plus complexe destiné à établir des protocoles de communication entre programmes. Les programmes doivent donc utiliser un protocole commun qui définit les types des données transmises dans un message.

VI. Entrées / sorties

Cela permet d'omettre les informations de typage dans la sérialisation, et d'avoir une assurance de la validité des données transmises.

— Page du projet `protobuf` : <https://developers.google.com/protocol-buffers> ↗

VI.8. Arguments de la ligne de commande

VI.8.1. Ligne de commande

Pour l'instant nous appelons nos programmes depuis la ligne de commande en tapant `python program.py` (mais nous savons aussi [comment utiliser `./program.py` sous Linux](#)).

Dans les deux cas, cela fait appel à l'interpréteur Python en lui donnant le chemin de notre programme en argument. Mais il est possible de renseigner d'autres arguments lors du lancement et ceux-ci seront transmis à notre programme.

Ils seront accessibles sous la forme d'une liste de chaînes de caractères, la liste `argv` qu'il faudra importer depuis le module `sys` (un module qui gère les informations sur le système).

```
1 import sys
2
3 print(sys.argv)
```

Listing 38 – program.py

À l'utilisation, nous recevons bien les différents arguments passés au programme.

```
1 % python program.py
2 ['program.py']
3 % python program.py foo bar
4 ['program.py', 'foo', 'bar']
5 % python program.py 1 2 3
6 ['program.py', '1', '2', '3']
```

On voit que le premier argument est toujours le nom du programme.

Comme indiqué, il ne s'agit que de chaînes de caractères et il va donc falloir convertir les types lorsque cela est nécessaire. Par exemple avec cette mini-calculatrice.

```
1 import sys
2
3 a = int(sys.argv[1])
4 b = int(sys.argv[2])
5 print(a + b)
```

Listing 39 – addition.py

```
1 % python addition.py 3 5
2 8
3 % python addition.py 10 -3
4 7
```

Mais attention, notre code plantera méchamment si nous ne fournissons pas suffisamment d'arguments.

```
1 % python addition.py 1
2 Traceback (most recent call last):
3   File "addition.py", line 4, in <module>
4     b = int(sys.argv[2])
5 IndexError: list index out of range
```

En effet, `sys.argv` est une liste ordinaire, et si sa taille n'est que de 2, alors elle ne possède pas d'élément à l'index 2.

Pour nous prémunir de ce genre d'erreurs, il faut donc vérifier la taille de la liste avant d'accéder à ses éléments. Et généralement dans ces cas là, on quittera le programme en affichant un message expliquant comment l'appeler.

Pour quitter un programme à tout moment, on peut faire appel à la fonction `sys.exit`.

```
1 import sys
2
3 if len(sys.argv) < 3:
4     print('Usage: addition.py nb1 nb2')
5     sys.exit()
6
7 a = int(sys.argv[1])
8 b = int(sys.argv[2])
9 print(a + b)
```

À l'utilisation c'est tout de suite plus propre.

```
1 % python addition.py 1 2
2 3
3 % python addition.py 1
4 Usage: addition.py nb1 nb2
```

Pour plus de genericité, on pourrait écrire `print(f'Usage: {sys.argv[0]} nb1 nb2')` évitant d'inscrire en dur le nom du programme. Le premier élément de `sys.argv` sera toujours présent, notre programme n'aurait pas pu être appelé sinon.

VI.8.1.1. Sortie standard et sortie d'erreur

Il y a un seul soucis avec notre message d'erreur : celui-ci est imprimé sur la sortie standard.

?

Qu'est-ce que la sortie standard ?

Sur Linux, les programmes qui tournent sont automatiquement reliés à 3 périphériques :

- L'entrée standard, celle qui récupère le texte entré sur le terminal, accessible via `input()`.
- La sortie standard, où est affiché par défaut tout ce qui sort du programme sur le terminal (avec `print` par exemple).
- La sortie d'erreur, spécifiquement dédiée aux erreurs.

Les sorties standard et d'erreur sont toutes deux affichées par défaut sur le terminal, mais elles sont pourtant différentes. Dans un shell Bash, il est possible d'utiliser l'opérateur `>` pour rediriger le flux de sortie standard vers un fichier, et l'opérateur `2>` pour la sortie d'erreur.

```
1 % python addition.py > out 2> err
```

Si l'on inspecte nos fichiers, on constate bien que `out` contient le message d'erreur et que `err` est vide. On aimerait que ce soit l'inverse en cas d'erreur.

En fait, chaque sortie correspond à un fichier ouvert par défaut par le programme. Pour la sortie standard, il s'agit de `sys.stdout`. (*standard output*). On peut l'utiliser comme tout autre fichier ouvert en écriture.

```
1 >>> import sys
2 >>> sys.stdout.write('hello\n')
3 hello
4 6
5 >>> print('world', file=sys.stdout)
6 world
```

Le 6 qui apparaît n'est que le retour de l'appel à `write` (6 caractères ont été écrits). Et de façon similaire, on a `sys.stderr` qui correspond à la sortie d'erreur.

```
1 >>> print('error', file=sys.stderr)
2 error
```

Bien sûr la différence n'est pas flagrante dans cet exemple, elle le sera si l'on redirige les sorties standard et d'erreur vers des fichiers différents. Ce qui est généralement fait pour la journalisation d'un programme.

```
1 import sys
2
3 print('standard output', file=sys.stdout)
4 print('error output', file=sys.stderr)
```

Listing 40 – outputs.py

```

1 % python outputs.py > out 2> err
2 % cat out
3 standard output
4 % cat err
5 error output

```

Ainsi, nous pouvons remplacer notre code de traitement d'erreur par le suivant.

```

1 if len(sys.argv) < 3:
2     print(f'Usage: {sys.argv[0]} nb1 nb2', file=sys.stderr)
3     sys.exit()

```

VI.8.2. Parseur d'arguments

Manipuler `sys.argv` ça va quand on a des arguments simples comme deux nombres ici, mais ça devient vite compliqué pour gérer les options passées à un programme.

En effet, il serait difficile de gérer manuellement les arguments d'un appel tel que `python cmd.py -v -f pdf --foo=bar --foo2 42 photo.jpg`. C'est pourquoi des outils existent pour analyser à votre place les arguments, les valider en fonction de ce qui est attendu, et les classer convenablement.

Le module `argparse` de la bibliothèque standard propose l'un de ces outils.

`argparse` fournit un type `ArgumentParser` que l'on peut instancier pour obtenir un parseur d'arguments.

```

1 import argparse
2
3 parser = argparse.ArgumentParser()

```

Listing 41 – `cmd.py`

Des méthodes sont ensuite disponibles sur ce parseur pour le personnaliser et préciser les arguments que l'on attend.

Notamment la méthode `add_argument` qui permet de demander à gérer un nouvel argument. Celle-ci accepte de nombreuses options que je ne détaillerai pas ici. Sachez simplement qu'elle attend en premier le nom voulu pour l'argument.

- Si ce nom est de type `-x` alors elle gèrera l'argument comme `-x VALEUR` lors de l'appel au programme.
- S'il est de type `--abc`, elle gèrera `--abc=VALEUR` et `--abc VALEUR`.
- S'il ne débute pas par un tiret, alors il s'agira d'un argument positionnel du programme.

Un paramètre `action` sert à préciser quoi faire de l'argument rencontré.

- `store_const` permet de simplement stocker la valeur associée à l'argument.
- `store_true` permet de stocker `True` si l'argument est présent et `False` sinon.

Une valeur par défaut pour l'argument peut être renseignée avec le paramètre `default`.

Par défaut, la valeur de l'argument sera stockée dans l'objet résultant sous le nom de l'argument. Il est cependant possible de choisir un autre nom pour le stockage à l'aide du paramètre `dest`.

Enfin, il est possible d'utiliser le paramètre `type` pour convertir automatiquement la valeur d'un argument vers le type voulu.

Voilà par exemple comment nous pourrions traiter les arguments présentés pour la commande plus haut.

```
1 parser.add_argument('-v', dest='verbose', action='store_true')
2 parser.add_argument('-f', dest='format', default='text')
3 parser.add_argument('--foo')
4 parser.add_argument('--foo2', type=int)
5 parser.add_argument('file')
6
7 args = parser.parse_args()
8 print(args)
9 print('Verbose:', args.verbose)
10 print('Format:', args.format)
```

Listing 42 – cmd.py

```
1 % python cmd.py -v -f pdf --foo=bar --foo2 42 photo.jpg
2 Namespace(verbose=True, format='pdf', foo='bar', foo2=42,
3           file='photo.jpg')
4 Verbose: True
4 Format: pdf
5 % python cmd.py doc.odt
6 Namespace(verbose=False, format='text', foo=None, foo2=None,
7           file='doc.odt')
7 Verbose: False
8 Format: text
```

Pour plus d'informations au sujet du module `argparse`, vous pouvez consulter sa page de documentation : <https://docs.python.org/fr/3/library/argparse.html> .

VI.9. Les paquets

Introduction

Les paquets (ou *packages*) forment une entité hiérarchique au-dessus des modules : un paquet est un module qui contient d'autres modules, un peu comme un dossier contient des fichiers. D'ailleurs, les paquets prennent généralement la forme de dossiers sur le système de fichiers. On en a déjà rencontré pendant ce cours : souvenez-vous du module `xml.etree.ElementTree` : il s'agissait en fait d'un module `ElementTree` dans un paquet `xml.etree`. On comprend par la même occasion qu'`etree` est lui-même imbriqué dans un paquet `xml`, car les paquets sont hiérarchiques.

VI.9.1. Construction d'un paquet

Pour créer notre propre paquet, on peut alors simplement créer un nouveau répertoire dans lequel on placera nos fichiers Python (nos modules).

Créons par exemple un dossier `operations` depuis le répertoire courant, avec deux fichiers `addition.py` et `soustraction.py` :

```
1 def addition(a, b):  
2     return a + b
```

Listing 43 – `operations/addition.py`

```
1 def soustraction(a, b):  
2     return a - b
```

Listing 44 – `operations/soustraction.py`

Nous voilà maintenant avec un nouveau paquet `operations`. Ce paquet forme un espace de noms supplémentaire autour de nos modules, et nous devons donc les préfixer de `operations.` pour les importer.

```
1 >>> from operations import addition  
2 >>> # on a importé le module addition  
3 >>> addition.addition(1, 2)  
4 3  
5 >>> from operations.soustraction import soustraction  
6 >>> # on a importé directement la fonction soustraction  
7 >>> soustraction(1, 2)
```


8	-1
---	----



Pensez à bien vous placer depuis le répertoire contenant le dossier `operations` et non dans le dossier `operations` lui-même pour exécuter ce code.

Par exemple si votre dossier `operations` se trouve dans un dossier `projet`, il faut que vous exécutiez l'interpréteur depuis ce dossier `projet` sans quoi Python ne serait pas en mesure de trouver le paquet.

On remarque que l'on ne peut pas simplement faire `import operations` puis utiliser par exemple `operations.addition.addition(1, 2)` comme on le ferait avec des modules. C'est parce que les modules d'un paquet ne sont pas directement chargés quand le paquet est importé, mais nous verrons ensuite comment y parvenir.

VI.9.1.1. Imports relatifs

Il peut arriver depuis un paquet que nous ayons besoin d'accéder à d'autres modules du même paquet. Par exemple, notre fonction `soustraction` pourrait vouloir faire appel à la fonction `addition`.

Pour cela, on va pouvoir importer la fonction `addition` dans le module `soustraction`, comme on vient de le faire dans l'interpréteur interactif.

```
1 from operations.addition import addition
2
3 def soustraction(a, b):
4     return addition(a, -b)
```

Listing 45 – `operations/soustraction.py`

Et tout fonctionne comme prévu :

```
1 >>> from operations.soustraction import soustraction
2 >>> soustraction(8, 5)
3 3
```

Mais la syntaxe peut paraître lourde, pourquoi avoir besoin de préciser `operations` alors qu'on est déjà dans ce paquet ? Python a prévu une réponse à ça : les imports relatifs.

Ainsi, pour un import au sein d'un même paquet, on peut simplement référencer un autre module en le préfixant d'un `.` sans indiquer explicitement le paquet (qui sera donc le paquet courant).

```
1 from .addition import addition
2
3 def soustraction(a, b):
4     return addition(a, -b)
```

Listing 46 – `operations/soustraction.py`

Et l'on peut vérifier en important le module `operations.soustraction` que tout fonctionne toujours correctement.



Attention cependant, cette syntaxe d'imports relatifs n'est valable que dans le cas d'un `from ... import ...`. Il n'est ainsi pas possible d'écrire simplement `import .addition` pour importer le module `addition`.

En revanche la syntaxe `from . import addition` est valide (équivalente à `from operations import addition`).

VI.9.2. Fichier `__init__.py`

Comme je le disais précédemment, le code des modules n'est pas directement chargé quand on importe le paquet. Qu'est-ce qui se passe alors quand on fait un `import operations` ?

À notre niveau pas grand chose en fait. Python identifie où se trouvent les fichiers du paquet `operations` et instancie un module vide qu'il nous renvoie.

Mais dans les faits, il cherche un fichier `__init__.py` à l'intérieur du paquet pour l'exécuter. C'est en fait ce fichier qui contient le code du paquet à proprement parler : tout ce qui sera présent dedans sera exécuté lors d'un `import operations`.

```
1 print('Hello')
```

Listing 47 – `operations/__init__.py`

```
1 >>> import operations
2 Hello
```



Attention au nommage du fichier, il faut bien deux *underscores* de part et d'autre de `init`.

Bien sûr cet exemple n'est pas très utile, mais ce fichier `__init__.py` peut aussi nous servir à charger directement le code des modules du paquet.

Par exemple on peut y importer nos fonctions `addition` et `soustraction` pour les rendre accessibles plus facilement.

```
1 from .addition import addition
2 from .soustraction import soustraction
```

Listing 48 – `operations/__init__.py`

```
1 >>> import operations
2 >>> operations.addition(3, 5)
3 8
```

```

4 >>> from operations import soustraction
5 >>> soustraction(8, 5)
6 3

```

i

Avant Python 3.3, le fichier `__init__.py` était nécessaire pour que Python considère le répertoire comme un paquet. Ce n'est plus le cas aujourd'hui mais ce fichier reste toutefois utile pour indiquer à Python que tout le code du paquet se trouve dans ce même répertoire. Prenez ainsi l'habitude de toujours avoir un fichier `__init__.py` (même vide) dans vos paquets, cela pourrait vous éviter certaines déconvenues.

VI.9.3. Fichier `main.py`

Il existe un autre fichier «magique» au sein des paquets, le fichier `__main__.py`. Mais avant d'y revenir, je dois vous parler de l'option `-m` de l'interpréteur Python.

C'est une option qui permet de demander à Python d'exécuter un module à partir de son nom. Cela permet de ne pas avoir à connaître le chemin complet vers le fichier du module pour le lancer. Et certains modules Python s'en servent pour mettre à disposition des petits programmes.

Par exemple le module `turtle` propose une démo si on l'exécute via `python -m turtle` :

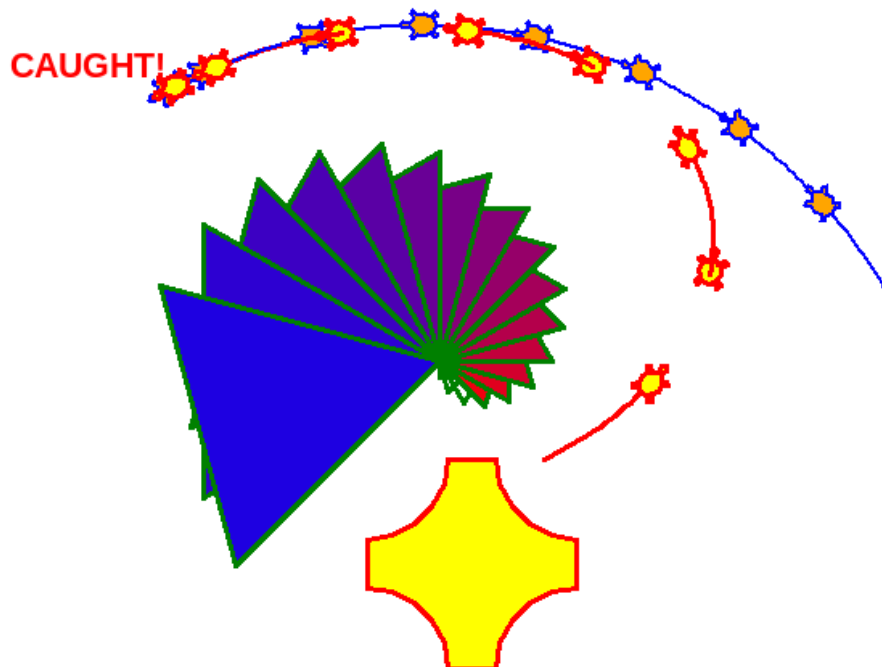


FIGURE VI.9.1. – `python -m turtle`

Cela fonctionne aussi avec nos propres modules.

```
1 def hello():
2     print('Hello World!')
3
4 if __name__ == '__main__':
5     hello()
```

Listing 49 – hello.py

```
1 % python -m hello
2 Hello World!
```



Pour rappel, le bloc conditionnel `if __name__ == '__main__':` permet de placer du code qui sera exécuté uniquement quand le module est lancé directement (`python hello.py` ou `python -m hello`) mais pas quand le module est importé.

Dans le cadre de notre paquet, `python -m operations` cherchera à exécuter son fichier `__main__.py`. Nous pouvons alors y créer un tel fichier pour nous aussi faire une démonstration de notre paquet.

```
1 from .addition import addition
2 from .soustraction import soustraction
3
4 if __name__ == '__main__':
5     print('Addition: 3+5 =', addition(3, 5))
6     print('Soustraction: 8-3 =', soustraction(8, 3))
```

Listing 50 – operations/__main__.py

```
1 % python -m operations
2 Addition: 3+5 = 8
3 Soustraction: 8-3 = 5
```

Conclusion

En conclusion, je vous invite à consulter mon billet [Notes sur les modules et packages en Python](#) qui répond à plusieurs problématiques au sujet des paquets et des imports en Python.

VI.10. TP : Sauvegarder la partie

VI.10.1. Découpage en modules

Une première étape dans l'avancement de notre TP va être de le découper en modules. En effet, nous l'avons précédemment découpé en fonctions, mais ces fonctions cohabitent toutes ensemble dans un joyeux bordel.

Nous pouvons donc aller plus loin et réunir ces fonctions en unités logiques dans un paquet.

Je vous propose pour cela un paquet `tp` contenant des modules pour les différentes parties de notre jeu : définitions des données, gestion des joueurs, gestion des entrées utilisateur. On gardera aussi un module `game` pour les fonctions principales du jeu.

VI.10.1.1. Solution

Un découpage possible est le suivant.

J'ai utilisé un module `definitions` pour stocker les dictionnaires `monsters` et `attacks`. Ce module est assimilable à une base de données, il n'y a que lui à faire évoluer pour ajouter de nouveaux monstres ou de nouvelles attaques.

En plus de ça, un module `prompt` reprend la fonction `get_choice_input` et un module `get_players` est dédié à l'instanciation des joueurs.

J'ai aussi ajouté un fichier `__main__.py` pour exécuter notre TP avec `python -m tp`.

👁 Contenu masqué n°14

i

Comme vous le voyez, j'ai ici laissé un fichier `__init__.py` vide car il ne nous est pas utile.

VI.10.2. Sauvegarde

Venons-en maintenant à l'objectif de ce TP : la sauvegarde du jeu.

On pourra pour cela ajouter un fichier `save.py` à notre paquet contenant une fonction `load_game(filename)` pour charger une sauvegarde depuis un fichier, renvoyant les deux joueurs ainsi chargés, et une fonction `save_game(filename, player1, player2)` pour enregistrer la sauvegarde (l'état des deux joueurs) dans un fichier.

Dans la boucle principale de notre jeu, on ajoutera donc une question pour demander à l'utilisateur s'il souhaite continuer ou s'arrêter. En cas d'arrêt, on lui demandera alors s'il souhaite sauvegarder la partie et dans quel fichier.

```
1 % python -m tp
2 Monstres disponibles :
3 - Pythachu
4 - Pythard
5 - Ponytha
6 Joueur 1 quel monstre choisissez-vous ?
7 > Pythachu
8 Quel est son nombre de PV ? 100
9 Joueur 2 quel monstre choisissez-vous ?
10 > Ponytha
11 Quel est son nombre de PV ? 120
12
13 Pythachu affronte Ponytha
14
15 Voulez-vous continuer ? [0/n] o
16 Joueur 1 quelle attaque utilisez-vous ?
17 - Tonnerre -50 PV
18 - Charge -20 PV
19 > tonnerre
20 Pythachu attaque Ponytha qui perd 50 PV, il lui en reste 70
21 Joueur 2 quelle attaque utilisez-vous ?
22 - Brûlure -40 PV
23 - Charge -20 PV
24 > brûlure
25 Ponytha attaque Pythachu qui perd 40 PV, il lui en reste 60
26 Voulez-vous continuer ? [0/n] n
27 Voulez-vous sauvegarder ? [o/N] o
28 Dans quel fichier sauvegarder ? game.dat
```

Dans l'autre sens, on permettra au programme de prendre un argument pour charger le jeu depuis la sauvegarde pointée par ce fichier.

```
1 % python -m tp game.dat
2 Pythachu affronte Ponytha
3
4 Voulez-vous continuer ? [0/n]
5 Joueur 1 quelle attaque utilisez-vous ?
6 - Tonnerre -50 PV
7 - Charge -20 PV
8 > tonnerre
9 Pythachu attaque Ponytha qui perd 50 PV, il lui en reste 20
10 [...]
```

On privilégiera le format JSON pour le fichier de sauvegarde.

VI.10.2.1. Solution

Voici maintenant la solution à cet exercice, qui repose principalement sur le fichier `tp/save.py`, dont les fonctions sont appelées dans le module `game`.

On peut voir aussi la fonction `get_yesno_input` dans le module `prompt`. C'est une fonction qui permet de poser une question qui attend pour réponse oui ou non (`[0/n]`). Elle propose aussi de définir une valeur par défaut, reconnaissable à la lettre en majuscule (`0` ici pour `Oui`). Ainsi si l'utilisateur entre une ligne vide, c'est cette valeur par défaut qui sera utilisée.

👁 Contenu masqué n°15

VI.10.3. Tests

Maintenant que l'on a un paquet dédié à notre TP, il va être plus simple de le tester depuis l'extérieur. On va pouvoir déplacer nos tests dans un fichier `test_tp.py` à l'extérieur du paquet. Depuis ce fichier, on importera les différentes fonctions que l'on souhaite tester.

À notre module, on va ajouter des fonctions pour tester nos fonctions de sauvegarde. Vérifier qu'une sauvegarde se fait correctement vers le fichier et qu'il est possible d'en charger une ensuite.

VI.10.3.1. Solution

Retrouvez maintenant ci-dessous la solution que je propose pour ce TP.

👁 Contenu masqué n°16

i

On remarque qu'après l'exécution de nos tests, un fichier `test_game.dat` persiste dans le répertoire courant. Ce n'est pas très grave pour l'instant mais ce n'est pas très propre non plus.

Il existe une manière d'éviter cela à l'aide du module `tempfile` [↗](#) de la bibliothèque standard pour créer un fichier temporaire.

Contenu masqué

Contenu masqué n°14

1

Listing 51 – `tp/__init__.py`

```

1 from . import game
2
3
4 if __name__ == '__main__':
5     game.main()

```

Listing 52 – tp/__main__.py

```

1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }

```

Listing 53 – tp/definitions.py

```

1 from .definitions import attacks
2 from .players import get_players
3 from .prompt import get_choice_input
4
5
6 def apply_attack(attack, opponent):
7     opponent['pv'] -= attack['damages']
8     if opponent['pv'] < 0:
9         opponent['pv'] = 0
10
11
12 def game_turn(player, opponent):
13     # Si le joueur est KO, il n'attaque pas
14     if player['pv'] <= 0:

```



```

15     return
16
17     print('Joueur', player['id'], 'quelle attaque utilisez-vous ?')
18     for name in player['monster']['attacks']:
19         print('-', name.capitalize(), -attacks[name]['damages'],
20               'PV')
21
22     attack = get_choice_input(attacks, 'Attaque invalide')
23     apply_attack(attack, opponent)
24
25     print(
26         player['monster']['name'],
27         'attaque',
28         opponent['monster']['name'],
29         'qui perd',
30         attack['damages'],
31         'PV, il lui en reste',
32         opponent['pv'],
33     )
34
35 def get_winner(player1, player2):
36     if player1['pv'] > player2['pv']:
37         return player1
38     else:
39         return player2
40
41
42 def main():
43     player1, player2 = get_players()
44
45     print()
46     print(player1['monster']['name'], 'affronte',
47           player2['monster']['name'])
48     print()
49     while player1['pv'] > 0 and player2['pv'] > 0:
50         game_turn(player1, player2)
51         game_turn(player2, player1)
52
53     winner = get_winner(player1, player2)
54     print('Le joueur', winner['id'], 'remporte le combat avec',
55           winner['monster']['name'])

```

Listing 54 – tp/game.py

```

1 from .definitions import monsters
2 from .prompt import get_choice_input

```

```

3
4
5 def get_player(player_id):
6     print('Joueur', player_id, 'quel monstre choisissez-vous ?')
7     monster = get_choice_input(monsters, 'Monstre invalide')
8     pv = int(input('Quel est son nombre de PV ? '))
9     return {'id': player_id, 'monster': monster, 'pv': pv}
10
11
12 def get_players():
13     print('Monstres disponibles :')
14     for monster in monsters.values():
15         print('-', monster['name'])
16     return get_player(1), get_player(2)

```

Listing 55 – tp/players.py

```

1 def get_choice_input(choices, error_message):
2     entry = input('> ').lower()
3     while entry not in choices:
4         print(error_message)
5         entry = input('> ').lower()
6     return choices[entry]

```

Listing 56 – tp/prompt.py

[Retourner au texte.](#)

Contenu masqué n°15

```

1

```

Listing 57 – tp/__init__.py

```

1 from . import game
2
3
4 if __name__ == '__main__':
5     game.main()

```

Listing 58 – tp/__main__.py

```

1 monsters = {
2     'pythachu': {

```

```

3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponythya': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }

```

Listing 59 – tp/definitions.py

```

1  import sys
2
3  from .definitions import attacks
4  from .players import get_players
5  from .prompt import get_choice_input, get_yesno_input
6  from .save import load_game, save_game
7
8
9  def apply_attack(attack, opponent):
10     opponent['pv'] -= attack['damages']
11     if opponent['pv'] < 0:
12         opponent['pv'] = 0
13
14
15  def game_turn(player, opponent):
16     # Si le joueur est KO, il n'attaque pas
17     if player['pv'] <= 0:
18         return
19
20     print('Joueur', player['id'], 'quelle attaque utilisez-vous ?')
21     for name in player['monster']['attacks']:
22         print('-', name.capitalize(), '-attacks[name][\'damages\'],
23             'PV')
24
25     attack = get_choice_input(attacks, 'Attaque invalide')
26     apply_attack(attack, opponent)

```

```

26
27     print(
28         player['monster']['name'],
29         'attaque',
30         opponent['monster']['name'],
31         'qui perd',
32         attack['damages'],
33         'PV, il lui en reste',
34         opponent['pv'],
35     )
36
37
38 def get_winner(player1, player2):
39     if player1['pv'] > player2['pv']:
40         return player1
41     else:
42         return player2
43
44
45 def main():
46     if len(sys.argv) > 1:
47         filename = sys.argv[1]
48         try:
49             # Chargement de la sauvegarde
50             player1, player2 = load_game(filename)
51         except:
52             print('Échec du chargement de la sauvegarde',
53                 filename, file=sys.stderr)
54             return
55         else:
56             player1, player2 = get_players()
57
58     print()
59     print(player1['monster']['name'], 'affronte',
60           player2['monster']['name'])
61     print()
62
63     while player1['pv'] > 0 and player2['pv'] > 0:
64         if not get_yesno_input('Voulez-vous continuer ? ', True):
65             if get_yesno_input('Voulez-vous sauvegarder ? ',
66                               False):
67                 filename =
68                     input('Dans quel fichier sauvegarder ? ')
69                 save_game(filename, player1, player2)
70             return
71
72     game_turn(player1, player2)
73     game_turn(player2, player1)
74
75     winner = get_winner(player1, player2)

```

```

72     print('Le joueur', winner['id'], 'remporte le combat avec',
          winner['monster']['name'])

```

Listing 60 – tp/game.py

```

1  from .definitions import monsters
2  from .prompt import get_choice_input
3
4
5  def get_player(player_id):
6      print('Joueur', player_id, 'quel monstre choisissez-vous ?')
7      monster = get_choice_input(monsters, 'Monstre invalide')
8      pv = int(input('Quel est son nombre de PV ? '))
9      return {'id': player_id, 'monster': monster, 'pv': pv}
10
11
12 def get_players():
13     print('Monstres disponibles :')
14     for monster in monsters.values():
15         print('-', monster['name'])
16     return get_player(1), get_player(2)

```

Listing 61 – tp/players.py

```

1  def get_choice_input(choices, error_message):
2      entry = input('> ').lower()
3      while entry not in choices:
4          print(error_message)
5          entry = input('> ').lower()
6      return choices[entry]
7
8
9  def get_yesno_input(prompt, default):
10     if default:
11         prompt += '[O/n] '
12     else:
13         prompt += '[o/N] '
14
15     resp = input(prompt).lower()
16     if resp.startswith('o'):
17         return True
18     elif resp.startswith('n'):
19         return False
20     return default

```

Listing 62 – tp/prompt.py

```

1 import json
2
3 from .definitions import monsters
4
5
6 def load_game(filename):
7     with open(filename) as f:
8         player1, player2 = json.load(f)
9         # On récupère les monstres à partir de leurs noms
10        player1['monster'] = monsters[player1['monster']]
11        player2['monster'] = monsters[player2['monster']]
12        return player1, player2
13
14
15 def save_game(filename, player1, player2):
16     player1 = dict(player1)
17     player2 = dict(player2)
18     # On enregistre seulement le nom des monstres
19     player1['monster'] = player1['monster']['name'].lower()
20     player2['monster'] = player2['monster']['name'].lower()
21     with open(filename, 'w') as f:
22         json.dump([player1, player2], f)

```

Listing 63 – tp/save.py

[Retourner au texte.](#)

Contenu masqué n°16

```

1 import json
2
3 from tp.definitions import monsters, attacks
4 from tp.game import apply_attack, get_winner
5 from tp.save import load_game, save_game
6
7
8 def test_apply_attack():
9     player = {'id': 0, 'monster': monsters['pythachu'], 'pv': 100}
10
11    apply_attack(attacks['brûlure'], player)
12    assert player['pv'] == 60
13
14    apply_attack(attacks['tonnerre'], player)
15    assert player['pv'] == 10
16
17    apply_attack(attacks['charge'], player)
18    assert player['pv'] == 0

```

```

19
20
21 def test_get_winner():
22     player1 = {'id': 0, 'monster': monsters['pythachu'], 'pv': 100}
23     player2 = {'id': 0, 'monster': monsters['pythard'], 'pv': 0}
24     assert get_winner(player1, player2) == player1
25     assert get_winner(player2, player1) == player1
26
27     player2['pv'] = 120
28     assert get_winner(player1, player2) == player2
29     assert get_winner(player2, player1) == player2
30
31     player1['pv'] = player2['pv'] = 0
32     assert get_winner(player1, player2) == player2
33     assert get_winner(player2, player1) == player1
34
35
36 def test_load_game():
37     filename = 'test_game.dat'
38     with open(filename, 'w') as f:
39         json.dump([
40             {'id': 1, 'monster': 'pythachu', 'pv': 50},
41             {'id': 2, 'monster': 'pythard', 'pv': 40},
42         ], f)
43
44     p1, p2 = load_game(filename)
45     assert p1 == {
46         'id': 1,
47         'monster': monsters['pythachu'],
48         'pv': 50,
49     }
50     assert p2 == {
51         'id': 2,
52         'monster': monsters['pythard'],
53         'pv': 40,
54     }
55
56
57 def test_save_game():
58     filename = 'test_game.dat'
59     player1 = {
60         'id': 1,
61         'monster': monsters['pythachu'],
62         'pv': 30,
63     }
64     player2 = {
65         'id': 2,
66         'monster': monsters['ponytha'],
67         'pv': 20,
68     }

```

```
69     save_game(filename, player1, player2)
70
71     with open(filename) as f:
72         doc = json.load(f)
73
74     assert doc == [
75         {'id': 1, 'monster': 'pythachu', 'pv': 30},
76         {'id': 2, 'monster': 'ponytha', 'pv': 20},
77     ]
78
79
80 if __name__ == '__main__':
81     test_apply_attack()
82     test_get_winner()
83     test_load_game()
84     test_save_game()
```

Listing 64 – test_tp.py

[Retourner au texte.](#)

Septième partie

Aller plus loin

Introduction

Vous connaissez maintenant le Python et êtes en mesure de réaliser pas mal de programmes avec lui.

Mais dans notre course pour arriver à ce but, j'ai omis certains aspects du langage qu'il est important de connaître pour aller plus loin. Laissez-moi donc maintenant vous présenter tout cela pour compléter votre apprentissage.

VII.1. Les autres types de données

Introduction

Python n'est pas juste un monde de chaînes de caractères, de listes et de dictionnaires. De nombreux autres types existent qui apportent leurs particularités pour répondre à différents besoins.

Voici donc un tour d'horizon de quelques autres types de la bibliothèque standard.

VII.1.1. Les ensembles

Les ensembles sont des collections de données pour représenter des valeurs uniques. Dans un ensemble, il ne peut pas y avoir de doublons, un peu comme pour les clés de dictionnaires. D'ailleurs, la syntaxe pour définir un ensemble ressemble à celle des dictionnaires : un ensemble se définit à l'aide d'accolades à l'intérieur desquelles on sépare les valeurs par des virgules.

```
1 >>> {0, 1, 2, 3, 4, 5}
2 {0, 1, 2, 3, 4, 5}
```

Si l'on essaie d'insérer des doublons, on voit que ceux-ci ne sont pas pris en compte.

```
1 >>> {0, 1, 2, 3, 4, 5, 2, 3}
2 {0, 1, 2, 3, 4, 5}
```

Une autre particularité commune aux ensembles et aux dictionnaires est que les valeurs d'un ensemble doivent être *hashables*, impossible donc d'y stocker des listes.

```
1 >>> {[]}
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: unhashable type: 'list'
```

Il est aussi possible de convertir en ensemble un autre objet en appelant explicitement `set` (le type des ensembles).

```
1 >>> set([0, 1, 2, 3, 4, 5, 2, 3])
2 {0, 1, 2, 3, 4, 5}
```

Par ailleurs, `{}` étant la syntaxe pour définir un dictionnaire vide, un ensemble vide se définit avec `set()`.

```

1 >>> set()
2 set()

```

VII.1.1.1. Opérations

Les ensembles peuvent être considérés au sens mathématique du terme, une collection contenant juste des valeurs. Et il est ainsi possible d'appliquer des opérations ensemblistes à ces collections.

Ainsi, on peut calculer l'union entre deux ensembles à l'aide de l'opérateur `|`. L'union de deux ensembles consiste en l'ensemble des valeurs contenues dans l'un ou dans l'autre (ou les deux).

```

1 >>> {0, 1, 3, 4} | {2, 3, 4, 5}
2 {0, 1, 2, 3, 4, 5}

```

À l'inverse, l'intersection est obtenue avec l'opérateur `&`. L'intersection ne contient que les valeurs présentes dans les deux ensembles.

```

1 >>> {0, 1, 3, 4} & {2, 3, 4, 5}
2 {3, 4}

```

La différence est l'opération qui consiste à soustraire au premier ensemble les éléments du second. Elle se calcule avec l'opérateur `-`.

```

1 >>> {0, 1, 3, 4} - {2, 3, 4, 5}
2 {0, 1}

```

Enfin, `^` est l'opérateur de différence symétrique. La différence symétrique calcule l'ensemble des valeurs qui ne sont pas communes aux deux ensembles, c'est l'inverse de l'intersection. Ou autrement dit la différence entre l'union et l'intersection.

```

1 >>> {0, 1, 3, 4} ^ {2, 3, 4, 5}
2 {0, 1, 2, 5}
3 >>> ({0, 1, 3, 4} | {2, 3, 4, 5}) - ({0, 1, 3, 4} & {2, 3, 4, 5})
4 {0, 1, 2, 5}

```

J'ai représenté ici les ensembles comme des collections d'éléments ordonnés, mais il n'en est rien, aucune relation d'ordre n'existe dans un ensemble.

Ainsi, deux ensembles sont égaux s'ils contiennent exactement les mêmes valeurs, et différents dans le cas contraire.

```

1 >>> {1, 2, 3} == {3, 2, 1}
2 True
3 >>> {1, 2, 3} != {2, 3, 4}

```

VII. Aller plus loin

```
4 True
```

Il n'y a d'ailleurs pas d'accès direct aux éléments comme il peut y avoir sur une liste, car les éléments ne sont associés à aucun index.

Pour autant, il reste possible de parcourir un ensemble avec une boucle `for` pour itérer sur ses valeurs.

```
1 >>> for i in {1, 2, 3}:
2 ...     print(i)
3 ...
4 1
5 2
6 3
```

On peut tester si une valeur est présente dans un ensemble à l'aide de l'opérateur `in`. Et c'est là tout l'intérêt des ensembles : cette opération est optimisée pour s'exécuter en temps constant (là où il peut être nécessaire de parcourir tous les éléments sur une liste).

```
1 >>> 3 in {1, 2, 3}
2 True
3 >>> 4 in {1, 2, 3}
4 False
```

L'opérateur `not in` est l'inverse de `in`, il permet de tester l'absence de valeur.

```
1 >>> 3 not in {1, 2, 3}
2 False
3 >>> 4 not in {1, 2, 3}
4 True
```

Enfin, on trouve d'autres opérations ensemblistes liées aux opérateurs d'égalité.

`<`, `<=`, `>` et `>=` permettent de tester les sur-ensembles et sous-ensembles.

Avec `a` et `b` deux ensembles, `a <= b` est vraie si tous les éléments de `a` sont présents dans `b` (`a` est un sous-ensemble de `b`).

```
1 >>> {2, 3} <= {1, 2, 3, 4}
2 True
3 >>> {2, 3, 5} <= {1, 2, 3, 4}
4 False
```

Et l'opération est équivalente à `b >= a`, vue dans l'autre sens (`b` est un sur-ensemble de `a`).

```
1 >>> {1, 2, 3, 4} >= {2, 3}
2 True
3 >>> {1, 2, 3, 4} >= {2, 3, 5}
```

VII. Aller plus loin

```
4 False
```

`<` et `>` sont les pendants stricts de ces opérateurs : `a < b` ne sera pas vraie si `a` et `b` contiennent exactement les mêmes valeurs.

```
1 >>> {1, 2, 3} < {1, 2, 3, 4}
2 True
3 >>> {1, 2, 3} < {1, 2, 3}
4 False
5 >>> {1, 2, 3} <= {1, 2, 3}
6 True
```

VII.1.1.2. Méthodes

Les ensembles étant des collections, il est naturellement possible d'utiliser la fonction `len` pour calculer leur taille.

```
1 >>> len({1, 2, 3})
2 3
3 >>> len({1, 2, 3, 5})
4 4
```

Étant modifiables, il est possible d'ajouter et de retirer des éléments dans des ensembles. Cela se fait avec les fonctions `add` et `remove`.

```
1 >>> values = set()
2 >>> values.add(2)
3 >>> values.add(4)
4 >>> values.add(6)
5 >>> values
6 {2, 4, 6}
7 >>> values.remove(4)
8 >>> values
9 {2, 6}
```

La méthode `discard` est semblable à `remove` mais ne lève pas d'erreur si l'élément à supprimer est absent.

```
1 >>> values.remove(8)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   KeyError: 8
5 >>> values.discard(8)
6 >>> values.discard(2)
7 >>> values
```

VII. Aller plus loin

```
8 {6}
```

Et la méthode `pop` permet aussi de retirer (et renvoyer) un élément de l'ensemble, sans sélectionner lequel. Elle lève une exception si l'ensemble est vide.

```
1 >>> values.pop()
2 6
3 >>> values.pop()
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   KeyError: 'pop from an empty set'
```

On retrouve sinon différentes méthodes sur les ensembles équivalentes aux opérateurs décrits au-dessus : `union`, `intersection`, `difference` et `symmetric_difference`. L'avantage par rapport aux opérateurs est que ces méthodes peuvent prendre plusieurs ensembles en paramètres, ou même des objets de tous types (itérables) et opérer dessus.

```
1 >>> {1, 2}.union({2, 3}, [4], (5, 6))
2 {1, 2, 3, 4, 5, 6}
```

Chacune de ces méthodes est doublée d'une version qui modifie en place l'ensemble courant, respectivement `update`, `intersection_update`, `difference_update` et `symmetric_difference_update`. Ces méthodes ne renvoient rien.

```
1 >>> values = {1, 2, 3}
2 >>> values.intersection_update([3, 4, 5])
3 >>> values
4 {3}
```

Les ensembles disposent aussi de méthodes booléennes, notamment `issubset` et `issuperset` équivalentes aux opérateurs `<=` et `>=`, ainsi que `isdisjoint` pour tester si deux ensembles sont disjoints (dont l'intersection est vide).

```
1 >>> {1, 2, 3}.isdisjoint({4, 5, 6})
2 True
3 >>> {1, 2, 3}.isdisjoint({3, 4, 5})
4 False
```

Enfin, on retrouve les méthodes `clear` et `copy`, comme sur les listes et les dictionnaires, respectivement pour vider l'ensemble et pour en faire une copie.

VII.1.1.3. `frozenset`

Un ensemble étant une collection de données mutable, il n'est pas *hashable* et ne peut donc pas être utilisé comme clé de dictionnaire. Ainsi, un autre type existe pour représenter un ensemble immuable de données : le `frozenset`.

VII. Aller plus loin

Un `frozenset` se définit en appelant explicitement le type avec n'importe quel itérable en argument.

```
1 >>> frozenset({1, 2, 3})
2 frozenset({1, 2, 3})
```

Il peut aussi s'appeler seul pour définir un ensemble vide.

```
1 >>> frozenset()
2 frozenset()
```

Le `frozenset` dispose des mêmes méthodes et opérateurs que les ensembles classiques, à l'exception de celles qui modifient l'objet.

```
1 >>> frozenset({1, 2, 3}) | frozenset({3, 4, 5})
2 frozenset({1, 2, 3, 4, 5})
3 >>> frozenset({1, 2, 3}).isdisjoint(frozenset({3, 4, 5}))
4 False
```

Les ensembles et les `frozenset` sont compatibles entre eux, mais attention au type de retour qui dépendra de l'objet sur lequel la méthode ou l'opérateur est appliqué.

```
1 >>> frozenset({1, 2, 3}) & {3, 4, 5}
2 frozenset({3})
3 >>> {3, 4, 5} & frozenset({1, 2, 3})
4 {3}
```

VII.1.2. Module collections

Python dispose encore de nombreux autres types définis dans différents modules de sa bibliothèque standard. Par exemple le module `collections` propose plusieurs types pour gérer des collections de données avec diverses spécificités.

VII.1.2.1. Counter

Un problème courant en programmation est de vouloir compter des choses. Pour cela, les dictionnaires sont une bonne structure de données : on peut facilement associer un nombre à un élément et ainsi incrémenter ce nombre pour compter les occurrences d'un élément.

```
1 >>> numbers = [1, 2, 2, 3, 4, 4, 4]
2 >>> occurrences = {}
3 >>> for number in numbers:
4 ...     occurrences[number] = occurrences.get(number, 0) + 1
5 ...
```


VII. Aller plus loin

```
6 >>> occurrences
7 {1: 1, 2: 2, 3: 1, 4: 3}
```

Il y a en fait beaucoup plus simple avec le type `Counter` du module `collections`, spécialement dédié à compter des objets.

Il se comporte comme un dictionnaire où chaque clé non existante serait considérée comme associée à la valeur 0.

```
1 >>> from collections import Counter
2 >>> occurrences = Counter()
3 >>> occurrences[4]
4 0
5 >>> occurrences
6 Counter()
```

On peut donc facilement modifier les valeurs sans avoir à se demander si la clé existe déjà.

```
1 >>> occurrences[3] += 1
2 >>> occurrences[5] += 2
3 >>> occurrences
4 Counter({5: 2, 3: 1})
```

Quand une valeur est redéfinie, elle est donc présente «pour de vrai» dans le dictionnaire, même si elle nulle.

```
1 >>> occurrences[4] = 0
2 >>> occurrences
3 Counter({5: 2, 3: 1, 4: 0})
```

Un objet `Counter` peut être initialisé comme un dictionnaire : à partir d'un dictionnaire existant ou à l'aide d'arguments nommés.

```
1 >>> Counter({'foo': 3, 'bar': 5})
2 Counter({'bar': 5, 'foo': 3})
3 >>> Counter(foo=3, bar=5)
4 Counter({'bar': 5, 'foo': 3})
```

Mais il peut aussi être instancié avec un itérable quelconque, auquel cas il s'initialisera en comptant les différentes valeurs de cet itérable.

```
1 >>> Counter([1, 2, 3, 4, 3, 1, 3])
2 Counter({3: 3, 1: 2, 2: 1, 4: 1})
3 >>> Counter('tortue')
4 Counter({'t': 2, 'o': 1, 'r': 1, 'u': 1, 'e': 1})
```

Très pratique donc pour compter directement ce qui nous intéresse.

VII. Aller plus loin

En plus des opérations communes aux dictionnaires, on trouve aussi des opérations arithmétiques. Il est ainsi possible d'additionner deux compteurs, ce qui renvoie un nouveau compteur contenant les sommes des valeurs.

```
1 >>> Counter(a=5, b=1) + Counter(a=3, c=2)
2 Counter({'a': 8, 'c': 2, 'b': 1})
```

À l'inverse, la soustraction entre compteurs renvoie les différences. Les valeurs négatives sont ensuite retirées du résultat.

```
1 >>> Counter(a=5, b=1) - Counter(a=3, c=2)
2 Counter({'a': 2, 'b': 1})
```

Il est possible de calculer l'union et l'intersection entre deux objets `Counter`, l'union étant composée des maximums de chaque valeur et l'intersection des minimums (zéro compris).

```
1 >>> Counter(a=5, b=1) | Counter(a=3, c=2)
2 Counter({'a': 5, 'c': 2, 'b': 1})
3 >>> Counter(a=5, b=1) & Counter(a=3, c=2)
4 Counter({'a': 3})
```

i

On peut voir cela comme des opérations sur des ensembles où les éléments peuvent avoir plusieurs occurrences. Logiquement, l'intersection entre un ensemble qui contient 5 occurrences de `a` et un ensemble qui en contient 3 est un ensemble avec 3 `a`.

Enfin, les compteurs ajoutent quelques méthodes par rapport aux dictionnaires.

`most_common` par exemple permet d'avoir la liste ordonnée des valeurs les plus communes, associées à leur nombre d'occurrences. La méthode prend un paramètre `n` pour spécifier le nombre de valeurs que l'on veut obtenir (par défaut toutes les valeurs seront présentes).

```
1 >>> count = Counter('abcdabcaba')
2 >>> count.most_common()
3 [('a', 4), ('b', 3), ('c', 2), ('d', 1)]
4 >>> count.most_common(2)
5 [('a', 4), ('b', 3)]
```

La méthode `elements` permet d'itérer sur les valeurs comme si elles étaient représentées plusieurs fois selon leur nombre d'occurrences.

```
1 >>> for item in count.elements():
2     ...     print(item)
3     ...
4 a
5 a
```

VII. Aller plus loin

```
6 a
7 a
8 b
9 b
10 b
11 c
12 c
13 d
```

`update` est une méthode déjà présente sur les dictionnaires, qui a pour effet d'affecter de nouvelles valeurs aux clés existantes. Sur les compteurs, la méthode se chargera de faire la somme des valeurs.

Elle peut prendre n'importe quel itérable en argument, qu'elle considérera comme un compteur.

```
1 >>> count.update('bcde')
2 >>> count
3 Counter({'a': 4, 'b': 4, 'c': 3, 'd': 2, 'e': 1})
```

Il est aussi possible de faire la même chose en soustrayant les compteurs avec la méthode `subtract`.

```
1 >>> count.subtract('abcd')
2 >>> count
3 Counter({'a': 3, 'b': 3, 'c': 2, 'd': 1, 'e': 1})
```

VII.1.2.2. `defaultdict`

On a vu il y a quelques chapitres que les dictionnaires possédaient une méthode `setdefault`. Cette méthode permettait d'assurer qu'une valeur soit toujours présente pour une clé.

Cela simplifie des problèmes où l'on veut associer des listes de valeurs à des clés, comme un annuaire où chaque personne pourrait avoir plusieurs numéros.

```
1 >>> phonebook = {}
2 >>> phonebook.setdefault('Bob', []).append('0663621029')
3 >>> phonebook.setdefault('Bob', []).append('0714381809')
4 >>> phonebook.setdefault('Alice', []).append('0633432380')
5 >>> phonebook
6 {'Bob': ['0663621029', '0714381809'], 'Alice': ['0633432380']}
```

Mais les `defaultdict` permettent cela encore plus facilement : les valeurs manquantes seront automatiquement instanciées, sans besoin d'appel explicite à `setdefault`. Pour cela, un `defaultdict` s'instancie avec une fonction (ou un type) qui sera appelée à chaque clé manquante pour obtenir la valeur à affecter.

Ainsi, l'exemple précédent pourrait se réécrire comme suit.

```
1 >>> from collections import defaultdict
2 >>> phonebook = defaultdict(list)
3 >>> phonebook['Bob'].append('0663621029')
4 >>> phonebook['Bob'].append('0714381809')
5 >>> phonebook['Alice'].append('0633432380')
6 >>> phonebook
7 defaultdict(<class 'list'>, {'Bob': ['0663621029', '0714381809'],
    'Alice': ['0633432380']})
```

Chaque fois qu'une clé n'existe pas dans le dictionnaire, `defaultdict` fait appel à `list` qui renvoie une nouvelle liste vide.

Il suffit d'ailleurs d'essayer d'accéder à la valeur associée à une telle clé pour provoquer sa création.

```
1 >>> phonebook['Alex']
2 []
3 >>> phonebook
4 defaultdict(<class 'list'>, {'Bob': ['0663621029', '0714381809'],
    'Alice': ['0633432380'], 'Alex': []})
```

Et bien sûr, toute fonction pourrait être utilisée comme argument à `defaultdict`.

```
1 >>> def get_default_color():
2 ...     return 'noir'
3 ...
4 >>> colors = defaultdict(get_default_color)
5 >>> colors['mur'] = 'bleu'
6 >>> colors['mur']
7 'bleu'
8 >>> colors['sol']
9 'noir'
```

VII.1.2.3. `OrderedDict`

Avant Python 3.6 les dictionnaires ne conservaient pas l'ordre d'insertion des clés. La seule manière d'avoir un dictionnaire ordonné était d'utiliser le type `OrderedDict` du module `collections`. Les choses ont évolué depuis et le type a un peu perdu de son intérêt.

Comme les dictionnaires, un `OrderedDict` se construit à partir d'un dictionnaire existant et/ou d'arguments nommés. Sans argument, on construit simplement un dictionnaire vide.

```
1 >>> from collections import OrderedDict
2 >>> OrderedDict()
3 OrderedDict()
4 >>> OrderedDict({'foo': 0, 'bar': 1})
5 OrderedDict([('foo', 0), ('bar', 1)])
```

VII. Aller plus loin

```
6 >>> OrderedDict(foo=0, bar=1)
7 OrderedDict([('foo', 0), ('bar', 1)])
```

On le voit par sa représentation, le dictionnaire ordonné est en fait vu comme une liste de couples clé/valeur.

Il reste néanmoins une différence importante entre les dictionnaires ordonnés et les dictionnaires standards : l'ordre des éléments fait partie de la sémantique du premier.

Là où deux dictionnaires seront considérés comme égaux s'ils ont les mêmes couples clé/valeur, quel que soit leur ordre, ça ne sera pas le cas pour les `OrderedDict` qui ne seront égaux que si leurs clés sont dans le même ordre.

```
1 >>> {'foo': 0, 'bar': 1} == {'bar': 1, 'foo': 0}
2 True
3 >>> OrderedDict(foo=0, bar=1) == OrderedDict(bar=1, foo=0)
4 False
5 >>> OrderedDict(foo=0, bar=1) == OrderedDict(foo=0, bar=1)
6 True
```

Ce n'est bien sûr valable que pour l'égalité entre deux dictionnaires ordonnés. L'égalité entre un dictionnaire ordonné et un standard ne tiendra pas compte de l'ordre.

```
1 >>> OrderedDict(foo=0, bar=1) == {'bar': 1, 'foo': 0}
2 True
```

Faites donc appel à `OrderedDict` si vous avez besoin d'un tel comportement, sinon vous pouvez vous contenter d'un dictionnaire standard.

VII.1.2.4. ChainMap

Parfois on a plusieurs dictionnaires que l'on aimerait pouvoir considérer comme un seul, sans pour autant nécessiter de copie vers un nouveau dictionnaire qui les intégrerait tous. En effet, la copie peut être coûteuse et elle n'a surtout lieu qu'une fois, le dictionnaire copié ne sera pas affecté si les dictionnaires initiaux sont modifiés.

```
1 >>> phonebook_sim = {'Alice': '0633432380', 'Bob': '0663621029'}
2 >>> phonebook_tel = {'Alex': '0714381809'}
3 >>> phonebook = dict(phonebook_sim) # Copie pour fusionner les
   deux dictionnaires
4 >>> phonebook.update(phonebook_tel)
5 >>> phonebook
6 {'Alice': '0633432380', 'Bob': '0663621029', 'Alex': '0714381809'}
7 >>> phonebook_tel['Mehdi'] = '0762253973'
8 >>> phonebook # phonebook n'a pas changé
9 {'Alice': '0633432380', 'Bob': '0663621029', 'Alex': '0714381809'}
```

Le type `ChainMap` répond à ce problème puisqu'il permet de chaîner des dictionnaires dans un seul tout.

VII. Aller plus loin

```
1 >>> from collections import ChainMap
2 >>> phonebook_sim = {'Alice': '0633432380', 'Bob': '0663621029'}
3 >>> phonebook_tel = {'Alex': '0714381809'}
4 >>> phonebook = ChainMap(phonebook_sim, phonebook_tel)
5 >>> phonebook
6 ChainMap({'Alice': '0633432380', 'Bob': '0663621029'}, {'Alex':
  '0714381809'})
```

Lors de la recherche d'une clé, les dictionnaires seront parcourus successivement pour trouver la valeur.

```
1 >>> phonebook['Bob']
2 '0663621029'
3 >>> phonebook['Alex']
4 '0714381809'
```

Si la clé n'existe dans aucun dictionnaire, on obtient une erreur `KeyError` comme habituellement.

```
1 >>> phonebook['Mehdi']
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "/usr/lib/python3.8/collections/__init__.py", line 891, in
    __getitem__
5     return self.__missing__(key)          # support subclasses
        that define __missing__
6   File "/usr/lib/python3.8/collections/__init__.py", line 883, in
    __missing__
7     raise KeyError(key)
8 KeyError: 'Mehdi'
```

L'objet `ChainMap` ne contient que des références vers nos dictionnaires, et donc reflète bien les modifications sur ces derniers.

```
1 >>> phonebook_tel['Mehdi'] = '0762253973'
2 >>> phonebook['Mehdi']
3 '0762253973'
```

Aussi, il est possible de directement affecter des valeurs au `ChainMap`, celles-ci seront affectées au premier dictionnaire de la chaîne.

```
1 >>> phonebook['Julie'] = '0619096810'
2 >>> phonebook_sim
3 {'Alice': '0633432380', 'Bob': '0663621029', 'Julie': '0619096810'}
```

Il en est de même pour les clés qui existaient dans les dictionnaires suivants, elles seraient

VII. Aller plus loin

tout de même assignées au premier (c'est le seul accessible en écriture).

```
1 >>> phonebook['Alex'] = '0734593960'
2 >>> phonebook
3 ChainMap({'Alice': '0633432380', 'Bob': '0663621029', 'Julie':
  '0619096810', 'Alex': '0734593960'}, {'Alex': '0714381809',
  'Mehdi': '0762253973'})
```

On voit ainsi comment se passe la priorité entre les dictionnaires en lecture : la chaîne est parcourue et s'arrête au premier dictionnaire contenant la clé.

```
1 >>> phonebook['Alex']
2 '0734593960'
```

Cette fonctionnalité est très pratique pour mettre en place des espaces de noms, comme les scopes des fonctions en Python : des variables existent à l'intérieur de la fonction et sont prioritaires par rapport aux variables extérieures.

La méthode `new_child` et l'attribut `parents` sont utiles pour cela puisqu'ils permettent respectivement d'ajouter un nouveau dictionnaire en tête de la chaîne (qui comprendra donc toutes les futures modifications sur le `ChainMap`) et de récupérer la suite de la chaîne (la chaîne formée par tous les dictionnaires sauf le premier).

Tous deux renvoient un nouvel objet `ChainMap` sans altérer la chaîne courante.

```
1 >>> new_phonebook = phonebook.new_child()
2 >>> new_phonebook['Max'] = '0704779572'
3 >>> new_phonebook
4 ChainMap({'Max': '0704779572'}, {'Alice': '0633432380', 'Bob':
  '0663621029', 'Julie': '0619096810', 'Alex': '0734593960'},
  {'Alex': '0714381809', 'Mehdi': '0762253973'})
5 >>> new_phonebook.parents
6 ChainMap({'Alice': '0633432380', 'Bob': '0663621029', 'Julie':
  '0619096810', 'Alex': '0734593960'}, {'Alex': '0714381809',
  'Mehdi': '0762253973'})
```

`new_child` peut s'utiliser sans argument, auquel cas un dictionnaire vide sera ajouté, ou en donnant directement le dictionnaire à ajouter en argument.

```
1 >>> phonebook.new_child({'Max': '0704779572'})
2 ChainMap({'Max': '0704779572'}, {'Alice': '0633432380', 'Bob':
  '0663621029', 'Julie': '0619096810', 'Alex': '0734593960'},
  {'Alex': '0714381809', 'Mehdi': '0762253973'})
```

On retrouve sinon les mêmes méthodes que sur les dictionnaires.

VII.1.2.5. deque

En Python les tableaux sont trompeusement appelés des listes là où ce terme fait souvent référence à des listes chaînées. Un tableau représente des données contigües en mémoire, qui ne peuvent pas être morcellées, et occupe donc une zone mémoire continue qui dépend de sa taille.

Ainsi, lorsque l'on ajoute ou retire des éléments à un tableau, il peut être nécessaire d'adapter la taille de la zone mémoire, voire d'en trouver une nouvelle plus grande et d'y copier tous les éléments. Python fait cela pour nous, mais ce sont des opérations qui peuvent s'avérer coûteuses.

Les listes chaînées à l'inverse sont des chaînes constituées de maillons, chaque maillon étant un élément avec son propre espace mémoire, ceux-ci peuvent être n'importe où dans la mémoire. L'idée est que chaque maillon référence le précédent et/ou le suivant dans la chaîne.

On pourrait par exemple voir un maillon comme un dictionnaire avec 2 clés : `next` pour référencer le maillon suivant et `value` pour la valeur contenue (car l'idée est quand même bien d'y stocker des valeurs).

Voici ainsi un équivalent en liste chaînée de la liste `[1, 2, 3, 4]`.

```
1 >>> node4 = {'next': None, 'value': 4}
2 >>> node3 = {'next': node4, 'value': 3}
3 >>> node2 = {'next': node3, 'value': 2}
4 >>> node1 = {'next': node2, 'value': 1}
5 >>> values = node1
```

i

On utilise `None` pour marquer la fin de la chaîne, indiquant qu'il n'y a plus d'autre maillon après `node4`.

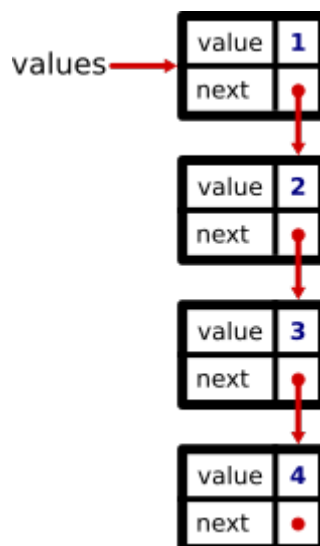


FIGURE VII.1.1. – Liste chaînée

Les variables `node1`, `node2` etc. ne sont que temporaires pour la création de notre liste, elles n'existent plus après, seule `values` référence notre chaîne de maillons.

VII. Aller plus loin

```
1 >>> del node1
2 >>> del node2
3 >>> del node3
4 >>> del node4
```

Il nous serait alors possible d'itérer sur notre liste chaînée pour accéder à chacune des valeurs.

```
1 >>> node = values # La liste représente le premier maillon
2 >>> while node is not None: # None représente la fin de liste
3 ...     print(node['value'])
4 ...     node = node['next'] # On passe au nœud suivant en
    réaffectant node
5 ...
6 1
7 2
8 3
9 4
```

Mais il n'est pas question ici de recoder une liste chaînée, Python en propose déjà une avec le type `deque` du module `collections`.

deque pour *double-end queue*, c'est-à-dire une queue (liste chaînée) dont les deux extrémités sont connues (le premier et le dernier maillon) et les liaisons sont doubles (chaque maillon référence le précédent et le suivant), contrairement à notre implémentation où seul le premier maillon était connu et les liaisons étaient simples (référence vers le maillon suivant uniquement).

Le principe est sinon le même. Un *deque* se construit comme une liste, soit vide soit à partir d'un itérable existant.

```
1 >>> deque()
2 deque([])
3 >>> deque([1, 2, 3, 4])
4 deque([1, 2, 3, 4])
```

Et le type propose les mêmes méthodes que les listes, ce sont juste les algorithmes derrière qui sont différents, et certaines opérations qui sont à privilégier plutôt que d'autres.

```
1 >>> values = deque([1, 2, 3, 4])
2 >>> values[0]
3 1
4 >>> values.append(5)
5 >>> values
6 deque([1, 2, 3, 4, 5])
```

Par exemple, contrairement aux tableaux (`list`) il est très facile (peu coûteux) d'ajouter des éléments au début ou à la fin, puisqu'il suffit d'insérer un nouveau maillon à l'extrémité et de changer la référence. De même pour supprimer un élément au début ou à la fin.

Les *deque* proposent d'ailleurs des méthodes dédiées avec `appendleft` et `popleft`, équivalentes

VII. Aller plus loin

à `append` et `pop` mais pour opérer au début de la liste.

```
1 >>> values.appendleft(0)
2 >>> values
3 deque([0, 1, 2, 3, 4, 5])
4 >>> values.popleft()
5 0
6 >>> values
7 deque([1, 2, 3, 4, 5])
```

En revanche, comme seules les extrémités sont connues, il est coûteux d'aller chercher un élément en milieu de liste, puisqu'il est nécessaire pour cette opération de parcourir tous les maillons jusqu'au bon élément.

```
1 >>> values[2]
2 3
```

Pour accéder à cette valeur, il a fallu parcourir 3 maillons. Il aurait fallu en parcourir 500 pour atteindre le milieu d'une liste chaînée de 1000 éléments. Là où pour un tableau l'accès à chaque élément est direct puisque son emplacement mémoire est connu : il se calcule facilement à partir de la position du premier élément, les éléments étant contigus en mémoire.

Ainsi, ne faites appel aux listes chaînées que pour des opérations qui nécessiteraient de souvent ajouter et/ou supprimer des données en début/fin de séquence, c'est là leur intérêt par rapport aux tableaux.

Ne les utilisez pas si vous devez accéder régulièrement à des éléments situés loin des extrémités, les performances pourraient être désastreuses.

VII.1.2.6. `namedtuple`

Pour terminer avec le module `collections`, j'aimerais vous parler des *named tuples* (tuples nommés).

Vous le savez, un tuple représente un ensemble cohérent de données, par exemple deux coordonnées qui identifient un point dans le plan. Il est sinon semblable à une liste (bien que non modifiable) et permet d'accéder aux éléments à partir de leur position.

```
1 >>> point = (3, 5)
2 >>> point[0]
3 3
```

Et par *unpacking* il est possible d'accéder à ses éléments indépendamment.

```
1 >>> x, y = point
2 >>> y
3 5
```

Mais ne serait-il pas plus pratique de pouvoir directement taper `point.y` pour accéder à

VII. Aller plus loin

l'ordonnée du point ? C'est plus facilement compréhensible que `point[1]` et moins contraignant que l'*unpacking* qui nécessite de définir une nouvelle variable.

Vous le voyez venir, c'est ce que proposent les tuples nommés : donner des noms aux éléments d'un tuple. Mais tout d'abord, il faut créer un type associé à ces tuples nommés, pour définir justement les noms de champs. Car un tuple nommé identifiant un point ne sera pas la même chose qu'un tuple nommé identifiant une couleur par exemple.

Nous allons donc devoir définir un nouveau type et c'est précisément ce que fait la fonction `namedtuple` du module `collections` : elle crée dynamiquement un type de tuples nommés en fonction des arguments qu'elle reçoit. Pour ça, elle prend en arguments le nom du type à créer (utilisé pour la représentation de l'objet) et la liste des noms des champs des tuples.

La fonction renvoie un type, mais il faudra assigner ce retour à une variable pour pouvoir l'utiliser, comme tout autre retour de fonction. Les types en Python sont en fait des objets comme les autres, qui peuvent donc être assignés à des variables. Par convention, on utilisera un nom commençant par une majuscule, pour identifier un type.

```
1 >>> from collections import namedtuple
2 >>> Point = namedtuple('Point', ('x', 'y'))
3 >>> Point
4 <class '__main__.Point'>
```

Ensuite, pour instancier un objet `Point`, on appelle le type en lui donnant en arguments les deux coordonnées `x` et `y`.

```
1 >>> point = Point(3, 5)
2 >>> point
3 Point(x=3, y=5)
4 >>> point.x
5 3
```

On le voit à sa représentation, il est aussi possible d'instancier l'objet en utilisant des arguments nommés.

```
1 >>> Point(x=10, y=7)
2 Point(x=10, y=7)
```

Notre objet `point` est toujours un tuple, et il reste possible de l'utiliser comme tel.

```
1 >>> point[0]
2 3
3 >>> x, y = point
4 >>> y
5 5
```

VII.1.3. Types

Nous avons maintenant vu de nombreux types Python, mais savons-nous reconnaître les valeurs d'un certain type ? Oui, à l'usage on sait différencier une chaîne de caractères d'un nombre, parce qu'ils se représentent différemment, et qu'on y applique des opérations différentes.

Mais il est possible d'aller plus loin dans la reconnaissance des types et nous allons voir quels outils sont mis à disposition par Python pour cela.

Premièrement, la *fonction* `type`¹ permet de connaître le type d'un objet. On lui donne une valeur en argument et la fonction nous renvoie simplement son type.

```
1 >>> type(5)
2 <class 'int'>
3 >>> type('foo')
4 <class 'str'>
5 >>> type([0])
6 <class 'list'>
```

Cela peut être utile dans des phases de débogage, pour s'assurer qu'une valeur est bien du type auquel on pense.

On peut aussi l'utiliser dans le code pour vérifier le type d'un objet mais ce n'est généralement pas recommandé (car trop strict, voir plus bas).

```
1 >>> def check_type(value):
2 ...     if type(value) is str:
3 ...         print("C'est une chaîne de caractères")
4 ...     else:
5 ...         print("Ce n'est pas une chaîne de caractères")
6 ...
7 >>> check_type('foo')
8 C'est une chaîne de caractères
9 >>> check_type(5)
10 Ce n'est pas une chaîne de caractères
```

L'autre outil mis à disposition de Python pour reconnaître le type d'une valeur est la fonction `isinstance`. Cette fonction reçoit une valeur et un type, et renvoie un booléen selon que la valeur soit de ce type ou non.

```
1 >>> isinstance('foo', str)
2 True
3 >>> isinstance(5, str)
4 False
```

Mais une valeur n'est pas d'un seul type, il existe en fait une hiérarchie entre les types. Par exemple, tous les objets Python sont des instances du type `object`, car `object` est le parent de tous les types.

1. C'est en fait plus compliqué que cela et je ne rentrerai pas dans les détails ici, mais `type` est lui-même un type, le type de tous les types. Nous ne l'utiliserons dans ce tutoriel que comme une fonction.

VII. Aller plus loin

```
1 >>> isinstance('foo', object)
2 True
3 >>> isinstance(5, object)
4 True
```

Ou encore, avec notre objet `point` construit précédemment, qui est à la fois une instance de `Point` et de `tuple`.

```
1 >>> type(point)
2 <class '__main__.Point'>
3 >>> isinstance(point, Point)
4 True
5 >>> isinstance(point, tuple)
6 True
```

Cela nous montre une première limitation de l'appel à `type` pour vérifier le type, qui ne verrait pas que nos valeurs sont aussi des `object`, ou notre point un `tuple`.

```
1 >>> type('foo') is object
2 False
3 >>> type(5) is object
4 False
5 >>> type(point) is tuple
6 False
```

Vérifier avec `type` est donc à limiter aux cas où l'on veut s'assurer strictement du type d'un objet, sans considérer la hiérarchie des types, et ce sont des cas assez rares.

Il faut cependant faire attention aussi aux appels à `isinstance` et les utiliser avec parcimonie, au risque de contrevenir à une caractéristique importante du Python, le *duck-typing*.



Le *duck-typing* (*typage canard*) est une philosophie dans la reconnaissance des types des valeurs. Elle repose sur la phrase «Si cela a un bec, marche comme un canard et cancanne comme un canard, alors je peux considérer que c'est un canard».

Appliqué au Python, cela veut dire par exemple qu'on préfère savoir qu'un objet se comporte comme une liste (que les mêmes opérations y sont applicables) plutôt que de vérifier que ce soit réellement une liste. On dit aussi que les valeurs doivent avoir la même interface qu'une liste.

Cela laisse la possibilité aux développeurs d'utiliser les types de leur choix tout en gardant une compatibilité avec les fonctions existantes.

C'est tout le principe des itérables : les fonctions de Python n'attendent jamais précisément une liste mais juste un objet sur lequel on puisse itérer. Que ce soit une liste, un *tuple*, une chaîne de caractères ou encore un fichier, peu importe.

Ainsi, on évitera les `if isinstance(value, list): ...` si ce n'est pas strictement nécessaire (un traitement particulier à réserver aux objets de ce type), pour ne pas laisser de côté les autres types qui auraient pu convenir tels que les *tuples*.

VII. Aller plus loin

Mais `isinstance` ne se limite pas à des types clairement définis et permet aussi de vérifier des interfaces. C'est ce que propose le module `collections.abc` qui fournit une collection de types abstraits (*abc* pour *abstract base classes*, classes mères abstraites), des interfaces correspondant à des comportements en particulier.

On trouve ainsi un type `Iterable`. Il n'est pas utilisable en tant que tel, on ne peut pas instancier d'objets du type `Iterable`, mais on peut l'utiliser pour vérifier qu'un objet est bien itérable en appelant `isinstance`.

```
1 >>> from collections.abc import Iterable
2 >>> isinstance([1, 2, 3], Iterable)
3 True
4 >>> isinstance((4, 5, 6), Iterable)
5 True
6 >>> isinstance('hello', Iterable)
7 True
8 >>> isinstance(42, Iterable)
9 False
```

Il y a aussi `Hashable` par exemple pour vérifier qu'une valeur est hashable, que l'on peut l'utiliser en tant que clé dans un dictionnaire ou la stocker dans un ensemble.

```
1 >>> from collections.abc import Hashable
2 >>> isinstance(42, Hashable)
3 True
4 >>> isinstance('hello', Hashable)
5 True
6 >>> isinstance([1, 2, 3], Hashable)
7 False
8 >>> isinstance((4, 5, 6), Hashable)
9 True
```

On trouve encore d'autres types abstraits définis dans `collections.abc` mais il est un peu tôt pour les aborder.

VII.2. Retour sur les conditions

VII.2.1. Instructions et expressions

Dans le cours j'ai plusieurs fois utilisé le terme d'«expression». Une expression est un élément de syntaxe Python qui possède une valeur quand il est évalué.

'foo', $3 * 5 + 2$ ou encore `max(range(10))` sont des expressions.

Si je dis ça maintenant, c'est parce qu'il n'y a pas uniquement des expressions en Python. Plus généralement, on trouve des instructions. L'instruction c'est la définition au sens large d'un élément de syntaxe, pour résumer on pourrait dire que c'est une ligne de code.

Ainsi, les expressions sont des instructions, mais toutes les instructions ne sont pas des expressions. Une expression c'est ce qu'on peut utiliser partout où une valeur est attendue : en argument à une fonction, dans une assignation de variable, dans une condition, etc.

```
1 >>> len('foo')
2 3
3 >>> x = 3 * 5 + 2
4 >>> if max(range(10)):
5 ...     print('ok')
6 ...
7 ok
```

Dit autrement, une expression c'est ce que l'on peut mettre entre parenthèses.

```
1 >>> ('foo')
2 'foo'
3 >>> (3 * 5 + 2)
4 17
5 >>> (max(range(10)))
6 9
```

Et par exemple une assignation de variable n'est pas une expression, elle ne possède aucune valeur, pas même `None`. Si l'on cherche à placer une assignation entre parenthèses on obtient une erreur de syntaxe.

```
1 >>> (foo = 'bar')
2 File "<stdin>", line 1
3     (foo = 'bar')
4         ^
5 SyntaxError: invalid syntax
```

De la même manière, les conditions ne sont pas des expressions, il s'agit de blocs de code.

```
1 >>> (if True: print('ok'))
2     File "<stdin>", line 1
3         (if True: print('ok'))
4         ^
5 SyntaxError: invalid syntax
```

Pourtant il serait pratique de pouvoir utiliser directement une condition dans un argument de fonction ou une assignation...

VII.2.2. Expressions conditionnelles

Et c'est heureusement possible grâce aux expressions conditionnelles. Comme leur nom l'indique, ce sont des conditions sous forme d'expressions.

Elles reprennent les mêmes mots-clés `if` et `else` mais sans construire de bloc, leur syntaxe est la suivante :

```
1 valeur if condition else autre_valeur
```

Cette expression vaut `valeur` si `condition` est vraie et `autre_valeur` sinon.

```
1 >>> 'good' if 5 + 3 == 8 else 'bad'
2 'good'
3 >>> 'good' if 5 + 3 == 7 else 'bad'
4 'bad'
```

Étant une expression, elle doit toujours avoir une valeur, c'est pourquoi le `else` est obligatoire dans tous les cas.

Les expressions conditionnelles permettent d'avoir un code plus concis lorsque les conditions à traiter sont simples.

```
1 >>> x = 3
2 >>> y = 5
3 >>> z = (2 * x if x < 10 else x) / (y if y else 1)
```

Sans elles, il nous aurait fallu écrire le code suivant :

```
1 >>> if x < 10:
2 ...     tmp1 = 2 * x
3 ... else:
4 ...     tmp1 = x
5 ...
6 >>> if y:
7 ...     tmp2 = y
8 ... else:
```


VII. Aller plus loin

```
9 ...     tmp2 = 1
10 ...
11 >>> z = tmp1 / tmp2
```

Elles sont souvent utilisées aussi lors d'appels de fonctions ou méthodes.

```
1 >>> sep = None
2 >>> 'a,b,c'.split(sep if sep is not None else ',')
3 ['a', 'b', 'c']
```

On parle aussi de «conditions ternaires» pour qualifier les expressions conditionnelles, car c'est un opérateur à 3 opérandes (`op1 if op2 else op3`).

VII.3. Retour sur les boucles

VII.3.1. Cas des boucles infinies

Nous avons vu les boucles `for` pour itérer sur des données, puis les boucles `while` pour boucler sur une condition. Et nous avons vu que, volontairement ou non, nous pouvions tomber dans des cas de boucles infinies.

```
1 while True:
2     print("Vers l'infini et au-delà !")
```



Pour rappel, utilisez la combinaison de touches `Ctrl` + `C` pour couper le programme.

Volontairement, ça peut être pour laisser tourner un programme en tâche de fond—un serveur par exemple—qui s'exécuterait continuellement pour traiter des requêtes. Et dans ce cas des dispositifs seront mis en place pour terminer proprement le programme quand on le souhaite. Mais il y a d'autres cas d'usages légitimes de boucles a priori infinies, car il existe d'autres moyens de terminer une boucle en cours d'exécution.

En effet, la condition d'un `while` est parfois difficile à exprimer, d'autant plus si elle repose sur des événements tels que des `input`. Dans ce cas, un idiome courant est d'écrire une boucle infinie et d'utiliser un autre moyen de sortir de la boucle : le mot-clé `break`.

Ce mot-clé, quand il est rencontré, a pour effet de stopper immédiatement la boucle en cours, sans repasser par la condition.

```
1 while True:
2     value = input('Entrez un nombre: ')
3     if value.isdigit():
4         value = int(value)
5         break
6     else:
7         print('Nombre invalide')
```

Avec cette boucle, nous attendons que l'entrée ne soit composée que de chiffres, auquel cas on rentre dans le `if` et l'on atteint le `break`. Sinon, on continue de boucler en redemandant à l'utilisateur de saisir un nouveau nombre.

La boucle, infinie en apparence (`while True`), possède en fait une condition de fin exprimée par un `if`.

VII.3.2. Contrôle du flux

`break` permet donc de stopper la boucle. Il n'est pas seulement disponible pour les boucles

VII. Aller plus loin

`while`, on peut aussi l'utiliser dans un `for`.

```
1 >>> for i in range(10):
2     ...     print(i)
3     ...     if i == 5:
4     ...         break
5     ...
6 0
7 1
8 2
9 3
10 4
11 5
```

Comme précédemment, la sortie de boucle est immédiate, l'effet ne serait donc pas le même si le `print` était placé après le bloc `if`.

```
1 >>> for i in range(10):
2     ...     if i == 5:
3     ...         break
4     ...     print(i)
5     ...
6 0
7 1
8 2
9 3
10 4
```

Il faut savoir que dans le cas de boucles imbriquées, `break` ne se rapporte qu'à la boucle juste au-dessus. Il n'est pas possible d'influer sur les boucles extérieures.

```
1 >>> for x in range(3):
2     ...     for y in range(3):
3     ...         if y == 2:
4     ...             break
5     ...         print(x, y)
6     ...
7 0 0
8 0 1
9 1 0
10 1 1
11 2 0
12 2 1
```

Mais `break` n'est pas le seul mot-clé de contrôle du flux d'une boucle et je vais maintenant vous parler de `continue`.

`continue` permet aussi de terminer immédiatement l'itération en cours, mais pour passer à la suivante. Quand un `continue` est rencontré, on est directement conduit à la ligne d'introduction

VII. Aller plus loin

de la boucle et sa condition est réévaluée.

```
1 while True:
2     value = input('Entrez un nombre: ')
3     if not value:
4         break
5     if not value.isdigit():
6         print('Nombre invalide')
7         continue
8     value = int(value)
9     print(f'{value} * 2 = {value * 2}')
```

C'est un mot-clé très utile quand on traite une liste de données et que l'une des valeurs est invalide, on peut alors simplement l'ignorer et passer à la suivante.

```
1 values = [1, 2, 3, -1, 4, 5]
2
3 total = 0
4 for value in values:
5     if value < 0:
6         print('Invalid value', value)
7         continue
8     total += value
```

On a aussi le mot-clé `else` qui est assez facile à comprendre sur une boucle `while` : il intervient après la boucle si la condition a été évaluée comme fausse.

```
1 pv = 50
2
3 while pv > 0:
4     print(f'Pythachu a {pv} PV')
5     pv -= 20
6     print('Pythachu perd 20 PV')
7 else:
8     print('Pythachu est KO')
```

Le `else` intervient donc dans tous les cas... sauf si on a quitté la boucle sans réévaluer la condition (qui ne peut donc pas être fausse), c'est-à-dire en utilisant un `break`.

Ainsi, `else` permet de savoir comment s'est terminée une boucle, si on en est sorti normalement (auquel cas on passe dans le bloc) ou si on l'a interrompue (le bloc n'est pas exécuté).

```
1 pv = 50
2
3 while pv > 0:
4     print(f'Pythachu a {pv} PV')
5     degats = input('Nombre de degats : ')
6     if not degats.isdigit():
7         continue
8     pv -= int(degats)
```

```

7         break
8     degats = int(degats)
9     pv -= degats
10    print(f'Pythachu perd {degats} PV')
11 else:
12    print('Pythachu est KO')
```

`else` est aussi applicable à la boucle `for` en ayant le même effet, il permet de savoir si la boucle est arrivée jusqu'au bout sans être interrompue.

Ainsi, sans `break` le `else` est bien exécuté.

```

1 >>> for i in range(5):
2 ...     print(i)
3 ... else:
4 ...     print('end')
5 ...
6 0
7 1
8 2
9 3
10 4
11 end
```

Avec un `break` il ne l'est pas.

```

1 >>> for i in range(5):
2 ...     print(i)
3 ...     if i == 3:
4 ...         break
5 ... else:
6 ...     print('end')
7 ...
8 0
9 1
10 2
11 3
```



Le mot-clé `else` est souvent mal compris—on pourrait croire qu'on entre dans le `else` uniquement s'il n'y a pas eu d'itérations—et donc peu recommandé pour lever toute ambiguïté.

VII.3.3. Outils

Le monde de l'itération est très vaste en Python, les itérables se retrouvent au cœur de nombreux mécanismes. C'est pourquoi Python propose de base de nombreux outils relatifs à l'itération

VII. Aller plus loin

tels que les fonctions `all` et `any` que l'on a déjà vues.

Vous êtes-vous déjà demandé comment itérer simultanément sur plusieurs listes ou comment répéter une liste ? Ce chapitre est fait pour vous !

VII.3.3.1. Fonctions natives (*builtins*)

On a déjà vu un certain nombre de *builtins* dans les chapitres précédents, mais il en reste quelques unes très intéressantes que j'ai omises jusqu'ici.

VII.3.3.1.1. `enumerate`

Notamment la fonction `enumerate`, qui prend une liste (ou n'importe quel itérable) et permet d'itérer sur ses valeurs tout en leur associant leur index. C'est-à-dire que pour chaque valeur on connaîtra la position qu'elle occupe dans la liste.

```
1 >>> values = ['abc', 'def', 'ghi']
2 >>> for i, value in enumerate(values):
3 ...     print(i, ': ', value)
4 ...
5 0 : abc
6 1 : def
7 2 : ghi
```

Cela remplace aisément les constructions à base de `range(len(values))` que l'on voit trop souvent et qui sont à éviter.

```
1 >>> for i in range(len(values)):
2 ...     print(i, ': ', values[i])
3 ...
4 0 : abc
5 1 : def
6 2 : ghi
```

On les évite justement parce qu'`enumerate` répond mieux au problème tout en étant plus polyvalent (on peut par exemple itérer sur un fichier), et qu'on a directement accès à la valeur (`value`) sans besoin d'une indirection supplémentaire par le conteneur (`values[i]`).

On notera au passage que la fonction `enumerate` accepte un deuxième argument pour préciser l'index de départ, qui est par défaut de zéro.

```
1 >>> with open('corbeau.txt') as f:
2 ...     for i, line in enumerate(f, 1):
3 ...         print(i, ': ', line.rstrip())
4 ...
5 1 : Maître Corbeau, sur un arbre perché,
6 2 : Tenait en son bec un fromage.
7 3 : Maître Renard, par l'odeur alléché,
8 4 : Lui tint à peu près ce langage :
```

VII. Aller plus loin

```
9 5 : Et bonjour, Monsieur du Corbeau.
10 6 : Que vous êtes joli ! que vous me semblez beau !
11 7 : Sans mentir, si votre ramage
12 8 : Se rapporte à votre plumage,
13 9 : Vous êtes le Phénix des hôtes de ces bois.
14 10 : À ces mots, le Corbeau ne se sent pas de joie ;
15 11 : Et pour montrer sa belle voix,
16 12 : Il ouvre un large bec, laisse tomber sa proie.
17 13 : Le Renard s'en saisit, et dit : Mon bon Monsieur,
18 14 : Apprenez que tout flatteur
19 15 : Vit aux dépens de celui qui l'écoute.
20 16 : Cette leçon vaut bien un fromage, sans doute.
21 17 : Le Corbeau honteux et confus
22 18 : Jura, mais un peu tard, qu'on ne l'y prendrait plus.
```

VII.3.3.2. `reversed`

`reversed` est une fonction très simple, elle permet d'inverser une séquence d'éléments, pour les parcourir dans l'ordre inverse.

```
1 >>> values = ['abc', 'def', 'ghi']
2 >>> for value in reversed(values):
3     ...     print(value)
4     ...
5 ghi
6 def
7 abc
```

La fonction ne modifie pas la séquence initiale (contrairement à la méthode `reverse` des listes).

```
1 >>> values
2 ['abc', 'def', 'ghi']
```

VII.3.3.3. `sorted`

Dans la même veine on a la fonction `sorted`, semblable à la méthode `sort` des listes mais renvoyant ici une copie.

```
1 >>> values = [5, 3, 2, 4, 6, 1, 9, 7, 8]
2 >>> sorted(values)
3 [1, 2, 3, 4, 5, 6, 7, 8, 9]
4 >>> values
5 [5, 3, 2, 4, 6, 1, 9, 7, 8]
```

VII. Aller plus loin

On notera que le tri se fait en ordre croissant (les plus petits éléments d'abord) par défaut, mais la fonction accepte un argument `reverse` pour trier en ordre décroissant (les plus grands d'abord).

```
1 >>> sorted(values, reverse=True)
2 [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Mieux encore, la fonction propose un paramètre `key` pour personnaliser la manière dont seront triés nos éléments. C'est une fonction qui recevra un élément en paramètre et renverra une valeur (par exemple un nombre), le tri se fera alors suivant l'ordre entre ces valeurs renvoyées.

i

Les fonctions en Python sont des valeurs comme les autres que l'on peut donc parfaitement passer en argument. Ces arguments-fonctions sont généralement appelés des *callbacks* (ou «fonctions de rappel»).

Par exemple, le tri par défaut pour les chaînes de caractères est l'ordre lexicographique (plus ou moins équivalent à l'ordre alphabétique).

```
1 >>> words = ['zèbre', 'autruche', 'cheval', 'oie']
2 >>> sorted(words)
3 ['autruche', 'cheval', 'oie', 'zèbre']
```

On pourrait alors préciser une fonction de tri `key=len` pour les trier par taille.

```
1 >>> sorted(words, key=len)
2 ['oie', 'zèbre', 'cheval', 'autruche']
```

En effet, la fonction `len` sera appelée pour chaque mot et les mots seront triés suivant le retour de la fonction (en l'occurrence 3, 5, 6 et 8). Mais il est possible d'utiliser n'importe quelle fonction en tant que clé de tri, tant que cette fonction renvoie quelque chose d'ordonnable. Voici un autre exemple avec une fonction pour trier les mots dans l'ordre alphabétique mais en commençant par la dernière lettre du mot.

```
1 >>> def key_func(word):
2 ...     return word[::-1] # On renvoie le mot à l'envers
3 ...
4 >>> key_func('autruche')
5 'ehcurtua'
6 >>> sorted(words, key=key_func)
7 ['autruche', 'oie', 'zèbre', 'cheval']
```

Ces deux arguments sont aussi disponibles sur la méthode `sort` des listes.


```
1 >>> words.sort(key=len, reverse=True)
2 >>> words
3 ['autruche', 'cheval', 'zèbre', 'oie']
```

VII.3.3.4. `min` et `max`

On a déjà vu les fonctions `min` et `max` qui permettent respectivement de récupérer le minimum/maximum parmi leurs arguments.

```
1 >>> min(3, 1, 2)
2 1
3 >>> max(3, 1, 2)
4 3
```

On sait aussi qu'on peut les appeler avec un seul argument (un itérable) et récupérer le minimum/maximum dans cet itérable.

```
1 >>> min({3, 1, 2})
2 1
3 >>> max([3, 1, 2])
4 3
```

Mais sachez maintenant que ces fonctions acceptent aussi un argument `key` qui fonctionne de la même manière que pour `sorted`.

Ainsi il est possible d'expliquer comment doivent être comparées les valeurs. On peut alors simplement demander la valeur minimale/minimale d'une liste en comparant les nombres selon leur valeur absolue.

```
1 >>> min([-5, -2, 1, 3], key=abs)
2 1
3 >>> max([-5, -2, 1, 3], key=abs)
4 -5
```

Ces fonctions acceptent aussi un argument `default` dont la valeur est renvoyée (plutôt qu'une erreur) si l'itérable est vide.

```
1 >>> min([])
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ValueError: min() arg is an empty sequence
5 >>> min([], default=42)
6 42
```

VII.3.3.5. `zip`

`zip` est une fonction très pratique de Python, puisqu'elle permet de parcourir simultanément plusieurs itérables. On appelle la fonction en lui fournissant nos itérables en arguments, et l'on itère ensuite sur l'objet qu'elle nous renvoie.

Les éléments que l'on obtient alors sont des tuples formés des éléments de nos itérables de départ.

```
1 >>> for elem in zip(words, 'abcd', range(4)):
2     ...     print(elem)
3     ...
4 ('autruche', 'a', 0)
5 ('cheval', 'b', 1)
6 ('zèbre', 'c', 2)
7 ('oie', 'd', 3)
```

Il est ainsi possible d'utiliser l'*unpacking* de Python pour avoir quelque chose de plus explicite.

```
1 >>> for word, letter, number in zip(words, 'abcd', range(4)):
2     ...     print(word, letter, number)
3     ...
4 autruche a 0
5 cheval b 1
6 zèbre c 2
7 oie d 3
```

`zip` accepte autant d'arguments que l'on souhaite, on peut l'appeler avec deux itérables comme avec dix.

Aussi, il s'arrête dès que l'un des itérables se termine, puisqu'il ne peut alors plus produire de tuple contenant un élément de chaque.

```
1 >>> for i, j in zip(range(2, 6), range(10)):
2     ...     print(i, j)
3     ...
4 2 0
5 3 1
6 4 2
7 5 3
```

VII.3.3.6. Module `itertools`

En plus des outils *built-in* pour manipuler les itérables, la bibliothèque standard fournit aussi une mine d'or : le module `itertools` [↗](#).

Je ne détaillerai pas tout ce que contient le module, la documentation fera cela beaucoup mieux que moi. Je veux juste vous présenter quelques fonctions qui pourraient vous être bien utiles.

VII.3.3.6.1. chain

Comme son nom l'indique, `chain` permet de chaîner plusieurs itérables, de façon transparente et quels que soient leurs types.

```
1 >>> from itertools import chain
2 >>> for letter in chain('ABC', ['D', 'E'], ('F', 'G')):
3 ...     print(letter)
4 ...
5 A
6 B
7 C
8 D
9 E
10 F
11 G
```

VII.3.3.6.2. zip_longest

`zip_longest` est un équivalent à `zip` qui ne s'arrête pas au premier itérable terminé mais qui continue jusqu'au dernier. Les valeurs manquantes seront alors complétées par `None`, ou par la valeur précisée au paramètre `fillvalue`.

```
1 >>> from itertools import zip_longest
2 >>> for i, j in zip_longest(range(2, 6), range(10)):
3 ...     print(i, j)
4 ...
5 2 0
6 3 1
7 4 2
8 5 3
9 None 4
10 None 5
11 None 6
12 None 7
13 None 8
14 None 9
15 >>> for letter1, letter2 in zip_longest('ABCD', 'EF',
16 ...     print(letter1, letter2)
17 ...
18 A E
19 B F
20 C .
21 D .
```

VII.3.3.6.3. product

`product` calcule le produit cartésien entre plusieurs itérables, c'est-à-dire qu'il produit toutes les combinaisons d'éléments possibles.

```
1 >>> from itertools import product
2 >>> for i, c in product(range(5), 'ABC'):
3 ...     print(i, c)
4 ...
5 0 A
6 0 B
7 0 C
8 1 A
9 1 B
10 1 C
11 2 A
12 2 B
13 2 C
14 3 A
15 3 B
16 3 C
17 4 A
18 4 B
19 4 C
```

Cela revient à écrire des boucles `for` imbriquées tout en économisant des niveaux d'indentation. L'exemple précédent est ainsi équivalent au code suivant.

```
1 for i in range(5):
2     for c in 'ABC':
3         print(i, c)
```

Le module propose d'autres fonctions combinatoires que je vous invite à regarder.

VII.3.3.6.4. Recettes

En plus de donner des explications et exemples pour chacune de ses fonctions, la documentation du module `itertools` [fournit aussi quelques «recettes»](#) [↗](#).

Il s'agit de fonctions qui répondent à des besoins trop particuliers pour être vraiment intégrées au module. Les recettes sont là pour que vous les repreniez dans votre code et que vous les adaptiez à votre convenance.

VII.3.4. Listes en intension

On a vu qu'il était possible d'écrire des conditions sous forme d'expressions, qu'en est-il des boucles ?

Une expression est une instruction qui possède une valeur. Pour une condition c'est facile : on a une valeur si la condition est vraie et une autre valeur sinon. Mais quelle pourrait être la valeur d'une boucle ?

VII. Aller plus loin

Il n'y a pas de réponse évidente à cette question, et c'est pourquoi il n'y a pas d'expression générale pour exécuter une boucle. Il existe en revanche les listes en intension, qui permettent de construire une liste à partir d'une boucle `for`.

L'intension est un concept mathématique qui s'oppose à l'extension pour définir un ensemble¹. La définition par extension, c'est celle que nous avons utilisée jusqu'ici, qui consiste à définir l'ensemble par les éléments qu'il possède.

```
1 powers = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

La définition par intension consiste elle à décrire l'ensemble selon une règle, par exemple «les dix premières puissances de 2». On la traduirait en Python par le code suivant :

```
1 >>> powers = [2**i for i in range(10)]
2 >>> powers
3 [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

On voit alors que l'on utilise le `for` dans une expression pour construire une liste. Le code précédent est équivalent à la boucle suivante.

```
1 powers = []
2 for i in range(10):
3     powers.append(2**i)
```

On peut ainsi transposer vers une liste en intension toute boucle `for` ne consistant qu'à évaluer une expression à chaque itération.

```
1 >>> [letter + '!' for letter in 'ABCD']
2 ['A!', 'B!', 'C!', 'D!']
3 >>> [len(word) for word in ['zeste', 'de', 'savoir']]
4 [5, 2, 6]
5 >>> [letter * i for letter, i in zip('ABCD', range(1, 5))]
6 ['A', 'BB', 'CCC', 'DDDD']
```

i

Le terme anglais pour les listes en intension est *list comprehensions*, aussi il est courant de rencontrer en français les expressions «liste en compréhension» ou «compréhension de liste», il s'agit évidemment de la même chose.

VII.3.4.1. Conditions de filtrage

Mais les listes en intension ne s'arrêtent pas là et permettent des constructions plus complexes : il est possible de filtrer les éléments à intégrer ou non à la liste. Pour cela on utilise une expression de la forme suivante.

1. https://fr.wikipedia.org/wiki/Intension_et_extension ↗

VII. Aller plus loin

```
1 [expression for item in iterable if condition]
```

La condition interviendra à chaque itération et déterminera s'il faut ajouter `expression` aux éléments de la liste en construction ou non. Voici par exemple la liste des entiers naturels pairs strictement inférieurs à 10.

```
1 >>> [i for i in range(10) if i % 2 == 0]
2 [0, 2, 4, 6, 8]
```

Ce code est équivalent à la boucle suivante :

```
1 values = []
2 for i in range(10):
3     if i % 2 == 0:
4         values.append(i)
```



Attention à ne pas confondre le `if` utilisé ici avec le `if` de l'expression conditionnelle. Ce premier n'autorise pas le `else` puisque cela n'aurait pas de sens sur une condition de filtrage.

Par ailleurs, les expressions conditionnelles étant des expressions à part entière, il est parfaitement possible de les utiliser dans des listes en intension.

```
1 >>> [i // 2 if i % 2 == 0 else i * 3 + 1 for i in range(10)]
2 [0, 4, 1, 10, 2, 16, 3, 22, 4, 28]
```

On peut même les combiner aux conditions de filtrage sans que cela ne pose problème, veillez tout de même à ce que le code reste toujours lisible.

```
1 >>> [i // 2 if i % 2 == 0 else i * 3 + 1 for i in range(10) if i %
2     3 == 0]
2 [0, 10, 3, 28]
```

Pour plus de clarté, il est ainsi parfois conseillé de placer des parenthèses autour de l'expression conditionnelle. Mais de manière générale, une liste en intension trop longue peut signifier que ce n'est pas la meilleure solution au problème et qu'une boucle «standard» irait tout aussi bien.

```
1 >>> [(i // 2 if i % 2 == 0 else i * 3 + 1) for i in range(10) if i
2     % 3 == 0]
2 [0, 10, 3, 28]
```

Il est aussi possible d'utiliser plusieurs `if` dans l'intension pour définir plusieurs conditions sur lesquelles filtrer, celles-ci s'additionnant les unes aux autres.

```
1 >>> [i for i in range(10) if i % 2 == 0 if i % 3 == 0] # Multiples
    de 2 et 3
2 [0, 6]
```

VII.3.4.2. Boucles imbriquées

D'ailleurs, les `for` aussi peuvent être chaînés au sein d'une même intension. Cela permet alors de faire la même chose qu'avec des boucles imbriquées pour remplir notre liste.

```
1 >>> [(i, c) for i in range(3) for c in 'AB']
2 [(0, 'A'), (0, 'B'), (1, 'A'), (1, 'B'), (2, 'A'), (2, 'B')]
```

Les boucles sont à lire de gauche à droite comme si elles étaient écrites de haut en bas, le code précédent est équivalent à :

```
1 values = []
2 for i in range(3):
3     for c in 'AB':
4         values.append((i, c))
```

Et il est possible d'enchaîner autant de `for` que l'on veut dans l'intension, comme l'on pourrait en imbriquer autant qu'on veut. Mais attention, nous obtenons bien une seule liste en sortie, comportant toutes les combinaisons parcourues lors de l'itération.

Les listes en intension étant des expressions comme les autres, il est aussi possible d'imbriquer les intensions. C'est ainsi que l'on peut construire des listes à plusieurs dimensions.

```
1 >>> table = [[0 for x in range(3)] for y in range(2)]
2 >>> table
3 [[0, 0, 0], [0, 0, 0]]
```

C'est un modèle de construction assez courant en Python puisqu'il ne souffre pas du problème de références multiples dont je parlais lors de la présentation des listes. Ici, chaque sous-liste est une instance différente et peut donc être modifiée indépendamment des autres.

```
1 >>> table[0][1] = 5
2 >>> table
3 [[0, 5, 0], [0, 0, 0]]
```

Souvenez-vous, ce n'est pas le résultat qu'on obtenait avec `[[0] * 3] * 2` où chaque ligne était une référence vers la même liste.

```
1 >>> table = [[0] * 3] * 2
2 >>> table
3 [[0, 0, 0], [0, 0, 0]]
4 >>> table[0][1] = 5
5 >>> table
6 [[0, 5, 0], [0, 5, 0]]
```

VII.3.4.3. Autres constructions en intension

On parle souvent de listes en intension mais ce n'est pas le seul type qui peut être construit ainsi. Au programme, on trouve aussi les ensembles et les dictionnaires. Pour les ensembles, la syntaxe est identique aux listes à l'exception qu'on utilise des accolades plutôt que des crochets.

```
1 >>> {i**2 for i in range(10)}
2 {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

Et on retrouve les mêmes fonctionnalités sur les intensions : il est possible d'avoir plusieurs boucles et d'utiliser des conditions de filtrage.

```
1 >>> {i+j for i in range(10) for j in range(10) if (i+j) % 2 == 0}
2 {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
```

Vous constaterez pour ce dernier exemple que le résultat ne serait pas du tout le même avec une liste, l'ensemble ne permettant pas les duplications.

Pour les dictionnaires on retrouve quelque chose de similaire mais utilisant la syntaxe `cle: valeur` plutôt qu'une simple expression (où `cle` et `valeur` sont aussi des expressions).

```
1 >>> {i: i**2 for i in range(10)}
2 {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

VII.3.5. Itérateurs

VII.3.5.1. Itérables et itérateurs

Depuis plusieurs chapitres j'utilise le terme d'itérables pour qualifier les objets qui peuvent être parcourus à l'aide d'une boucle `for`, mais qu'en est-il ? On a vu qu'il existait un grand nombre d'itérables, tels que les chaînes de caractères, les listes, les *range*, les dictionnaires, les fichiers, etc.

Il y en a d'autres encore et l'on en a vu plus récemment dans ce chapitre : les retours des fonctions `enumerate` ou `zip` sont aussi des itérables. Mais si on les regarde de plus près, on voit qu'ils sont un peu particuliers.

VII. Aller plus loin

```
1 >>> enumerate('abcde')
2 <enumerate object at 0x7f30749e0240>
3 >>> zip('abc', 'def')
4 <zip object at 0x7f30749e02c0>
```

Ou plutôt on ne voit pas grand chose justement, ces objets sont assez intrigants. On sait qu'ils sont itérables, on l'a vu plus tôt, et on peut donc se servir de cette propriété pour les transformer en liste si c'est ce qui nous intéresse.

```
1 >>> list(enumerate('abcde'))
2 [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
3 >>> list(zip('abc', 'def'))
4 [('a', 'd'), ('b', 'e'), ('c', 'f')]
```

Mais ce qui est plus étonnant c'est qu'on ne peut itérer dessus qu'une seule fois.

```
1 >>> values = enumerate('abcde')
2 >>> for v in values:
3 ...     print(v)
4 ...
5 (0, 'a')
6 (1, 'b')
7 (2, 'c')
8 (3, 'd')
9 (4, 'e')
10 >>> for v in values:
11 ...     print(v)
12 ...
```

On constate le même comportement avec la conversion en liste.

```
1 >>> values = zip('abc', 'def')
2 >>> list(values)
3 [('a', 'd'), ('b', 'e'), ('c', 'f')]
4 >>> list(values)
5 []
```

Une fois parcourus une première fois, il n'est plus possible d'itérer à nouveau sur leurs valeurs. Contrairement à d'autres itérables comme les listes ou les *ranges* que l'on parcourt autant de fois que l'on veut.

En fait, ces objets *enumerate* et *zip* ne sont pas seulement des itérables, ils sont des itérateurs. Un itérateur peut se voir comme un curseur qui se déplace le long d'un itérable, et qui logiquement se consume à chaque étape. Ici l'objet *enumerate* est donc un itérateur le long de notre chaîne `'abcde'`.

La fonction `next` en Python permet de récupérer la prochaine valeur d'un itérateur. Elle prend l'itérateur en argument et renvoie la valeur pointée par le curseur tout en le faisant avancer.

VII. Aller plus loin

Puisque l'itérateur avance, le retour de la fonction sera différent à chaque appel.

```
1 >>> values = enumerate('abcde')
2 >>> next(values)
3 (0, 'a')
4 >>> next(values)
5 (1, 'b')
6 >>> next(values)
7 (2, 'c')
```

En fin de parcours, l'itérateur lève une exception `StopIteration` pour signaler que l'itération est terminée.

```
1 >>> next(values)
2 (3, 'd')
3 >>> next(values)
4 (4, 'e')
5 >>> next(values)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 StopIteration
```

On ne peut alors pas revenir en arrière : une fois notre itérateur parcouru il est entièrement consommé. C'est pourquoi il n'est pas possible de faire deux `for` à la suite sur un même objet `enumerate` ou `zip`, ils sont à usage unique.

i

À noter que la fonction `next` accepte un second argument qui est la valeur à renvoyer dans le cas où l'itérateur est consommé plutôt que lever une exception.

```
1 >>> next(values, '')
2 ''
```

Mais ces objets se basent sur des itérables réutilisables que sont les chaînes de caractères, listes ou autres : on peut donc à nouveau appeler `enumerate` pour obtenir un itérateur tout neuf et recommencer à boucler.

```
1 >>> values = 'abcde'
2 >>> for v in enumerate(values):
3 ...     print(v)
4 ...
5 (0, 'a')
6 (1, 'b')
7 (2, 'c')
8 (3, 'd')
9 (4, 'e')
```

```
10 >>> for v in enumerate(values):
11     ...     print(v)
12     ...
13 (0, 'a')
14 (1, 'b')
15 (2, 'c')
16 (3, 'd')
17 (4, 'e')
```

VII.3.5.2. Fonctions `map` et `filter`

En évoquant les outils d'itération plus tôt, j'ai volontairement omis les fonctions `map` et `filter`. Parce que leurs fonctionnalités sont couvertes par les listes en intension et parce qu'elles renvoient des itérateurs.

`map` et `filter` sont issues de la programmation fonctionnelle et servent respectivement à convertir et à filtrer les données d'un itérable.

`map` prend en arguments une fonction et un itérable, et applique la fonction à chaque élément de l'itérable, renvoyant un itérateur sur les résultats.

```
1 >>> values = [1.3, 2.5, 3.8, 4.2]
2 >>> map(round, values)
3 <map object at 0x7f4ae2db16a0>
4 >>> list(map(round, values))
5 [1, 2, 4, 4]
```

Cela revient donc à utiliser la liste en intension suivante.

```
1 >>> [round(v) for v in values]
2 [1, 2, 4, 4]
```

`filter` est le pendant pour le filtrage des éléments. Ici le premier argument est une fonction utilisée comme prédicat : l'élément est conservé si le prédicat est vrai et ignoré sinon.

```
1 >>> def greater_than_two(n):
2     ...     return n >= 2
3     ...
4 >>> list(filter(greater_than_two, values))
5 [2.5, 3.8, 4.2]
```

Ici, la liste en intension équivalente serait la suivante.

```
1 >>> [v for v in values if v >= 2]
2 [2.5, 3.8, 4.2]
```

`map` et `filter` existaient avant les listes en intension et sont moins utilisées aujourd'hui, surtout lorsqu'il s'agit de les transformer en listes. Elles restent parfois utilisées quand on n'attend rien

VII. Aller plus loin

de plus qu'un itérateur, par exemple pour fournir en argument à une autre fonction. C'est le cas de `str.join` qui attend un itérable de chaînes de caractères et nécessite donc que les données soient converties en chaînes, ce que permet `map`.

```
1 >>> ', '.join(map(str, values))
2 '1.3, 2.5, 3.8, 4.2'
```

VII.3.5.3. Itérateurs infinis

Comme je disais, un itérateur ne représente qu'un curseur, il a donc une empreinte très faible en mémoire. Mieux encore, il n'a même pas besoin de s'appuyer sur des données qui existent déjà, celles-ci peuvent être générées à la volée lors du parcours.

C'est déjà le principe des objets *range* qui occupent très peu d'espace : tous les nombres de l'intervalle ne sont pas stockés en mémoire à la création du *range*, ils sont simplement calculés pendant l'itération et disparaissent après.

On peut pousser le concept plus loin et itérer sur des données qui ne pourraient jamais tenir dans la mémoire de l'ordinateur, des données infinies. C'est le cas des itérateurs que nous allons voir ici, ils ne se terminent jamais.

Ces itérateurs infinis sont tirés du module `itertools`.

Le plus simple d'entre tous c'est `count`, qui permet de compter de 1 en 1.

```
1 >>> from itertools import count
2 >>> numbers = count()
3 >>> next(numbers)
4 0
5 >>> next(numbers)
6 1
7 >>> next(numbers)
8 2
```

À quoi cela peut-il servir ? C'est très pratique pour générer des identifiants uniques puisque chaque appel à `next` renverra un nombre différent.

```
1 >>> id_seq = count()
2 >>> def new_event():
3 ...     return {'id': next(id_seq), 'type': 'monstre', 'message':
4 ...             'Un pythachu sauvage apparaît'}
5 >>> new_event()
6 {'id': 0, 'type': 'monstre', 'message': 'Un pythachu sauvage
   apparaît'}
7 >>> new_event()
8 {'id': 1, 'type': 'monstre', 'message': 'Un pythachu sauvage
   apparaît'}
9 >>> new_event()
```

VII. Aller plus loin

Cela peut être aussi utile mathématiquement, pour simplement calculer un seuil à partir duquel une propriété est vraie.

```
1 >>> for i in count():
2 ...     if 2**i > 1000:
3 ...         break
4 ...
5 >>> i
6 10
```

On sait ainsi que 2^{10} est la première puissance de 2 à être supérieur à 1000.

On notera que `count` peut prendre deux arguments : le premier est le nombre de départ (0 par défaut) et le second est le pas (1 par défaut).

```
1 >>> numbers = count(1, 2)
2 >>> next(numbers)
3 1
4 >>> next(numbers)
5 3
6 >>> next(numbers)
7 5
```

Un autre itérateur infini est `repeat`, qui répète simplement en boucle le même élément.

```
1 >>> from itertools import repeat
2 >>> values = repeat('hello')
3 >>> next(values)
4 'hello'
5 >>> next(values)
6 'hello'
```

On pourra le voir utilisé dans des `zip` pour simuler une séquence de même longueur qu'une autre.

```
1 >>> def additions(seq1, seq2):
2 ...     for i, j in zip(seq1, seq2):
3 ...         print(f'{i} + {j} = {i+j}')
4 ...
5 >>> additions(range(10), repeat(5))
6 0 + 5 = 5
7 1 + 5 = 6
8 2 + 5 = 7
9 3 + 5 = 8
10 4 + 5 = 9
11 5 + 5 = 10
12 6 + 5 = 11
13 7 + 5 = 12
```

VII. Aller plus loin

```
14 8 + 5 = 13
15 9 + 5 = 14
```

`repeat` peut aussi prendre un argument qui indique le nombre de répétitions à effectuer, auquel cas il ne sera plus infini.

```
1 >>> list(repeat('hello', 5))
2 ['hello', 'hello', 'hello', 'hello', 'hello']
```

Dans le même genre on trouve enfin `cycle` pour boucler (indéfiniment) sur un même itérable.

```
1 >>> from itertools import cycle
2 >>> values = cycle(['hello', 'world'])
3 >>> next(values)
4 'hello'
5 >>> next(values)
6 'world'
7 >>> next(values)
8 'hello'
```

C'est aussi un cas d'utilisation pour avoir un itérable que l'on voudrait au moins aussi grand qu'un autre.

```
1 >>> additions(range(10), cycle([3, 5, 8]))
2 0 + 3 = 3
3 1 + 5 = 6
4 2 + 8 = 10
5 3 + 3 = 6
6 4 + 5 = 9
7 5 + 8 = 13
8 6 + 3 = 9
9 7 + 5 = 12
10 8 + 8 = 16
11 9 + 3 = 12
```

VII.3.5.4. Fonction `iter`

Pour terminer ce chapitre je voudrais vous parler d'`iter`, une fonction qui renvoie un simple itérateur sur l'itérable donné en argument. Un nouvel itérateur est construit et renvoyé à chaque appel sur l'itérable.

```
1 >>> values = [0, 1, 2, 3, 4]
2 >>> iter(values)
3 <list_iterator object at 0x7f3074a28850>
4 >>> iter(values)
```

VII. Aller plus loin

```
5 <list_iterator object at 0x7f3074a28bb0>
```

Ces itérateurs sont semblables à nos objets `enumerate`, on peut appeler `next` dessus et récupérer la valeur suivante. Ils sont donc utiles si l'on souhaite parcourir manuellement un itérable à coups de `next`.

```
1 >>> it = iter(values)
2 >>> next(it)
3 0
4 >>> next(it)
5 1
6 >>> next(it)
7 2
```

Et bien sûr on peut aussi les parcourir avec un `for`. Attention encore, l'itérateur avance pendant le parcours, et le `for` continuera donc l'itération à partir d'où il se trouve.

```
1 >>> for v in it:
2 ...     print(v)
3 ...
4 3
5 4
6 >>> for v in it:
7 ...     print(v)
8 ...
```

Les itérateurs étant des itérables, il est possible de les donner à leur tour à `iter`. La fonction renverra alors simplement le même itérateur.

```
1 >>> it
2 <list_iterator object at 0x7f3074a21070>
3 >>> iter(it)
4 <list_iterator object at 0x7f3074a21070>
```

On constate bien que les deux valeurs ont la même adresse.

VII.4. Retour sur les fonctions

VII.4.1. Arguments optionnels

Nous savons déclarer une fonction avec des paramètres simples, et leur associer des arguments lors de l'appel, qu'ils soient positionnels ou nommés.

```
1 >>> def log(message, component, level):
2 ...     print(f'[{level}] {component}: {message}')
3 ...
4 >>> log('Une erreur est survenue', 'system', 'error')
5 [error] system: Une erreur est survenue
6 >>> log('Une erreur est survenue', 'system', level='error')
7 [error] system: Une erreur est survenue
```

Nous obtenons un message d'erreur si nous omettons un des arguments.

```
1 >>> log('Une erreur est survenue', 'system')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: log() missing 1 required positional argument: 'level'
```

Pourtant nous avons vu qu'il existait dans la bibliothèque standard des fonctions avec des arguments optionnels, comment font-elles ? Cela se passe au moment de la définition des paramètres de la fonction, ou une valeur par défaut est donnée à certains paramètres. Les arguments optionnels correspondent simplement aux paramètres ayant une valeur par défaut. Pour définir une valeur par défaut à un paramètre, il suffit d'écrire `parametre=valeur` plutôt que `parametre` dans la liste des paramètres de la fonction. Voici ainsi une version plus évoluée de notre fonction `log`, s'appuyant sur des valeurs par défaut.

```
1 >>> def log(message, component=None, level='info'):
2 ...     if component is None:
3 ...         print(f'[{level}] {message}')
4 ...     else:
5 ...         print(f'[{level}] {component}: {message}')
6 ...
7 >>> log('Une erreur est survenue', 'system', 'error')
8 [error] system: Une erreur est survenue
9 >>> log("Message d'information")
10 [info] Message d'information
11 >>> log('Fonction dépréciée', level='warning')
```


12 [warning] Fonction dépréciée

VII.4.1.1. Paramètres par défaut mutables

C'est aussi simple que cela... ou presque ! Il y a une chose à laquelle il faut faire attention, comme toujours, ce sont les types mutables.

Eh oui, les valeurs par défaut sont définies une seule fois pour toutes, quand la fonction elle-même est définie. C'est-à-dire que ces valeurs seront partagées entre tous les appels à la fonction.

Pour les valeurs immutables, pas de problème, il n'y a pas de risque d'effets de bord. Mais pour les mutables, faites bien attention à ce que vous faites, on arrive rapidement à des situations problématiques.

```

1 >>> def get_monster(name, attacks=[]):
2 ...     return {'name': name, 'attacks': attacks}
3 ...
4 >>> pythachu = get_monster('Pythachu')
5 >>> pythachu['attacks'].append('tonnerre')
6 >>> pythachu
7 {'name': 'Pythachu', 'attacks': ['tonnerre']}
8 >>> pythard = get_monster('Pythard')
9 >>> pythard
10 {'name': 'Pythard', 'attacks': ['tonnerre']}
```

Et oui, la même liste d'attaque a été utilisée et donc partagée entre nos deux dictionnaires, d'où le bug. C'est pourquoi il est généralement conseillé d'éviter les mutables comme valeurs par défaut de paramètres.

Pour cela, on utilisera une valeur comme `None` (appelée sentinelle) qui indiquera l'absence de valeur et permettra donc d'instancier un objet (ici une liste) dans le corps de la fonction, évitant le problème de l'instance partagée.

```

1 >>> def get_monster(name, attacks=None):
2 ...     if attacks is None:
3 ...         attacks = []
4 ...     return {'name': name, 'attacks': attacks}
5 ...
6 >>> pythachu = get_monster('Pythachu')
7 >>> pythachu['attacks'].append('tonnerre')
8 >>> pythachu
9 {'name': 'Pythachu', 'attacks': ['tonnerre']}
10 >>> pythard = get_monster('Pythard')
11 >>> pythard
12 {'name': 'Pythard', 'attacks': []}
```

Je dis «généralement» car il y a des cas où c'est le comportement voulu, cela permet de mettre en place facilement un mécanisme de cache¹ sur une fonction par exemple.

1. Un cache est une mémoire associée à une fonction, pour éviter de réexécuter des calculs coûteux.

```
1 >>> def compute(x, cache={}):
2 ...     if x in cache:
3 ...         return cache[x]
4 ...     print('Calcul complexe...')
5 ...     ret = x**3 - x**2
6 ...     cache[x] = ret
7 ...     return ret
8 ...
9 >>> compute(2)
10 Calcul complexe...
11 4
12 >>> compute(3)
13 Calcul complexe...
14 18
15 >>> compute(2) # Réutilisation du cache
16 4
17 >>> compute(5)
18 Calcul complexe...
19 100
```

VII.4.1.2. Ordre de placement des paramètres

Nous l'avons vu, lors d'un appel de fonction les arguments positionnels doivent toujours être placés avant les arguments nommés. C'est ce qui permet à Python de faire correctement la correspondance entre arguments et paramètres.

```
1 >>> log(level='warning', 'Avertissement')
2 File "<stdin>", line 1
3 SyntaxError: positional argument follows keyword argument
```

Une règle similaire existe pour les paramètres : ceux qui prennent une valeur par défaut doivent se placer après les autres. Cela est logique puisqu'ils sont optionnels, et qu'on ne pourrait pas savoir dans le cas contraire à quel paramètre est censé correspondre un argument.

```
1 >>> def log(component=None, level='info', message):
2 ...     pass
3 ...
4 File "<stdin>", line 1
5 SyntaxError: non-default argument follows default argument
```

VII.4.2. Arguments variadiques

Vous pensiez avoir tout vu sur les arguments ? Que nenni ! Certaines fonctions que nous utilisons couramment exploitent encore des fonctionnalités inconnues.

VII. Aller plus loin

Si je vous demandais par exemple de recoder la fonction `print`, comment procéderiez-vous ? Pour rappel, la fonction permet de recevoir un nombre variable d'arguments.

```
1 >>> print()
2
3 >>> print(1)
4 1
5 >>> print(1, 2, 3)
6 1 2 3
```

On pourrait essayer de placer plusieurs paramètres optionnels à la suite mais on ne couvrirait jamais tous les cas : si l'on créait une fonction avec 10 paramètres optionnels il ne serait pas possible de l'appeler avec 11 arguments.

Il doit donc y avoir autre chose, une manière de gérer un nombre variable d'arguments : les arguments variadiques ! L'idée derrière ce nom est simplement de récupérer les arguments positionnels sous forme d'une liste (ou plutôt d'un tuple).

Et cela se fait avec une syntaxe plutôt simple en Python, il suffit de placer `*args` dans la liste des paramètres de la fonction. On obtiendra ainsi un tuple `args` contenant ces arguments.

```
1 >>> def print_args(*args):
2 ...     print(args)
3 ...
4 >>> print_args()
5 ()
6 >>> print_args(1)
7 (1,)
8 >>> print_args(1, 2, 3)
9 (1, 2, 3)
```

`args` est ici un nom complètement arbitraire (mais très couramment utilisé) pour nommer cette liste, et n'importe quel autre nom fonctionnerait tout aussi bien. C'est le `*` placé avant qui a pour effet de récupérer les arguments et non le nom donné au paramètre.

Avec `*args`, tous les arguments sont ainsi optionnels. Mais il est aussi possible de préciser d'autres paramètres avant `*args`, qui ne récupérera alors que le reste des arguments : cela permet alors de conserver des arguments obligatoires.

```
1 >>> def my_sum(first, *args):
2 ...     for n in args:
3 ...         first += n
4 ...     return first
5 ...
6 >>> my_sum(1)
7 1
8 >>> my_sum(1, 2, 3, 4, 5)
9 15
10 >>> my_sum()
11 Traceback (most recent call last):
```

```

12 File "<stdin>", line 1, in <module>
13 TypeError: my_sum() missing 1 required positional argument: 'first'

```

Si vous testez un peu, vous remarquerez que cette syntaxe est valide pour les arguments positionnels mais pas les arguments nommés.

```

1 >>> print_args(foo='bar')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: print_args() got an unexpected keyword argument 'foo'

```

En effet, comment un tuple d'arguments pourrait représenter nos arguments nommés ? Mais il existe une autre syntaxe pour récupérer les arguments nommés, sous forme d'un dictionnaire cette fois : `**kwargs`. `kwargs` pour *keyword arguments* (arguments nommés) car il ne pourra récupérer que les arguments qui sont explicitement nommés. Là encore le nom du paramètre n'est qu'une convention.

```

1 >>> def print_args(*args, **kwargs):
2     ...     print(args, kwargs)
3     ...
4 >>> print_args()
5 () {}
6 >>> print_args(3, 5, foo='bar', toto='tata')
7 (3, 5) {'foo': 'bar', 'toto': 'tata'}

```

Le paramètre spécial `**kwargs` ne peut se placer que tout à la fin de la liste des paramètres puisqu'il récupère les arguments qui n'ont pas été attrapés par les paramètres précédents. `*args` quant à lui peut se placer à peu près où vous le souhaitez (avant `**kwargs`) mais souvenez-vous qu'il attrape tous les arguments positionnels, donc les paramètres situés après ne pourront récupérer que des arguments nommés.

```

1 >>> def print_args(foo, *args, bar, **kwargs):
2     ...     print(foo, args, bar, kwargs)
3     ...
4 >>> print_args(1, 2, 3, bar=4, baz=5)
5 1 (2, 3) 4 {'baz': 5}

```

Dans cet exemple, il n'est pas possible de fournir un argument positionnel au paramètre `bar`. `bar` est ce qu'on appelle un paramètre *keyword-only* (nommé uniquement).

VII.4.2.1. Opérateur *splat*

L'opérateur `*` utilisé dans la liste des paramètres est appelé *splat*, et ce n'est pas sa seule utilisation.

Il permet en effet aussi de réaliser l'opération inverse, celle de transmettre à une fonction les éléments d'une liste (ou autre itérable) comme arguments positionnels différents.

```

1 >>> def addition(a, b):
2     ...     return a + b
3     ...
4 >>> addition(*[1, 2])
5 3
6 >>> args = [1, 2]
7 >>> addition(*args)
8 3

```

`addition(*[1, 2])` est ainsi strictement équivalent à `addition(1, 2)`.

Et on voit que le *splat* du côté de l'appel n'est pas lié au *splat* dans la définition des paramètres puisque notre fonction n'accepte pas d'arguments variadiques ici. Mais les deux sont bien sûr compatibles.

```

1 >>> print_args(*[1, 2, 3])
2 (1, 2, 3) {}

```

Contrairement aux paramètres, rien ne nous empêche ici d'utiliser plusieurs *splats* pour envoyer des arguments de plusieurs listes, ni d'utiliser des arguments «normaux» en plus de nos listes.

```

1 >>> print_args(1, 2, *[3, 4, 5], 6)
2 (1, 2, 3, 4, 5, 6) {}
3 >>> print_args(*[1, 2, 3], 4, *[5, 6])
4 (1, 2, 3, 4, 5, 6) {}

```

De manière équivalente, `**` est l'opérateur *double-splat* et peut s'utiliser lors d'un appel pour transmettre le contenu d'un dictionnaire comme arguments nommés. Il est alors nécessaire que les clés du dictionnaire soient des chaînes de caractères (un nom de paramètre ne peut pas être autre chose qu'une chaîne).

```

1 >>> print_args(**{'foo': 0, 'bar': 'baz'})
2 () {'foo': 0, 'bar': 'baz'}

```

VII.4.3. Documentation et annotations

Je vous ai présenté plus tôt la fonction `help` qui permet d'obtenir des informations sur un module ou une fonction. Mais pour avoir accès à ces informations il faut que celles-ci aient été renseignées, que les fonctions aient été documentées.

C'est le cas dans la bibliothèque standard et c'est pourquoi nous obtenons des pages d'aide si complètes. Mais qu'en est-il de nos propres fonctions ?

Prenons cette fonction `operation` capable d'appliquer différentes opérations arithmétiques à deux valeurs.

VII. Aller plus loin

```
1 def operation(op, a, b):
2     if op == '+':
3         return a + b
4     elif op == '-':
5         return a - b
6     elif op == '*':
7         return a * b
8     elif op == '/':
9         return a / b
10    print('error')
11    return -1
```

La gestion d'erreurs est nulle mais ce n'est pas le sujet ici, nous y reviendrons plus tard. 🍊
Voyons pour le moment à quoi ressemble la page d'aide de notre fonction.

```
1 >>> help(operation)
2 Help on function operation in module __main__:
3
4 operation(op, a, b)
```

C'est... succinct. D'un côté, on n'a rien renseigné d'autre à notre fonction, et il serait difficile à Python de nous donner plus d'informations.

VII.4.3.1. Docstring

Une première étape vers la documentation est de rédiger une *docstring* dans notre fonction. Une *docstring* c'est simplement une chaîne de caractères placée au début de notre fonction, sans assignation ni rien.

```
1 def operation(op, a, b):
2     "Renvoie le résultat de l'opération a (op) b"
3     ...
```

Dans le déroulement de notre fonction ça ne change rien puisqu'une telle chaîne de caractères n'a aucun effet. Mais Python la détecte et la rend accessible comme documentation de notre fonction.

```
1 >>> help(operation)
2 Help on function operation in module __main__:
3
4 operation(op, a, b)
5     Renvoie le résultat de l'opération a (op) b
```

Mais la documentation ne se résume généralement pas en une ligne. Aussi, on trouvera souvent une chaîne multi-lignes délimitée par des triple-guillemets pour documenter notre fonction.

```
1 def operation(op, a, b):
2     """
3     Renvoie le résultat de l'opération a (op) b
4
5     Avec op l'un des opérateurs suivants :
6     +: addition
7     -: soustraction
8     *: multiplication
9     /: division
10    """
11    ...
```

Ne vous inquiétez pas pour les retours à la ligne introduits au début et à la fin de la chaîne, ils disparaissent dans la documentation.

```
1 >>> help(operation)
2 Help on function operation in module __main__:
3
4 operation(op, a, b)
5     Renvoie le résultat de l'opération a (op) b
6
7     Avec op l'un des opérateurs suivants :
8     +: addition
9     -: soustraction
10    *: multiplication
11    /: division
```

VII.4.3.2. Comment documenter ?

Dans notre documentation, on ne va pas définir en détails ce que fait notre fonction, expliquer toutes les conditions qui y sont faites, ça n'aurait pas trop d'intérêt. Non, le but d'une documentation est de décrire l'usage : comment utiliser notre fonction et à quoi s'attendre en retour.

Si notre fonction a des comportements particuliers (erreurs qu'elle gère ou pas, astuces d'optimisation, etc.), il est bon aussi de les indiquer dans la documentation.

Pour notre fonction `operation`, la documentation devrait indiquer que notre fonction réalise des opérations arithmétiques de 4 types (addition, soustraction, multiplication et division flottante) et renvoie le résultat de l'opération, l'opération à effectuer et les deux opérands étant récupérés depuis les paramètres.

Il serait ajouté que la fonction affiche un message d'erreur en cas d'opération inconnue (en renvoyant `-1`), et qu'elle ne traite pas l'erreur de division par zéro.

VII.4.3.3. Annotations de types

La *docstring* n'est pas l'unique manière de documenter une fonction, d'autres informations peuvent être apportées par les annotations de types. Comme leur nom l'indique, ces annotations servent à décrire les types des paramètres de la fonction.



Les annotations sont parfaitement facultatives, elles sont utiles à la documentation et pour des outils d'analyse statique (tel que `mypy` présenté en annexe [↗](#)). Elles existent et vous pouvez donc en rencontrer dans un code, c'est pourquoi je vous les présente, mais ne vous sentez pas obligé de les utiliser si vous n'en ressentez pas le besoin.

Notre fonction peut s'annoter simplement : le premier paramètre est une chaîne de caractère, et les deux suivants sont des nombres, que l'on va pour le moment considérer comme des `int`. Pour annoter un paramètre, on le fait suivre d'un `:` et du type que l'on veut préciser.

```
1 def operation(op: str, a: int, b: int):  
2     ...
```

Ces informations sont ajoutées à la signature de la fonction dans la documentation fournie par `help`.

```
1 >>> help(operation)  
2 Help on function operation in module __main__:  
3  
4 operation(op: str, a: int, b: int)  
5 ...
```

Il est aussi possible de préciser une annotation sur la fonction en elle-même pour indiquer le type de la valeur de retour. Pour cela, on utilise un `->` derrière la liste des paramètres, suivi du type de retour.

```
1 def operation(op: str, a: int, b: int) -> int:  
2     ...
```



Les annotations ne changent rien lors de l'exécution du programme, elles sont là à titre indicatif, la fonction peut être appelée avec d'autres types que ceux précisés sans que cela ne provoque d'erreur.

On le constate d'ailleurs si l'on appelle notre fonction avec des nombres flottants.

```
1 >>> operation('+', 1.2, 3.4)  
2 4.6
```

VII.4.3.4. Module `typing`

Cela nous pose tout de même un problème : notre fonction est documentée pour être utilisée avec des `int`, mais on aimerait pouvoir l'appeler avec des `float`. On pourrait la documenter avec des `float` mais se poserait alors le problème inverse.

Heureusement, il existe un module Python pour travailler et modeler ces annotations de types,

VII. Aller plus loin

le module `typing`. Celui-ci contient des outils qui vont nous être utiles pour préciser des cas plus complexes d'utilisation des types, comme ici avec le choix entre deux types.

Pour ce problème il existe donc `typing.Union`, un objet particulier qui comme son nom l'indique permet de créer des unions (au sens mathématique) de types. Il s'utilise à l'aide de crochets à l'intérieur desquels sont précisés les types autorisés.

Dans notre cas on aurait `typing.Union[int, float]`.

Cet objet définit une forme spéciale de typage, et s'utilise donc directement en tant qu'annotation.

Un paramètre annoté ainsi sera considéré comme pouvant être de n'importe lequel des types précisés.

```
1 import typing
2
3 def operation(op: str, a: typing.Union[int, float], b:
4     typing.Union[int, float]) -> typing.Union[int, float]:
5     ...
```

Ce type n'a pas pour but d'être instancié, et vous obtiendrez d'ailleurs une erreur si vous essayez de le faire. Il n'est utile que pour renseigner des annotations.

On notera qu'il est possible de créer un alias à notre type particulier, de façon à le renseigner plus facilement, tout simplement en l'assignant à une variable.

```
1 Number = typing.Union[int, float]
2
3 def operation(op: str, a: Number, b: Number) -> Number:
4     ...
```

i

En l'occurrence dans un cas comme celui-ci on utiliserait plutôt le type `Number` du module `numbers` présenté dans la partie suivante. Il est plus générique que notre solution avec `typing.Union` puisqu'il autorise aussi les complexes et d'autres types encore.

D'autres problèmes peuvent se poser lorsque l'on cherche à documenter les types d'une fonction. Par exemple prenons la fonction `my_sum` suivante, équivalente à la fonction `sum` de la bibliothèque standard, pour calculer une somme de nombres.

```
1 def my_sum(values, start=0):
2     "Calcule et renvoie les somme des éléments de `values`"
3     for value in values:
4         start += value
5     return start
```

Comment l'annoter de façon à préciser que l'on attend une liste de nombres comme premier argument ? On ne peut pas simplement utiliser `list` qui serait bien trop générique, autorisant des éléments d'autres types et mêmes des listes disparates (composées d'éléments de types différents).

`typing` vient à la rescousse en proposant un `typing.List` que l'on spécialise avec le type

VII. Aller plus loin

voulu, ici le type spécial `Number` défini plus haut.

```
1 def my_sum(values: typing.List[Number], start : typing.Number = 0)
  -> typing.Number:
2     ...
```

On pourrait aussi utiliser `typing.Iterable[Number]` qui aurait l'intérêt d'autoriser tout type d'itérable (tuple, range, etc.) et non uniquement les listes.

i

Depuis Python 3.9, le type `list` est directement spécialisable comme annotation de type, rendant `typing.List` obsolète. On peut ainsi simplement utiliser `list[int]` pour indiquer une liste de nombres entiers.

C'est le cas aussi pour les autres conteneurs de la bibliothèque standard tels `tuple` et `dict`.

Encore une fois, ces types particuliers n'ont pas pour but d'être instanciés et n'apporteraient aucune garantie sur leurs objets. Ils ne sont utiles qu'à des fins d'annotations.

Le module `typing` comprend de nombreuses choses, certaines dépassant le cadre de ce cours, et je ne vais donc pas m'appesantir sur sa présentation. Pour terminer, sachez simplement qu'il existe un type spécial `typing.Any`, pour préciser qu'un paramètre peut accepter une valeur de n'importe quel type.

```
1 def print(value: typing.Any):
2     ...
```

VII.4.4. Décorateurs

Les décorateurs sont faits pour décorer. Enfin pas exactement, il faut comprendre le terme comme «envelopper». Un décorateur, c'est un objet que l'on va appliquer à une fonction pour en changer le comportement.

Le décorateur est indépendant et extérieur à la fonction, il se contente de l'envelopper.

Lors des appels à notre fonction, c'est le décorateur qui prendra le pas et choisira les opérations à effectuer. Il pourra choisir d'appeler notre fonction ou non, d'exécuter des opérations avant ou après, etc.

Il s'agit de l'application directe du [patron de conception décorateur](#) [↗](#).

Par exemple, plus tôt dans ce chapitre nous avons vu un mécanisme de cache (mémoïsation) appliqué à une fonction, à l'aide de la valeur par défaut d'un paramètre. Ce mécanisme aurait pu être implémenté à l'aide d'un décorateur, celui-ci décidant si l'appel à la fonction est nécessaire ou non, en fonction de ce qu'il a dans son cache, puis il se chargerait d'y enregistrer les résultats reçus.

La syntaxe pour appliquer un décorateur à une fonction est très simple, il suffit de précéder la définition de notre fonction par une ligne composée d'un `@` et du nom du décorateur.

VII. Aller plus loin

```
1 @decorator
2 def function(a, b):
3     ...
```

En pratique, on notera que ce code est équivalent au suivant, l'application à l'aide de l'@ n'étant que du sucre syntaxique apporté par Python.

```
1 def function(a, b):
2     ...
3
4 function = decorator(function)
```

Le mécanisme de cache dont je parlais est fourni par le décorateur `lru_cache` du module `functools`. Appliqué à une fonction, il permettra donc de garder en mémoire les résultats de la fonction et éviter de réexécuter des calculs coûteux.

```
1 from functools import lru_cache
2
3 @lru_cache
4 def addition(a, b):
5     print(f'Calcul de {a} + {b}...')
6     return a + b
```

On le voit à l'utilisation, la fonction n'est appelée que si son résultat n'est pas déjà connu.

```
1 >>> addition(3, 5)
2 Calcul de 3 + 5...
3 8
4 >>> addition(1, 2)
5 Calcul de 1 + 2...
6 3
7 >>> addition(3, 5)
8 8
```

Un tel mécanisme par décorateur a l'intérêt de ne rien changer au code de la fonction, qui reste le même que si le décorateur n'était pas là.

VII.4.4.1. Décorateur paramétré

Mais que signifie ce *lru* dans `lru_cache` ? Il s'agit du sigle *Least Recently Used* (*Utilisés le Plus Récemment* en français) explicitant le comportement du cache.

En effet, votre ordinateur a une mémoire limitée et le cache cherche donc à minimiser son empreinte. Pour cela, il ne conservera pas tous les résultats en mémoire mais seulement ceux qu'il juge prioritaires. C'est là qu'intervient le mécanisme *LRU* qui signifie simplement que les résultats prioritaires sont ceux utilisés le plus récemment.

Cela signifie aussi que les résultats les plus anciens finiront par disparaître du cache lorsque

VII. Aller plus loin

celui-ci aura été rempli par d'autres valeurs.

La taille maximale par défaut du cache est de 128 résultats, mais il est possible d'en changer en paramétrant le décorateur.

Comme une fonction, un décorateur peut-être suivi de parenthèses contenant des arguments pour le paramétrer et donc influencer sur son comportement.

```
1 @decorator('foo', 5)
2 def function(a, b):
3     ...
```

Ici, la taille du cache est un paramètre du décorateur. Nous allons d'ailleurs voir le mécanisme *LRU* en action en réduisant la taille allouée à ce cache, disons à 3.

```
1 @lru_cache(3)
2 def addition(a, b):
3     print(f'Calcul de {a} + {b}...')
4     return a + b
```

Et maintenant, regardez bien ce qu'il se passe quand la taille maximale est atteinte.

```
1 >>> addition(3, 5)
2 Calcul de 3 + 5...
3 8
4 >>> addition(1, 2)
5 Calcul de 1 + 2...
6 3
7 >>> addition(4, 7)
8 Calcul de 4 + 7...
9 11
10 >>> addition(3, 5) # la valeur est toujours en cache
11 8
12 >>> addition(9, 6)
13 Calcul de 9 + 6...
14 15
15 >>> addition(1, 2) # la valeur est sortie du cache
16 Calcul de 1 + 2...
17 3
```

Notre cache est limité à 3 résultats, pour enregistrer **9 + 6** il doit donc faire de la place. Le résultat le plus anciennement utilisé, ici **1 + 2**, est donc supprimé.

Attention, je dis bien plus anciennement utilisé et non calculé. Car le résultat le plus anciennement calculé est **3 + 5**, mais on l'a redemandé par la suite à la fonction, signifiant au cache qu'il était à nouveau utilisé. Le résultat supprimé est celui auquel on a accédé le moins récemment.

Il est aussi parfaitement possible de se passer de ce mécanisme *LRU* et donc de conserver tous les résultats sans limite de taille (autre que celle de l'ordinateur), en spécifiant **None** comme taille maximale au décorateur.

```

1 @lru_cache(None)
2 def addition(a, b):
3     print(f'Calcul de {a} + {b}...')
4     return a + b

```

i

On notera que depuis Python 3.9 il existe aussi le décorateur `cache` dans le module `functools` remplissant le rôle de cache illimité (`lru_cache(None)`).

```

1 from functools import cache
2
3 @cache
4 def addition(a, b):
5     print(f'Calcul de {a} + {b}...')
6     return a + b

```

VII.4.5. Fonctions lambdas

Les conditions peuvent être des expressions, les boucles (`for`) peuvent être des expressions, les définitions de fonction peuvent aussi être des expressions.

Je dis bien les définitions car les fonctions en elles-mêmes, nous l'avons vu précédemment, sont déjà des valeurs et donc des expressions à part entière.

```

1 >>> def add_one(n):
2     ...     return n + 1
3     ...
4 >>> add_one
5 <function add_one at 0x7f662ecf61f0>
6 >>> x = add_one
7 >>> x(5)
8 6

```

Leur définition, c'est le bloc `def` qui définit leur nom, leurs paramètres et leur code ; lui ne peut pas être assigné à une variable ou passé en argument.

La solution se trouve du côté des fonctions lambdas (ou «fonctions anonymes») introduites par le mot-clé `lambda`. Ce mot-clé, suivi d'un `:` et d'une expression permet de définir une fonction sans nom, l'expression étant le code de la fonction.

Une fonction lambda ne peut alors être composée que d'une unique expression.

```

1 >>> lambda: 42
2 <function <lambda> at 0x7f616ffe93a0>

```

Il s'agit d'une fonction à part entière, et si nous l'appelons nous obtenons bien 42 comme

réponse.

```
1 >>> (lambda: 42) ()
2 42
```



Les parenthèses autour de la lambda sont nécessaires pour la gestion des priorités, `lambda: 42()` serait compris comme `lambda: (42())` et n'aurait pas de sens.

La lambda est une expression et peut donc être assignée à une variable.

```
1 >>> get_42 = lambda: 42
2 >>> get_42
3 <function <lambda> at 0x7f616ffe9430>
4 >>> get_42()
5 42
```

Tout comme les fonctions, les lambdas peuvent recevoir des paramètres en tous genres, il suffit pour cela de préciser la liste des paramètres avant le signe `:`.

```
1 >>> addition = lambda a, b: a + b
2 >>> addition(3, 5)
3 8
```

Mais l'intérêt principal des fonctions comme expressions réside dans le fait de pouvoir être passées comme arguments à d'autres fonctions. Souvenez-vous par exemple de la fonction `sorted` et de son paramètre `key` recevant une fonction.

Avec une lambda, il n'est pas nécessaire de définir une fonction au préalable : on peut directement passer l'expression de tri sous forme de lambda à la fonction.

Voici par exemple un tri de mots ne tenant pas compte de la casse (différence entre lettres minuscules et capitales), en s'appuyant sur la conversion en minuscules des chaînes.

```
1 >>> words = ['poire', 'Ananas', 'banane', 'abricot', 'FRAISE']
2 >>> sorted(words, key=lambda w: w.lower())
3 ['abricot', 'Ananas', 'banane', 'FRAISE', 'poire']
```

VII.4.6. Fonctions récursives

Deux grands modèles s'opposent en informatique lorsqu'il est question de répéter des tâches : le modèle itératif et le modèle récursif.

Le modèle itératif, nous le connaissons, c'est celui des boucles. On place notre tâche dans une boucle et celle-ci sera donc répétée un certain nombre de fois.

Le modèle récursif est assez différent dans sa conception, il repose sur des fonctions. L'idée étant que la fonction s'appelle elle-même, provoquant ainsi une répétition, on parle alors de fonction

VII. Aller plus loin

récursive.

En mode itératif, marcher c'est mettre un pied devant l'autre et recommencer. En mode récursif, marcher c'est mettre un pied devant l'autre et marcher.

<https://twitter.com/framaka/status/1327220641150496768> ↗

C'est un concept issu des mathématiques qui se définit assez bien et intuitivement, nous l'appliquons même généralement sans le savoir.

Prenons par exemple la somme d'une liste de N nombres : de quoi s'agit-il ? Simplement de l'addition entre le premier nombre de la liste et la somme des $N-1$ autres nombres.

Par exemple `sum([1, 2, 3, 4, 5])` est égal à `1 + sum([2, 3, 4, 5])`, `sum([2, 3, 4, 5])` à `2 + sum([3, 4, 5])` et ainsi de suite.

Nous venons de définir la somme de manière récursive. On a réduit une opération complexe (la somme de N nombres) à une succession d'opérations plus simples (une addition entre deux nombres) et un procédé récursif (exécuter à nouveau la procédure sur un ensemble restreint).

En Python, cela donnerait le code suivant.

```
1 def my_sum(numbers):
2     return numbers[0] + my_sum(numbers[1:])
```

Mais on remarque tout de suite un problème, on ne sait pas quand ça va s'arrêter. Cette fonction va-t-elle même s'arrêter ? En l'occurrence oui, mais en provoquant une erreur.

```
1 >>> my_sum([1, 2, 3, 4, 5])
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in my_sum
5   File "<stdin>", line 2, in my_sum
6   File "<stdin>", line 2, in my_sum
7   [Previous line repeated 3 more times]
8 IndexError: list index out of range
```

En effet, nos appels récursifs finissent par appeler `my_sum([])` qui échoue car n'a pas de premier élément (`numbers[0]`).

Le soucis c'est que c'est à nous de gérer explicitement la condition de fin et que nous ne l'avons pas fait. Ici la condition de fin est facilement identifiable, la somme des nombres d'une liste vide est toujours nulle. On devrait donc, dans le cas où l'on rencontre une liste vide, renvoyer directement zéro.

Nous pouvons ajouter cette condition à notre fonction récursive et constater que le comportement est alors bon.

```
1 >>> def my_sum(numbers):
2     ...     if not numbers:
3     ...         return 0
4     ...     return numbers[0] + my_sum(numbers[1:])
5     ...
```

VII. Aller plus loin

```
6 >>> my_sum([1, 2, 3, 4, 5])
7 15
```

Mais il y a des cas où la condition de fin est moins évidente à trouver, cela pouvant mener à une récursion infinie.

VII.4.6.1. Récursion infinie

Notre premier cas ne possédait pas de condition de fin mais s'est arrêté en raison d'une `IndexError`. Que ce serait-il passé sans cette erreur ?

On peut prendre un cas assez similaire qui est de calculer la taille d'une chaîne de caractères. Récursivement, la taille d'une chaîne se conçoit comme l'addition entre deux tailles de sous-chaînes, par exemple entre la taille du premier caractère (1) et la taille du reste.

On a `len('abcdef')` égal à `1 + len('bcdef')`.

```
1 def my_len(s):
2     return 1 + my_len(s[1:])
```

Là encore, nous avons oublié de prévoir la condition de fin (renvoyer 0 sur une chaîne vide), et patatra !

```
1 >>> my_len('abcdef')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 2, in my_len
5   File "<stdin>", line 2, in my_len
6   File "<stdin>", line 2, in my_len
7   [Previous line repeated 996 more times]
8 RecursionError: maximum recursion depth exceeded
```

Cette fois ce n'est pas une erreur dans notre code qui nous arrête, mais une erreur de Python lui-même qui nous indique que nous avons dépassé le nombre maximum de récursions. Nous sommes entrés dans une récursion infinie.

En fait, chaque appel récursif occupe un peu de mémoire dans notre programme, pour stocker le contexte de la fonction (les arguments qui lui sont passés par exemple). Quand nous empilons les appels récursifs, la mémoire utilisée croît, jusqu'à atteindre une limite. En Python c'est l'interpréteur qui fixe arbitrairement une limite de 1000 appels.



Certains langages (les langages fonctionnels notamment) mettent en œuvre des optimisations pour supprimer cette limite, mais ce n'est pas le cas de Python qui est assez peu porté sur le modèle récursif.

VII.4.6.2. Récursions croisées

La récursivité ne se limite pas à une fonction seule, il est aussi possible de croiser des fonctions qui s'appelleraient les unes les autres.

VII. Aller plus loin

Par exemple, comment déterminer si un nombre `n` est impair ? En regardant si `n-1` est pair ! Et pour savoir si `n-1` est pair on teste si `n-2` est impair. On répète cela jusqu'à zéro que l'on sait pair (et donc non impair).

En Python, cela nous donnerait les deux fonctions suivantes.

```
1 def odd(n): # impair
2     if n == 0:
3         return False
4     return even(n - 1)
5
6 def even(n): # pair
7     if n == 0:
8         return True
9     return odd(n - 1)
```

Qui se comportent bien comme on veut pour calculer la parité des nombres.

```
1 >>> odd(5)
2 True
3 >>> even(5)
4 False
5 >>> odd(4)
6 False
7 >>> even(4)
8 True
```

Bien sûr je ne présente ces fonctions qu'à titre d'exemple. Les fonctions récursives étant déjà assez rares en Python pour les raisons expliquées plus haut, les récursions croisées le sont encore plus.

Et l'exemple présenté ci-dessus est particulièrement inefficace (on peut directement tester la parité d'un nombre avec `n % 2`).

Aussi, préférez dans la mesure du possible opter pour des solutions itératives.

VII.5. Retour sur les variables

VII.5.1. Expressions d'assignation

On a vu que pas mal d'instructions que l'on utilisait pouvaient être remplacées par des expressions. Qu'en est-il des assignations de variables ?

Il existe aussi des expressions d'assignation, apportées par la version 3.8 de Python, ce qui est assez récent. Elles découlent d'un besoin de pouvoir assigner des variables à des endroits où ce n'était pas possible avant, comme dans des conditions de `if`.

Imaginons par exemple que nous souhaitions tester successivement plusieurs paramètres pour définir une valeur. Disons que nous ayons une configuration où plusieurs paramètres pourraient servir à définir une même valeur dans des conditions différentes (définir depuis un chemin de fichier, définir depuis une URL, etc.) et que par commodité nous souhaitions nettoyer les paramètres en supprimant les espaces de début et de fin.

Avec des assignations simples, il nous faut imbriquer plusieurs conditions les unes dans les autres.

```
1 >>> config = {'path': ' ', 'text': 'toto'}
2 >>> url = config.get('url', '').strip()
3 >>> if url:
4 ...     uri = url
5 ... else:
6 ...     path = config.get('path', '').strip()
7 ...     if path:
8 ...         uri = f'file://{path}'
9 ...     else:
10 ...         text = config.get('text', '').strip()
11 ...         if text:
12 ...             uri = f'data:text/plain;charset=utf-8,{text}'
13 ...         else:
14 ...             uri = None
15 ...
16 >>> uri
17 'data:text/plain;charset=utf-8,toto'
```

Les expressions d'assignation permettent de simplifier cela. Elles sont introduites avec l'opérateur `:=` aussi appelé opérateur «walrus» (pour sa ressemblance avec un morse si on penche la tête vers la gauche).

`a := foo` devient alors une expression assignant la valeur `foo` à la variable `a`.

Pour reprendre l'exemple précédent, on pourrait alors le réécrire comme suit :

VII. Aller plus loin

```
1 >>> config = {'path': ' ', 'text': 'toto'}
2 >>> if url := config.get('url', '').strip():
3 ...     uri = url
4 ... elif path := config.get('path', '').strip():
5 ...     uri = f'file://{path}'
6 ... elif text := config.get('text', '').strip():
7 ...     uri = f'data:text/plain;charset=utf-8,{text}'
8 ... else:
9 ...     uri = None
10 ...
11 >>> uri
12 'data:text/plain;charset=utf-8,toto'
```

Dans cet exemple, `url := config.get('url', '').strip()` assigne la variable `url` puis la teste pour savoir si on entre dans le bloc conditionnel ou non.

L'opérateur `:=` est l'un des moins prioritaires de Python, ainsi il sera souvent nécessaire de placer l'expression entre parenthèses pour la prioriser.

```
1 >>> if (x := round(3.5)) > 0:
2 ...     print(x)
3 ...
4 4
```

Là où sans parenthèses, la condition aurait été évaluée comme `x := (round(3.5) > 0)` donc `x` aurait été un booléen.

```
1 >>> if x := round(3.5) > 0:
2 ...     print(x)
3 ...
4 True
```



Il est cependant à noter que `:=` est prioritaire par rapport à la virgule. Ainsi, `x := 1, 2` équivaut à `(x := 1), 2` et non à `x := (1, 2)`.

Enfin, pour éviter toute confusion avec l'opérateur `=`, les expressions d'assignation ne sont pas autorisées là où les instructions sont autorisées. Il n'est ainsi pas possible d'écrire `foo := 'bar'` directement dans l'interpréteur Python.

```
1 >>> foo := 'bar'
2 File "<stdin>", line 1
3     foo := 'bar'
4             ^
5 SyntaxError: invalid syntax
```

Mais on peut placer l'expression entre parenthèses pour lever la confusion et la rendre valide.

```
1 >>> (foo := 'bar')
2 'bar'
3 >>> foo
4 'bar'
```

On remarque d'ailleurs bien dans cet exemple que l'assignation est une expression, puisque la ligne `(foo := 'bar')` a été évaluée comme valant `'bar'`.

VII.5.2. Annotations de types

Tout comme il est possible de typer les paramètres de fonction à l'aide d'annotations, de telles annotations sont disponibles aussi pour typer nos variables. Là encore les annotations n'ont qu'un rôle purement indicatif (pour de la documentation) ou peuvent être analysées par des outils externes comme `mypy`.

Pour annoter une variable avec un type, on place simplement un signe `:` suivi du type avant le signe `=` de l'assignation.

```
1 >>> foo: str = 'bar'
```

Il est aussi possible d'annoter une variable sans la définir, en faisant juste suivre un nom de variable d'une annotation, sans assignation.

```
1 >>> number: int
```



Attention, dans ce cas la variable est annotée mais n'a pas de valeur. `number` reste une variable indéfinie.

```
1 >>> number
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   NameError: name 'number' is not defined
```

On peut alors la définir plus tard, sans avoir besoin de repréciser l'annotation de type qui sera conservée pour la variable.

```
1 >>> number = 42
2 >>> number
3 42
```

VII.5.3. Scopes

Je vais être assez bref sur ce sujet car je ne souhaite pas vous inonder d'informations compliquées, mais je pense qu'il est temps de parler un peu des scopes.

On a vu plus tôt que les fonctions définissaient un espace de noms (un **scope**) : les variables définies dans une fonction n'existent pas à l'extérieur. L'inverse n'est pas vrai, les mécanismes de Python permettent d'accéder depuis une fonction à une variable définie à l'extérieur.

```
1 >>> x = 5
2 >>>
3 >>> def get_x():
4 ...     print('x vaut', x)
5 ...
6 >>> get_x()
7 x vaut 5
```

On a vu aussi qu'il était possible depuis une fonction de définir une variable locale du même nom qu'une variable extérieure, sans que cela ne provoque d'erreur ou d'interférence.

```
1 >>> def set_x():
2 ...     x = 12
3 ...     print('x vaut', x)
4 ...
5 >>> set_x()
6 x vaut 12
7 >>> x
8 5
```

On a vu enfin que cela pouvait poser problème si l'on tente d'accéder à une variable extérieure avant de la redéfinir, Python croyant avoir affaire à une variable locale qui n'a pas encore été définie.

```
1 >>> def set_x():
2 ...     print('x vaut', x)
3 ...     x = 12
4 ...
5 >>> set_x()
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "<stdin>", line 2, in set_x
9 UnboundLocalError: local variable 'x' referenced before assignment
```

Ces points mènent à une question : est-il possible de redéfinir une variable extérieure depuis une fonction Python ? La réponse est oui, mais à l'aide de mots-clés spécifiques.

VII.5.3.1. Variables globales

Dans notre exemple, `x` est appelée une variable globale, car elle est définie à la racine (au plus haut espace de noms) du module courant, et est donc accessible dans tout le module. Pour redéfinir une variable globale depuis une fonction du module, il nous faut la déclarer au sein de la fonction. La déclarer c'est indiquer à Python que la variable existe quelque part et lui permettre de la retrouver, afin de la distinguer d'une variable locale.

Ici, on va utiliser le mot-clé `global` pour indiquer que la variable existe dans le scope global du module. Après, il nous sera possible de l'utiliser comme une variable locale, et donc même de la redéfinir.

```

1  >>> def set_x():
2  ...     global x
3  ...     x = 12
4  ...     print('x vaut', 12)
5  ...
6  >>> x
7  5
8  >>> set_x()
9  x vaut 12
10 >>> x
11 12

```

Bien sûr, l'instruction `global x` doit être exécutée dans la fonction avant toute utilisation de la variable `x`.

`global` peut aussi être utilisé avec une variable n'existant pas encore dans le scope global. Mais cela signifie à Python que c'est dans ce scope qu'il faudra la définir.

```

1  >>> def set_y():
2  ...     global y
3  ...     y = 17
4  ...
5  >>> set_y()
6  >>> y
7  17

```



L'utilisation de `global` induit naturellement des effets de bord et rend le flux d'exécution du programme plus difficile à suivre. Dans la mesure du possible, évitez donc de l'utiliser à moins d'être sûrs de ce que vous faites.

On notera aussi que le mot-clé `global` n'est utile que pour redéfinir une variable globale, il n'est pas nécessaire pour `y` accéder ni même pour la modifier.

```

1  >>> items = []
2  >>>
3  >>> def add_item(value):

```

VII. Aller plus loin

```
4 ...     items.append(value)
5 ...     print(items)
6 ...
7 >>> add_item(3)
8 [3]
9 >>> add_item(5)
10 [3, 5]
11 >>> add_item(8)
12 [3, 5, 8]
```

Il faut bien sûr pour cela que la valeur en question soit modifiable (comme une liste dans l'exemple ci-dessus), ça ne pourrait pas fonctionner avec un nombre ou une chaîne de caractères par exemple.

Mais il reste dans ces cas-là possible de passer par un conteneur intermédiaire modifiable tel qu'un dictionnaire.

```
1 >>> values = {'number': 12, 'string': 'salut'}
2 >>>
3 >>> def reset(number, string):
4 ...     values['number'] = number
5 ...     values['string'] = string
6 ...
7 >>> reset(-8, 'pouet')
8 >>> values
9 {'number': -8, 'string': 'pouet'}
```

VII.5.3.2. Fonctions imbriquées

En Python les fonctions sont des valeurs de premier ordre, c'est-à-dire des valeurs à part entière comme le sont les nombres ou les chaînes de caractères. Il est donc possible de les manipuler, de les mettre dans des variables, d'utiliser ces variables comme des fonctions etc.

```
1 >>> func = print
2 >>> func('toto')
3 toto
```

Et donc naturellement, il est aussi possible de renvoyer des fonctions depuis des fonctions.

```
1 >>> def get_print():
2 ...     return print
3 ...
4 >>> p = get_print()
5 >>> p('toto')
6 toto
7 >>> get_print()('tata')
8 tata
```

VII. Aller plus loin

Mais mieux encore, il est possible de définir des fonctions au sein d'autres fonctions. Ces fonctions seront comme toutes les variables locales, perdues si elles ne sont pas renvoyées par la fonction mère.

```
1 >>> def get_print():
2 ...     def special_print(*args):
3 ...         print(':', *args)
4 ...     return special_print
5 ...
6 >>> p = get_print()
7 >>> p('toto')
8 : toto
```

VII.5.3.3. Variables non-locales

Le sujet des scopes ne se résume donc pas aux variables locales et globales : avec les fonctions imbriquées, les scopes s'imbriquent eux aussi. Il existe alors des variables intermédiaires qui ne sont ni locales (appartenant au scope le plus bas) ni globales (scope le plus haut), que l'on appelle variables non-locales.

```
1 def add_by(a):
2     def inner(b):
3         return a + b
4     return inner
```

Cet exemple permet de générer des fonctions pour additionner par un nombre en particulier.

```
1 >>> add_by_3 = add_by(3)
2 >>> add_by_3(5)
3 8
4 >>> add_by_3(10)
5 13
```

Dans l'exemple, `b` est une variable locale à la fonction `inner`, et `a` lui est une variable non-locale. Mais la définition dépend toujours du scope dans lequel on regarde : `a` est aussi une variable locale de `add_by`.

Et d'ailleurs, une variable globale n'est rien de plus qu'une variable locale à un module.

De la même manière que pour les globales, il est possible de redéfinir les variables non-locales. Mais cela dépasse le cadre de ce cours, et je vous conseille alors de vous diriger vers mon tutoriel [Variables, scopes et closures en Python](#) si vous souhaitez en apprendre plus sur ces variables, et découvrir les mécanismes de capture.

VII.6. Retour sur les exceptions

VII.6.1. Bloc `except`

Précédemment nous avons vu le bloc `except` qui, associé à un `try`, permet de traiter une exception qui surviendrait au cours de l'exécution.

```
1 def get_10th(seq):
2     try:
3         return seq[10]
4     except IndexError:
5         return None
```

```
1 >>> get_10th('abcdefghijkl')
2 'k'
3 >>> get_10th('abcd')
```

L'idée étant que le contenu du `try` peut lever une erreur qui sera attrapée par le bloc `except` si son type correspond (`IndexError` ici).

Plusieurs blocs `except` peuvent être placés à la suite sur des types d'erreurs différents pour leur offrir un traitement particulier.

```
1 def get_10th(seq):
2     try:
3         return seq[10]
4     except IndexError:
5         return None
6     except KeyError:
7         return None
```

```
1 >>> get_10th({5: 'a', 10: 'b'})
2 'b'
3 >>> get_10th({})
```

Mais on le voit ici, il peut arriver que le traitement soit le même pour différentes erreurs.

Dans ce cas, il est possible de spécifier les différents types d'exceptions au sein d'une même clause `except`, simplement en les plaçant dans un tuple.

```
1 def get_10th(seq):
2     try:
3         return seq[10]
4     except (IndexError, KeyError):
5         return None
```

VII.6.1.1. Données complémentaires des exceptions

Une exception possède certes un type pour expliciter la cause de l'erreur, mais d'autres informations complémentaires sont aussi accessibles.

En effet, une exception n'est rien d'autre qu'un objet Python, qui possède donc des attributs et des méthodes. Pour récupérer l'objet de cette exception, il suffit de placer un `as nom_de_la_variable` derrière le `except` afin de l'affecter à une variable `nom_de_la_variable`.

Variable que l'on a tendance à appeler `error` / `exception`, ou plus simplement `err`, `exc` ou `e`.

```
1 >>> seq = []
2 >>> try:
3     ...     seq[10]
4     ... except IndexError as e:
5     ...     print(e)
6     ...
7 list index out of range
```

On voit qu'ici dans le cas d'une `IndexError`, l'exception contient un message nous expliquant la raison de l'erreur (l'index choisi est en dehors des bornes).

Ce message est un argument de l'exception, il est accessible via son attribut `args`.

```
1 >>> try:
2     ...     seq[10]
3     ... except IndexError as e:
4     ...     print(e.args)
5     ...     msg, = e.args
6     ...     print(msg)
7     ...
8 ('list index out of range',)
9 list index out of range
```

Dans le cas d'une `KeyError` (clé invalide sur un dictionnaire), l'argument de l'erreur est simplement la clé.

```
1 >>> dic = {}
2 >>> try:
3     ...     dic['abc']
```

```
4 ... except KeyError as e:
5 ...     print(e.args)
6 ...
7 ('abc',)
```

Et c'est ce qui peut être un peu difficile avec le traitement des erreurs : chaque type d'exception présente des métadonnées qui lui sont propres, sans qu'il n'y ait forcément beaucoup de cohérence entre les types.

On notera que le `as ...` est aussi possible quand un tuple de types est précisé au `except` et s'utilise de la même manière.

```
1 def get_10th(seq):
2     try:
3         return seq[10]
4     except (IndexError, KeyError) as e:
5         return e.args
```

```
1 >>> get_10th([])
2 ('list index out of range',)
3 >>> get_10th({})
4 (10,)
```

VII.6.2. Autres mots-clés

Le mot-clé `try` ne s'accompagne pas uniquement de `except`. D'autres blocs sont aussi disponibles pour réagir à différents types de situations.

VII.6.2.1. `else`

Par exemple, le bloc `else` permet de traiter le cas où tout s'est bien passé et qu'aucune exception n'a été levée (attrapée ou non).

```
1 def get_10th(seq):
2     try:
3         seq[10]
4     except IndexError:
5         print("erreur d'index")
6     else:
7         print("pas d'erreur")
```

```
1 >>> get_10th([])
2 erreur d'index
3 >>> get_10th({})
```

VII. Aller plus loin

```
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "<stdin>", line 3, in get_10th
7   KeyError: 10
8 >>> get_10th([1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89])
9 pas d'erreur
10 >>> get_10th({10: 'foo'})
11 pas d'erreur
```

Cela est utile dans le cas d'une action qui dépendrait d'un traitement précédent, par exemple voici comment on pourrait implémenter la méthode `pop` des dictionnaires.

Pour rappel, cette méthode permet de supprimer une clé d'un dictionnaire et d'en renvoyer la valeur, et permet de renvoyer une valeur par défaut si la clé n'existe pas (ce que nous ferons par défaut dans notre implémentation).

```
1 def dict_pop(dic, key, default=None):
2     try:
3         value = dic[key]
4     except KeyError:
5         value = default
6     else:
7         del dic[key]
8     return value
```

```
1 >>> dic = {'a': 42}
2 >>> dict_pop(dic, 'a')
3 42
4 >>> dic
5 {}
6 >>> dict_pop(dic, 'a')
7 >>> dict_pop(dic, 'a', 'pouet')
8 'pouet'
```

VII.6.2.2. finally

Le bloc `finally` permet lui de réagir dans tous les cas, qu'une erreur soit survenue ou non, qu'elle ait été attrapée ou non.

```
1 def get_10th(seq):
2     try:
3         seq[10]
4     except IndexError:
5         print("erreur d'index")
6     finally:
7         print("traitement final")
```

VII. Aller plus loin

```
1 >>> get_10th([])
2 erreur d'index
3 traitement final
4 >>> get_10th({})
5 traitement final
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "<stdin>", line 3, in get_10th
9   KeyError: 10
10 >>> get_10th([1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89])
11 traitement final
12 >>> get_10th({10: 'foo'})
13 traitement final
```

On l'utilise par exemple pour la libération d'une ressource qui aurait été acquise avant le `try`¹.

```
1 def read_int(path):
2     f = open(path)
3     try:
4         return int(f.read())
5     finally:
6         print('Fermeture')
7         f.close()
```

Par exemple avec les fichiers suivants :

```
1 salut
```

Listing 65 – hello.txt

```
1 123
```

Listing 66 – number.txt

On constate bien que l'appel à `close` se fait dans tous les cas, même si une erreur survient.

```
1 >>> read_int('number.txt')
2 Fermeture
3 123
4 >>> read_int('hello.txt')
5 Fermeture
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
```

1. Même si l'on verra plus généralement un bloc `with` dans ce cas, qui permet de faire la même chose sans se prendre la tête.

VII. Aller plus loin

```
8 File "<stdin>", line 4, in read_int
9 ValueError: invalid literal for int() with base 10: 'salut\n'
```

On remarque aussi que le `finally` est exécuté même si un `return` est présent, il s'agit simplement d'un code exécuté à la toute fin de la fonction, mais qui n'en change pas la valeur de retour.

Attention, cela est bien sûr différent de placer un traitement à l'extérieur du bloc d'exception, qui lui ne sera pas exécuté en cas d'exception non attrapée.

```
1 def get_10th(seq):
2     try:
3         seq[10]
4     except IndexError:
5         print("erreur d'index")
6     finally:
7         print("traitement final")
8     print("Fin de la fonction")
```

```
1 >>> get_10th([])
2 erreur d'index
3 traitement final
4 Fin de la fonction
5 >>> get_10th({})
6 traitement final
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   File "<stdin>", line 3, in get_10th
10 KeyError: 10
```

VII.6.3. Bloc with

Nous avons vu les gestionnaires de contexte (blocs `with`) plus tôt, quand nous apprenions à utiliser les fichiers. Permettant de gérer l'acquisition/libération de ressources, ils sont en fait une autre manière de traiter les exceptions en Python.

```
1 with open('hello.txt') as f:
2     print(int(f.read()))
```

Dans le code précédent, même si la ligne 2 échoue (si le fichier ne contient pas un nombre), le fichier sera correctement fermé (Python appellera `f.close()` pour nous).

Car c'est ce que garantit le bloc `with` : assurer que le code de libération de la ressource sera toujours appelé¹.

1. Un gestionnaire de contexte se compose en fait d'une fonction pour initialiser la ressource et d'une autre pour la libérer, comme expliqué dans [ce cours](#) [↗](#), qui nécessite des notions de [programmation objet](#) [↗](#) en Python.

VII. Aller plus loin

En cela, il s'apparente à un `try / finally`, puisqu'il s'agit d'exécuter une action pour acquérir la ressource (avant le `try`) puis pour la libérer (dans le `finally`).

Mais on n'a pas à faire d'appel explicite à `f.close()` pour fermer notre fichier, tout cela est fait de façon transparente.

Le bloc précédent est alors équivalent à :

```
1 f = open('hello.txt')
2 try:
3     print(int(f.read()))
4 finally:
5     f.close()
```

VII.6.3.1. Supprimer une exception

En plus de ça, le bloc `with` peut aussi influencer sur la remontée d'exceptions, et donc stopper une exception qui serait levée à l'intérieur du bloc.

C'est ce que permet facilement le gestionnaire de contexte `suppress` du module `contextlib` de la bibliothèque standard.

Il s'utilise en précisant les types d'erreurs que l'on veut voir supprimés.

```
1 >>> from contextlib import suppress
2 >>> with suppress(ValueError):
3 ...     print(int('abc'))
4 ...
```

Plusieurs types peuvent être donnés en arguments pour tous les supprimer.

```
1 >>> with suppress(ValueError, TypeError):
2 ...     print(1 + 'b')
3 ...
```

Il ne permet pas de traitement plus avancé que ça, et se limite bien sûr à n'attraper que les erreurs des types spécifiés.

```
1 >>> with suppress(ValueError):
2 ...     print(1 + 'b')
3 ...
4 Traceback (most recent call last):
5   File "<stdin>", line 2, in <module>
6   TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

VII.6.4. Lever une exception

Les exceptions ont deux faces. D'un côté il s'agit de les attraper pour faire un traitement correct des erreurs, ce qui était l'objet des précédentes parties.

Mais de l'autre il est aussi question de lever des exceptions pour signaler les erreurs.

Souvenez-vous de notre factorielle qui ne gérait pas correctement les nombres négatifs en entrée, ce qui pouvait mener à des bugs¹.

La factorielle d'un nombre négatif n'a pas de sens et notre fonction ne devrait même pas les accepter. Elle devrait lever une exception quand un tel nombre lui est donné, pour que l'appelant sache que la valeur passée est problématique.

Cela se fait avec le mot-clé `raise`. Celui-ci peut simplement être suivi du type de l'exception à lever. Il a pour effet de lever immédiatement l'exception voulue, et donc de couper tout traitement en cours.

L'exception remontera ensuite la pile d'exécution du programme jusqu'à être attrapée.

```
1 def factorielle(n):
2     if n < 0:
3         raise ValueError
4
5     ret = 1
6     for i in range(2, n + 1):
7         ret *= i
8     return ret
```

Nous utilisons ici une `ValueError` pour signaler qu'il s'agit d'un problème avec la valeur en elle-même. Lors de l'appel, nous obtenons bien une exception `ValueError` en cas de valeur invalide.

```
1 >>> factorielle(5)
2 120
3 >>> factorielle(-1)
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "<stdin>", line 3, in factorielle
7   ValueError
```

Mais l'erreur n'est pas très explicite. Nous savons par le type d'erreur qu'il est question de la valeur, mais aucune autre information ne nous est donnée.

Parce que lors du `raise` nous avons simplement précisé un type sans plus d'informations.

Il est en fait possible d'appeler un type d'exception pour l'instancier, en lui donnant les arguments que l'on veut (généralement un message d'erreur), et d'utiliser cette instance pour le `raise`. Les arguments seront accessibles via l'attribut `args` de l'exception reçue comme nous l'avons vu précédemment, et affichés si l'erreur est imprimée à l'écran.

Ainsi, on peut modifier notre `raise` pour ajouter à l'exception un message d'erreur.

1. Voir chapitre [Boucler sur une condition \(while\)](#) ↗.


```
1 if n < 0:
2     raise ValueError('Le nombre doit être positif')
```

```
1 >>> factorielle(-1)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 3, in factorielle
5 ValueError: Le nombre doit être positif
```

On peut aller encore plus loin et générer un message d'erreur précis en ajoutant d'autres informations.

```
1 if n < 0:
2     raise ValueError(f_
    'Le nombre doit être positif ({n} est négatif)')
```

```
1 >>> factorielle(-1)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "<stdin>", line 3, in factorielle
5 ValueError: Le nombre doit être positif (-1 est négatif)
```

VII.6.4.1. Hiérarchie des exceptions

Nous avons rencontré plusieurs types d'exceptions pour coller à différentes situations. Il faut savoir que ces types sont hiérarchisés, afin de pouvoir traiter plus ou moins finement les erreurs qui surviennent.

Ainsi, faire un `except` sur un type d'exception arrêtera les exceptions de ce type mais aussi de tous les types qui en descendent.

Par exemple, toutes les exceptions que nous avons vues descendent d'un même type `Exception` : cela signifie qu'il suffit d'attraper `Exception` pour les attraper toutes.²

`TypeError` et `ValueError` sont alors deux des principales exceptions, la première indiquant une erreur dans le type des données et la seconde sur la valeur elle-même (le type correspond mais la valeur est incohérente). `ValueError` rassemble aussi des exceptions plus précises telles que `UnicodeDecodeError` et `UnicodeEncodeError` que nous avons déjà rencontrées.

`IndexError` et `KeyError` que l'on a beaucoup utilisées dans ce chapitre descendent d'une même exception `LookupError` qui attrape donc toutes les erreurs liées à la recherche dans un conteneur.

2. Allez dire ça au Professeur Chen, il sera vert.

VII. Aller plus loin

```
1 def get_10th(seq):
2     try:
3         return seq[10]
4     except LookupError as e:
5         print('erreur', e)
```

```
1 >>> get_10th([])
2 erreur list index out of range
3 >>> get_10th({})
4 erreur 10
```

On trouve aussi une grande famille d'erreurs sous `OSError` qui regroupe toutes les exceptions liées aux entrées/sorties, comme `FileNotFoundError`, `FileExistsError` ou `PermissionError`.

Voici un bref aperçu de cette hiérarchie :

```
1 Exception
2 +-- TypeError
3 +-- ValueError
4 |   +-- UnicodeError
5 |       +-- UnicodeDecodeError
6 |       +-- UnicodeEncodeError
7 +-- ArithmeticError
8 |   +-- ZeroDivisionError
9 +-- NameError
10 |   +-- UnboundLocalError
11 +-- LookupError
12 |   +-- IndexError
13 |   +-- KeyError
14 +-- OSError
15 |   +-- FileNotFoundError
16 |   +-- FileExistsError
17 |   +-- PermissionError
18 +-- SyntaxError
19 +-- AssertionError
20 +-- RuntimeError
21     +-- RecursionError
```

La hiérarchie complète des exceptions Python peut être trouvée à l'adresse suivante : <https://docs.python.org/fr/3/library/exceptions.html#exception-hierarchy> .

VII.7. Débogage

Introduction

Les bugs sont monnaie courante en programmation. Une erreur d'inattention est vite arrivée, et hop, un bug se glisse dans le programme.

Ils peuvent prendre de multiples formes : parfois ils feront planter purement et simplement l'application, d'autres mèneront à des traitements incohérents voire à des failles de sécurité, d'autres encore pourront être invisibles.

Mais quand un bug est repéré, il est encore loin d'être identifié : il faut trouver quelle fonction n'a pas eu un traitement correct et sur quelles données le problème survient.

Il s'agit alors de tenter de reproduire l'erreur dans différentes conditions pour l'identifier, pour enfin être en mesure de la corriger. C'est ce que l'on appelle le débogage !

Je ne peux que vous conseiller d'être attentif et de bien tester vos codes pour les éviter au maximum, malheureusement ce n'est pas toujours suffisant.

Aussi, pour ne pas vous retrouver désarmé quand un bug survient (qu'il soit décelé lorsque l'application tourne ou lors de tests), voici un petit guide pour apprendre à trouver l'origine du bug et la corriger.

VII.7.1. Programme de référence

Nous prendrons pour exemple au long de ce chapitre le programme suivant de combat entre monstres et qui présente plusieurs bugs :

```
1 import json
2
3
4 def input_choice(prompt, choices):
5     value = None
6     prompt += '(' + '/'.join(choices) + ') '
7     while value not in choices:
8         print('Valeur invalide')
9         value = input(prompt)
10    return value
11
12
13 def input_int(prompt):
14     while True:
15         try:
16             return int(input(prompt))
17         except ValueError:
18             print('Nombre invalide')
```

```

19
20
21 with open('data.json') as f:
22     data = json.load(f)
23     attacks = data['attacks']
24     monsters = data['monsters']
25
26
27 def input_player():
28     name = input_choice('Monstre: ', monsters)
29     monster = monsters[name]
30     pv = input_int('PV du monstre: ')
31     return {'monster': monster, 'pv': pv}
32
33
34 def input_attack(player):
35     monster = player['monster']
36     name = input_choice(f"Attaque de {monster['name']}: ",
37                         monster['attacks'])
38     return attacks[name]
39
40 def apply_attack(player1, player2, attack):
41     print(player1['monster']['name'], 'utilise', attack['name'],
42           ': ',
43           player2['monster']['name'], 'perd', attack['damage'],
44           'PV')
45     player1['pv'] -= attack['damage']
46
47
48 if __name__ == '__main__':
49     player1 = input_player()
50     player2 = input_player()
51     print(player1['monster']['name'], 'vs',
52           player2['monster']['name'])
53
54     while player1['pv'] and player2['pv'] > 0:
55         print(player1['monster']['name'], player1['pv'], 'PV')
56         print(player2['monster']['name'], player2['pv'], 'PV')
57
58         attack = input_attack(player1)
59         apply_attack(player1, player2, attack)
60
61         if player2['pv'] > 0:
62             attack = input_attack(player2)
63             apply_attack(player1, player2, attack)
64
65     if player1['pv'] > 0:
66         print(player1['monster']['name'], 'gagne')
67     else:

```

```
65 print(player2['monster']['name'], 'gagne')
```

Listing 67 – battle.py

Il s'accompagne du fichier de données ci-dessous.

```
1 {
2   "attacks": {
3     "charge": {"name": "Charge", "damage": 20},
4     "tonnerre": {"name": "Tonnerre", "damage": 50},
5     "jet-de-flotte": {"name": "Jet de flotte", "damages": 50},
6     "jet-de-flamme": {"name": "Jet de flamme", "damage": 60}
7   },
8   "monsters": {
9     "pythachu": {
10      "name": "Pythachu",
11      "attacks": ["charge", "tonnerre", "eclair"]
12    },
13    "pythard": {
14      "name": "Pythard",
15      "attacks": ["charge", "jet-de-flotte"]
16    },
17    "ponytha": {
18      "name": "Ponytha",
19      "attacks": ["charge", "jet-de-flamme"]
20    }
21  }
22 }
```

Listing 68 – data.json

Ces deux fichiers sont à retrouver sur le Gist suivant : <https://gist.github.com/entwanne/630e73d59696b0bab2899b0db1ea201b> ↗.

Vous pouvez doré et déjà tenter d'exécuter le programme, et constater que celui-ci a un comportement incohérent voire lève une exception.

```
1 % python battle.py
2 Valeur invalide
3 Monstre: (pythachu/pythard/ponytha) pythachu
4 PV du monstre: 10
5 Valeur invalide
6 Monstre: (pythachu/pythard/ponytha) pythard
7 PV du monstre: 10
8 Pythachu vs Pythard
9 Pythachu 10 PV
10 Pythard 10 PV
11 Valeur invalide
12 Attaque de Pythachu: (charge/tonnerre/eclair) tonnerre
13 Pythachu utilise Tonnerre : Pythard perd 50 PV
```

```
14 Valeur invalide
15 Attaque de Pythard: (charge/jet-de-flotte) charge
16 Pythachu utilise Charge : Pythard perd 20 PV
17 Pythachu -60 PV
18 Pythard 10 PV
19 Valeur invalide
20 Attaque de Pythachu: (charge/tonnerre/eclair) éclair
21 Traceback (most recent call last):
22   File "battle.py", line 55, in <module>
23     attack = input_attack(player1)
24   File "battle.py", line 37, in input_attack
25     return attacks[name]
26 KeyError: 'éclair'
```

VII.7.2. Introspection

VII.7.2.1. Informations sur la valeur

Avant d'en venir proprement au débogage de notre programme, faisons un tour des outils qui sont à notre disposition pour l'examiner. Il s'agit de fonctions proposées par Python pour inspecter différentes valeurs du programme.

On parle d'outils d'introspection car ils permettent au programme de s'examiner lui-même.

La première information, toute bête, c'est la valeur en elle-même, ou plutôt sa représentation. C'est ce que l'on obtient quand on tape juste le nom de la variable dans l'interpréteur interactif par exemple.

```
1 >>> value = 'toto'
2 >>> value
3 'toto'
```

Cette représentation est fournie par la fonction `repr`, qui renvoie donc une chaîne de caractères représentant la valeur. Souvent, cette représentation va être la manière dont il est possible de définir cette valeur en Python, c'est pour ça que des guillemets apparaissent autour des chaînes de caractères.

Elle peut tout à fait être appelée depuis un programme pour afficher (avec `print`) l'état d'une variable.

```
1 >>> print(repr(value))
2 'toto'
3 >>> print(repr(10))
4 10
5 >>> print(repr([1, 2, 'abc']))
6 [1, 2, 'abc']
```

Grâce à cette simple information, on identifie déjà à quoi correspond notre valeur.

Mais une autre information pertinente que l'on connaît aussi sur notre valeur, c'est son type,

VII. Aller plus loin

renvoyé par la fonction `type`.

Cela nous permet, pour peu que l'on connaisse le type, de s'avoir quelles opérations et méthodes sont applicables à notre objet.

```
1 >>> type(value)
2 <class 'str'>
3 >>> type([])
4 <class 'list'>
```

Et si on ne connaît pas ce type, on peut toujours se documenter dessus. Soit en consultant la documentation en ligne, soit à l'aide de la fonction `help` que j'ai présentée plus tôt.

On sait que cette fonction peut prendre un type en argument, il est donc tout à fait possible de lui donner directement le retour de la fonction `type`.

```
1 >>> help(type(value))
2 Help on class str in module builtins:
3
4 class str(object)
5 |   str(object='') -> str
6 |   str(bytes_or_buffer[, encoding[, errors]]) -> str
7 |   [...]
8 >>> help(type([]))
9 Help on class list in module builtins:
10
11 class list(object)
12 |   list(iterable=(), /)
13 |   [...]
```

Mais plus simple encore : on peut directement donner à `help` la valeur sur laquelle on a besoin d'aide, la fonction s'occupera de renvoyer la documentation du type correspondant.

```
1 >>> help([])
2 Help on list object:
3
4 class list(object)
5 |   list(iterable=(), /)
6 |   [...]
```



Attention, cela fonctionne pour toutes les valeurs sauf les chaînes de caractères.

En effet, la fonction `help` interprète les chaînes comme un sujet d'aide en particulier : `help('NUMBERS')` affichera de l'aide sur les nombres en Python et pas sur le type `str`.

```
1 >>> help('NUMBERS')
2 Numeric literals
```

```
3 *****
4
5 There are three types of numeric literals: integers, floating point
6 numbers, and imaginary numbers.
7 [...]
8 >>> help('toto')
9 No Python documentation found for 'toto'.
10 Use help() to get the interactive help utility.
11 Use help(str) for help on the str class.
```

Par ailleurs, il est possible de connaître tous les sujets sur lesquels `help` est capable de fournir de l'aide avec l'appel `help('topics')`.

VII.7.2.2. Contenu de la valeur

On a maintenant des informations globales sur notre valeur et l'on sait comment la manipuler, mais il peut-être utile de l'examiner encore plus loin pour savoir ce qu'elle contient. C'est l'objectif de la fonction `dir` qui va permettre de lister des méthodes et attributs d'un objet. Un appel à `dir` permet donc de savoir de façon plus concise que `help` ce que contient un objet, en ne nous renvoyant que les noms des méthodes/attributs.

```
1 >>> dir('toto')
2 ['__add__', '__class__', '__contains__', ..., 'strip', 'swapcase',
  'title', 'translate', 'upper', 'zfill']
```

Les méthodes de type `__xxx__` sont des méthodes spéciales et ne nous intéressent pas ici, elles sont abordées dans le cours sur [la programmation orientée objet en Python](#) .

Mais nous voyons ensuite les autres méthodes de l'objet telles que nous les connaissons déjà.

```
1 >>> 'toto'.title()
2 'Toto'
3 >>> 'toto'.upper()
4 'TOTO'
```

Pour les objets plus complexes (qui possèdent des attributs), la fonction `vars` permet de récupérer le dictionnaire de ces attributs. Par exemple on peut obtenir ainsi tout le contenu d'un module.

```
1 >>> vars(math)
2 {'__name__': 'math', ..., 'pi': 3.141592653589793, 'e':
  2.718281828459045, 'tau': 6.283185307179586, 'inf': inf,
  'nan': nan}
3 >>> vars(math)['pi']
4 3.141592653589793
```

À part les modules, on manipule assez peu d'objets avec des attributs dans les *built-ins* ou la bibliothèque standard, mais ils sont assez courants dans les bibliothèques tierces. On a tout de

VII. Aller plus loin

même la fonction `open` qui nous renvoie un tel objet par exemple.

```
1 >>> vars(open('hello.txt'))
2 {'mode': 'r'}
```

Un appel à `vars` sur un objet sans dictionnaire d'attributs lèvera une exception `TypeError`.

```
1 >>> vars('toto')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: vars() argument must have __dict__ attribute
```

i

Vous avez peut-être déjà rencontré la notation `obj.__dict__` pour accéder au dictionnaire d'attributs d'un objet, sachez qu'elle est équivalente à `vars(obj)`.

Notez enfin que `vars` peut s'utiliser sans argument, elle renverra alors le dictionnaire des variables définies dans l'espace de nom courant, ce qui peut aussi être utile au débogage.

```
1 >>> vars()
2 {'__name__': '__main__', '__doc__': None, '__package__': None,
   ..., 'value': 'toto'}
```

VII.7.3. Déboguer « à la main »

Maintenant que nous sommes en mesure de nous dépatouiller pour inspecter les valeurs, il est temps de comprendre comment elles évoluent au cours du programme pour mener jusqu'au bug.

VII.7.3.1. Suivre et comprendre les exceptions

Une manière courante de procéder au débogage, bien qu'elle ne soit pas des plus efficaces, est de placer des appels à `print` à plusieurs endroits du programme afin d'afficher des informations de debug.

Par exemple si l'on reprend le programme présenté en introduction, on remarque que le premier problème que l'on rencontre est que le programme affiche sans cesse « Valeur invalide » avant même que l'on ait entré quelque chose.

Pour comprendre ce qui se passe, on peut donc ajouter un `print` pour afficher ce que l'on connaît, la valeur de `value` et celle de `choices`.

```
4 def input_choice(prompt, choices):
5     value = None
6     prompt += '(' + '/'.join(choices) + ') '
7     while value not in choices:
```

```

8         print('debug', repr(value), repr(choices))
9         print('Valeur invalide')
10        value = input(prompt)
11    return value

```

Listing 69 – battle.py

```

1  % python battle.py
2  debug None {'pythachu': {'name': 'Pythachu', 'attacks': ['charge',
   'tonnerre', 'eclair']}, 'pythard': {'name': 'Pythard',
   'attacks': ['charge', 'jet-de-flotte']}, 'ponytha': {'name':
   'Ponytha', 'attacks': ['charge', 'jet-de-flamme']}}
3  Valeur invalide
4  Monstre: (pythachu/pythard/ponytha)

```

On peut déjà s'interroger sur le fait que `choices` soit un dictionnaire mais c'est normal : on lui passe directement l'objet `monsters` de notre JSON, et comme le `in` sur un dictionnaire fait une recherche sur les clés ça ne pose pas de problème.

Non le problème vient de ce `None`, la valeur initiale de notre variable, qui n'est effectivement pas dans les choix. On peut donc conditionner l'affichage du message d'erreur au fait que `value` ne soit pas `None` pour résoudre le premier problème.

```

7  while value not in choices:
8      if value is not None:
9          print('Valeur invalide')
10     value = input(prompt)

```

Listing 70 – battle.py

Une autre erreur que l'on remarque, c'est le plantage à la fin après avoir sélectionné l'attaque «eclair». C'est une erreur qui se produit systématiquement dans le cas où l'on choisit cette attaque, et qui n'arrive pas autrement.

On peut donc facilement la reproduire pour l'analyser.

Là encore, on peut afficher quelques informations au moment où l'on manipule cette information pour comprendre ce qu'il se passe.

La trace de l'erreur nous dit déjà que celle-ci se produit dans la fonction `input_attack`, c'est donc à cette fonction que nous allons nous intéresser en premier.

De la même manière que précédemment, on peut vérifier les différentes valeurs que l'on manipule pour s'assurer qu'elles correspondent à ce que l'on attend, notamment `player`, `monster`, `name` et `attacks`.



Pour nous aider à afficher nos valeurs, on peut s'appuyer sur le module `pprint`. Ce module fournit une fonction `pprint` (pour *pretty print*, soit *affichage joli*) qui donne un rendu plus aéré que `print`.

VII. Aller plus loin

```
35 def input_attack(player):
36     from pprint import pprint
37     print('player')
38     pprint(player)
39     monster = player['monster']
40     print('monster')
41     pprint(monster)
42     name = input_choice(f"Attaque de {monster['name']}: ",
43                         monster['attacks'])
44     print('name', repr(name))
45     print('attacks')
46     pprint(attacks)
47     return attacks[name]
```

Listing 71 – battle.py

On obtient alors le résultat suivant en relançant le programme.

```
1 player
2 {'monster': {'attacks': ['charge', 'tonnerre', 'eclair'], 'name':
3   'Pythachu'},
4   'pv': 10}
5 monster
6 {'attacks': ['charge', 'tonnerre', 'eclair'], 'name': 'Pythachu'}
7 Attaque de Pythachu: (charge/tonnerre/eclair) eclair
8 name 'eclair'
9 attacks
10 {'charge': {'damage': 20, 'name': 'Charge'},
11   'jet-de-flamme': {'damage': 60, 'name': 'Jet de flamme'},
12   'jet-de-flotte': {'damages': 50, 'name': 'Jet de flotte'},
13   'tonnerre': {'damage': 50, 'name': 'Tonnerre'}}
14 Traceback (most recent call last):
15   File "battle.py", line 61, in <module>
16     attack1 = input_attack(player1)
17   File "battle.py", line 50, in input_attack
18     return attacks[name]
19 KeyError: 'eclair'
```

Et là on constate que l'attaque `eclair` proposée pour le monstre n'existe pas dans le dictionnaire des attaques car elle n'est pas encore implémentée : on a donc simplement renseigné une mauvaise valeur dans notre JSON.

Il nous suffit de corriger ce dernier en supprimant `eclair` pour résoudre l'erreur, on peut alors retirer tous les `print` de debug de notre programme.

```
9 "pythachu": {
10     "name": "Pythachu",
11     "attacks": ["charge", "tonnerre"]
12 }
```

12	},
----	----

Listing 72 – data.json

VII.7.3.2. S'appuyer sur des tests unitaires

Mais on le voit, utiliser `print` pour déboguer peut être assez fastidieux. Heureusement un autre outil peut nous venir en aide : un ensemble de tests unitaires.

Je vous en parlais d'ailleurs [plus tôt](#) ¹, les tests unitaires nous permettent de déceler des bugs dans nos fonctions en vérifiant que le retour correspond à ce qui est attendu.

C'est pourquoi je ne peux que vous recommander de découper vos programmes en fonctions afin de plus facilement pouvoir les déboguer. L'idéal serait aussi de disposer les fonctions en différents modules pour pouvoir tester unitairement chacun des modules.

Mais revenons-en à notre code. Il possède peu de fonctions que nous pouvons tester en l'état car beaucoup reposent sur des entrées utilisateurs que nous ne savons pas simuler¹.

Il n'y a en fait que la fonction `apply_attack` qui est déterministe : elle doit toujours faire la même chose quand on lui renseigne les mêmes arguments.

Pour la tester, il faut alors que l'on donne à la fonction des données dans le format qu'elle attend (deux joueurs et une attaque) puis que l'on vérifie son retour. Ici la fonction ne renvoie rien mais elle peut altérer ses paramètres, c'est donc sur ceux-ci que nous ferons nos assertions afin de vérifier que les points de vie sont bien mis à jour (en l'occurrence que les dégâts sont retirés du second monstre).

Les arguments n'ont pas besoin d'être exhaustifs mais simplement de contenir les informations qui seront utilisées par la fonction. Ici les joueurs n'ont besoin de n'avoir par exemple qu'un nombre de points de vie et un nom de monstre, et l'attaque seulement un nom et un nombre de dégâts.

```

1  from battle import apply_attack
2
3
4  def test_apply_attack():
5      p1 = {
6          'monster': {'name': 'Pythachu'},
7          'pv': 50,
8      }
9      p2 = {
10         'monster': {'name': 'Ponytha'},
11         'pv': 100,
12     }
13     attack = {'name': 'électrocution', 'damage': 30}
14
15     apply_attack(p1, p2, attack)
16     assert p2['pv'] == 70
17
18

```

1. Nous apprendrons à le faire par la suite à l'aide de *mocks* intégré aux *frameworks* de tests, mais ce n'est pas l'objet de ce chapitre.

VII. Aller plus loin

```
19 if __name__ == '__main__':  
20     test_apply_attack()
```

Listing 73 – test_battle.py

Et là... c'est le drame !

```
1 % python test_battle.py  
2 Pythachu utilise électrocution : Ponytha perd 30 PV  
3 Traceback (most recent call last):  
4   File "test_battle.py", line 20, in <module>  
5     test_apply_attack()  
6   File "test_battle.py", line 16, in test_apply_attack  
7     assert p2['pv'] == 70  
8 AssertionError
```

Notre assertion échoue parce que les PV du second joueur ne valent pas 70 comme attendu. Sans plus d'outils à notre disposition pour le moment, on peut associer à nos tests un `print` comme précédemment afin d'obtenir plus d'informations.

```
15 apply_attack(p1, p2, attack)  
16     print('résultat', p2['pv'])  
17     assert p2['pv'] == 70
```

Listing 74 – test_battle.py

À l'exécution du test on comprend mieux le problème : les PV du deuxième joueur n'ont pas bougé.

```
1 % python test_battle.py  
2 résultat 100  
3 Traceback (most recent call last):  
4   File "test_battle.py", line 21, in <module>  
5     test_apply_attack()  
6   File "test_battle.py", line 17, in test_apply_attack  
7     assert p2['pv'] == 70  
8 AssertionError
```

On peut alors se demander où sont retirés les PV, et logiquement ajouter une assertion sur le premier joueur.

```
15 apply_attack(p1, p2, attack)  
16     print('résultat', p1['pv'], p2['pv'])  
17     assert p1['pv'] == 50  
18     assert p2['pv'] == 70
```

Listing 75 – test_battle.py

```
1 % python test_battle.py
2 Pythachu utilise électrocution : Ponytha perd 30 PV
3 résultat 20 100
4 Traceback (most recent call last):
5   File "test_battle.py", line 22, in <module>
6     test_apply_attack()
7   File "test_battle.py", line 17, in test_apply_attack
8     assert p1['pv'] == 50
9 AssertionError
```

Cette fois-ci c'est clair : les dégâts sont appliqués au premier joueur plutôt qu'au deuxième. Et à regarder notre fonction `apply_attack`, c'est vrai que les noms des paramètres `player1` et `player2` prêtent à confusion.

Nous leur préférons alors respectivement les noms plus descriptifs de `attacker` (attaquant) et `target` (cible).

```
41 def apply_attack(attacker, target, attack):
42     print(attacker['monster']['name'], 'utilise', attack['name'],
43           ': ',
44           target['monster']['name'], 'perd', attack['damage'],
45           'PV')
46     target['pv'] -= attack['damage']
```

Listing 76 – battle.py

On peut alors exécuter à nouveau nos tests et constater que tout se passe bien.

```
1 % python test_battle.py
2 Pythachu utilise électrocution : Ponytha perd 30 PV
3 résultat 50 70
```

VII.7.4. Utilisation d'un débogueur (Pdb)

Vous avez peut-être remarqué un autre bug dans notre programme : le jeu ne s'arrête pas quand le premier joueur est censé être KO.

```
1 % python battle.py
2 Monstre: (pythachu/pythard/ponytha) pythachu
3 PV du monstre: 100
4 Monstre: (pythachu/pythard/ponytha) ponytha
5 PV du monstre: 120
6 Pythachu vs Ponytha
7 Pythachu 100 PV
8 Ponytha 120 PV
```

VII. Aller plus loin

```
9  Attaque de Pythachu: (charge/tonnerre) charge
10 Pythachu utilise Charge : Ponytha perd 20 PV
11 Attaque de Ponytha: (charge/jet-de-flamme) jet-de-flamme
12 Ponytha utilise Jet de flamme : Pythachu perd 60 PV
13 Pythachu 40 PV
14 Ponytha 100 PV
15 Attaque de Pythachu: (charge/tonnerre) charge
16 Pythachu utilise Charge : Ponytha perd 20 PV
17 Attaque de Ponytha: (charge/jet-de-flamme) jet-de-flamme
18 Ponytha utilise Jet de flamme : Pythachu perd 60 PV
19 Pythachu -20 PV
20 Ponytha 80 PV
21 Attaque de Pythachu: (charge/tonnerre) charge
22 Pythachu utilise Charge : Ponytha perd 20 PV
23 Attaque de Ponytha: (charge/jet-de-flamme) jet-de-flamme
24 Ponytha utilise Jet de flamme : Pythachu perd 60 PV
25 Pythachu -80 PV
26 Ponytha 60 PV
27 [...]
```

Pour déceler son origine, nous allons cette fois-ci faire appel à un débogueur, j'ai nommé Pdb (pour *Python Debugger*).

VII.7.4.1. Lancer un programme pas-à-pas avec Pdb

Pour commencer, on va lancer l'exécution de notre programme pas-à-pas à l'aide de Pdb, en l'exécutant via `python -m pdb battle.py`.

```
1 % python -m pdb battle.py
2 > ../../battle.py(1)<module>()
3 -> import json
4 (Pdb)
```

Pdb nous indique quel fichier est exécuté (`battle.py`) et quelle ligne (`import json`). Puis on se retrouve face à un prompt qui attend nos ordres pour continuer l'exécution. Ce prompt comprend plusieurs commandes que nous allons voir ici.

Premièrement nous pouvons lui demander un peu de contexte. Cela se fait avec la commande `list` (ou simplement `l`) qui va afficher les lignes autour de nous.

```
1 (Pdb) l
2 1 ->         import json
3 2
4 3
5 4         def input_choice(prompt, choices):
6 5             value = None
7 6             prompt += '(' + '/'.join(choices) + ') '
8 7             while value not in choices:
```

VII. Aller plus loin

```
9      8          if value is not None:
10     9              print('Valeur invalide')
11    10              value = input(prompt)
12    11          return value
```

On peut continuer l'exécution jusqu'à l'instruction suivante à l'aide de la commande `next` (ou `n`).

```
1 (Pdb) n
2 > /.../battle.py(4)<module>()
3 -> def input_choice(prompt, choices):
4 (Pdb) n
5 > /.../battle.py(14)<module>()
6 -> def input_int(prompt):
7 (Pdb) n
8 > /.../battle.py(22)<module>()
9 -> with open('data.json') as f:
```

Mais on se rend vite compte que c'est un peu long à avancer. On pourrait plutôt se rendre directement là où ça nous intéresse : au début de la boucle de jeu.

Pour cela il existe la commande `until` (ou `unt`) qui prend en argument un numéro de ligne, le programme continuera alors son exécution jusqu'à cette ligne.

Mais comment connaître le numéro de ligne que l'on souhaite atteindre ? Si vous avez le fichier ouvert en parallèle, vous pouvez voir que le `while` se trouve ligne 52. Sinon, un appel à la commande `longlist` (ou `ll`) permet d'afficher toutes les lignes du fichier.

```
1 (Pdb) ll
2 [...]
3 47      if __name__ == '__main__':
4 48          player1 = input_player()
5 49          player2 = input_player()
6 50          print(player1['monster']['name'], 'vs',
7              player2['monster']['name'])
8 51
9 52          while player1['pv'] and player2['pv'] > 0:
10 53              print(player1['monster']['name'],
11                  player1['pv'], 'PV')
12 54              print(player2['monster']['name'],
13                  player2['pv'], 'PV')
14 [...]
15
```

On va maintenant pouvoir entrer la commande `until 52` pour avancer jusqu'à notre boucle. Là notre programme reprend son exécution normale sur les lignes intermédiaires, et nous demande alors d'entrer les informations sur les joueurs.


```

1 (Pdb) until 52
2 Monstre: (pythachu/pythard/ponytha) pythachu
3 PV du monstre: 100
4 Monstre: (pythachu/pythard/ponytha) ponytha
5 PV du monstre: 120
6 Pythachu vs Ponytha
7 > /.../battle.py(52)<module>()
8 -> while player1['pv'] and player2['pv'] > 0:
9 (Pdb)

```

Nous entrons maintenant dans la boucle et nous pouvons reprendre l'exécution pas-à-pas à l'aide de `next`.



Petite astuce : il est possible d'utiliser les flèches haut/bas de votre clavier pour naviguer dans l'historique des commandes entrées à Pdb.

```

1 > /.../battle.py(52)<module>()
2 -> while player1['pv'] and player2['pv'] > 0:
3 (Pdb) next
4 > /.../battle.py(53)<module>()
5 -> print(player1['monster']['name'], player1['pv'], 'PV')
6 (Pdb) next
7 Pythachu 100 PV
8 > /.../battle.py(54)<module>()
9 -> print(player2['monster']['name'], player2['pv'], 'PV')
10 (Pdb) next
11 Ponytha 120 PV
12 > /.../battle.py(56)<module>()
13 -> attack = input_attack(player1)
14 (Pdb) next
15 Attaque de Pythachu: (charge/tonnerre) charge
16 > /.../battle.py(57)<module>()
17 -> apply_attack(player1, player2, attack)
18 (Pdb) next
19 Pythachu utilise Charge : Ponytha perd 20 PV
20 > /.../battle.py(59)<module>()
21 -> if player2['pv'] > 0:
22 (Pdb) next
23 > /.../battle.py(60)<module>()
24 -> attack = input_attack(player2)
25 (Pdb) next
26 Attaque de Ponytha: (charge/jet-de-flamme) jet-de-flamme
27 > /.../battle.py(61)<module>()
28 -> apply_attack(player2, player1, attack)
29 (Pdb) next
30 Ponytha utilise Jet de flamme : Pythachu perd 60 PV

```

VII. Aller plus loin

```
31 > ../../battle.py(52)<module>()
32 -> while player1['pv'] and player2['pv'] > 0:
33 (Pdb)
```

On a fait un tour de boucle et on ne voit rien d'anormal pour le moment. On peut afficher à l'aide de la commande `pp` (*pretty-print*) les valeurs de certaines variables pour vérifier que tout va bien.

```
1 (Pdb) pp player1
2 {'monster': {'attacks': ['charge', 'tonnerre'], 'name':
   'Pythachu'}, 'pv': 40}
3 (Pdb) pp player2
4 {'monster': {'attacks': ['charge', 'jet-de-flamme'], 'name':
   'Ponytha'},
5  'pv': 100}
```

On pourrait alors recommencer les mêmes opérations pour effectuer un tour de boucle supplémentaire, mais comme nous l'avons vu c'est un peu long.

Nous allons donc utiliser une autre fonctionnalité offerte par Pdb : les points d'arrêts (ou *breakpoints*). Il s'agit de points dans le programme qui provoqueront sa mise en pause chaque fois qu'ils seront atteints.

C'est notre boucle qui nous intéresse, et nous posons donc un point d'arrêt sur la ligne 52 à l'aide de la commande `break 52`.

```
1 (Pdb) break 52
2 Breakpoint 1 at ../../battle.py:52
```

Pdb nous informe bien qu'un *breakpoint* a été posé. La commande `break` utilisée sans argument nous permet de lister tous les *breakpoints*.

```
1 (Pdb) break
2 Num Type          Disp Enb   Where
3 1 breakpoint      keep yes   at ../../battle.py:52
```

Nous pouvons maintenant demander à Pdb de continuer normalement l'exécution du programme à l'aide de la commande `continue` (abrégée en `cont` ou `c`). Le programme reprendra alors son cours normal jusqu'à la prochaine interruption (notre point d'arrêt).

```
1 (Pdb) c
2 Pythachu 40 PV
3 Ponytha 100 PV
4 Attaque de Pythachu: (charge/tonnerre) charge
5 Pythachu utilise Charge : Ponytha perd 20 PV
6 Attaque de Ponytha: (charge/jet-de-flamme) jet-de-flamme
7 Ponytha utilise Jet de flamme : Pythachu perd 60 PV
8 > ../../battle.py(52)<module>()
```

VII. Aller plus loin

```
9 -> while player1['pv'] and player2['pv'] > 0:
10 (Pdb)
```

Nous sommes bien revenus à la condition de notre boucle, et cette fois Pythachu est censé être KO, donc nous devrions en sortir. On va s'en assurer à l'aide d'un simple `next` : nous verrons tout de suite où nous emmène la prochaine instruction.

```
1 (Pdb) next
2 > ../../battle.py(53)<module>()
3 -> print(player1['monster']['name'], player1['pv'], 'PV')
```

Et là c'est le drame. La boucle ne s'est pas arrêtée comme ça aurait dû être le cas. On peut jeter un œil à la valeur de `player1` pour essayer de comprendre.

```
1 (Pdb) pp player1
2 {'monster': {'attacks': ['charge', 'tonnerre'], 'name':
  'Pythachu'}, 'pv': -20}
```

Les PV sont négatifs, la condition de boucle aurait alors dû être fausse. `pp` n'accepte pas seulement une variable en argument mais n'importe quelle expression. On peut alors exécuter `pp` sur la condition de notre boucle pour voir ce qui cloche.

```
1 (Pdb) pp player1['pv'] and player2['pv'] > 0
2 True
```

Bien que les points de vie du joueur 1 soient négatifs, cette condition est tout de même considérée comme vraie. La source de notre bug se trouve donc ici.

Et effectivement, si nous analysons notre condition de plus près, nous pouvons voir qu'elle est équivalente à `(player1['pv']) and (player2['pv'] > 0)`.

On ne teste donc jamais si les points de vie du premier joueur sont positifs, mais seulement s'ils ne sont pas nuls.

Il ne nous reste plus qu'à corriger notre programme dans l'éditeur de texte pour utiliser la condition `player1['pv'] > 0 and player2['pv'] > 0` et à recommencer le débogage une fois notre fichier enregistré à l'aide de la commande `restart`.

Le programme repart alors de zéro depuis la première ligne, on peut entrer la commande `continue` pour continuer l'exécution jusqu'au point d'arrêt.

On refait deux tours d'attaque comme précédemment pour revenir sur la condition de notre boucle qui doit maintenant être fausse.

```
1 Ponytha utilise Jet de flamme : Pythachu perd 60 PV
2 > ../../battle.py(52)<module>()
3 -> while player1['pv'] > 0 and player2['pv'] > 0:
4 (Pdb) pp player1
5 {'monster': {'attacks': ['charge', 'tonnerre'], 'name':
  'Pythachu'}, 'pv': -20}
```

```
6 (Pdb) pp player1['pv'] > 0 and player2['pv'] > 0
7 False
```

On peut alors exécuter `next` et vérifier que l'on sort bien de la boucle.

```
1 (Pdb) next
2 > ../../battle.py(63)<module>()
3 -> if player1['pv'] > 0:
```

Le bug en question est donc corrigé !

VII.7.4.2. Invoquer Pdb depuis le programme

Mais ce mode d'utilisation de Pdb n'est pas le plus intuitif. Généralement on a déjà constaté le bug en dehors du débogueur et on sait donc à peu près à quel endroit il va se produire. On pourrait alors directement poser notre point d'arrêt dans le programme pour invoquer Pdb. Cela est rendu possible à l'aide de la fonction `breakpoint` de Python, callable depuis n'importe où dans le programme. On lancera alors notre jeu normalement, et la fonction aura pour effet de le mettre en pause et de nous amener sur une console Pdb.

i

Avant Python 3.7, la fonction `breakpoint` n'existait pas. Il fallait alors écrire `import pdb; pdb.set_trace()` dans le code pour placer un point d'arrêt.

Vous avez peut-être pu constater un autre bug dans le jeu en utilisant le monstre Pythard, celui-ci se produit quand on essaie d'utiliser l'attaque *jet-de-flotte*.

```
1 % python battle.py
2 Monstre: (pythachu/pythard/ponyth) pythard
3 PV du monstre: 100
4 Monstre: (pythachu/pythard/ponyth) pythachu
5 PV du monstre: 100
6 Pythard vs Pythachu
7 Pythard 100 PV
8 Pythachu 100 PV
9 Attaque de Pythard: (charge/jet-de-flotte) jet-de-flotte
10 Traceback (most recent call last):
11   File "../../battle.py", line 57, in <module>
12     apply_attack(player1, player2, attack)
13   File "../../battle.py", line 43, in apply_attack
14     target['monster']['name'], 'perd', attack['damage'], 'PV')
15   KeyError: 'damage'
```

On constate donc que le bug survient dans la fonction `apply_attack` et l'on va pouvoir placer un point d'arrêt directement dans cette fonction.

VII. Aller plus loin

```
41 def apply_attack(attacker, target, attack):
42     breakpoint()
43     print(attacker['monster']['name'], 'utilise', attack['name'],
44           ':',
45           target['monster']['name'], 'perd', attack['damage'],
46           'PV')
47     target['pv'] -= attack['damage']
```

Il nous suffit ensuite de relancer normalement notre programme. Et après avoir saisi les informations de jeu, on se retrouve interrompu par notre *breakpoint*.

```
1 % python battle.py
2 Monstre: (pythachu/pythard/ponyth) pythard
3 PV du monstre: 100
4 Monstre: (pythachu/pythard/ponyth) pythachu
5 PV du monstre: 100
6 Pythard vs Pythachu
7 Pythard 100 PV
8 Pythachu 100 PV
9 Attaque de Pythard: (charge/jet-de-flotte) jet-de-flotte
10 > /.../battle.py(43)apply_attack()
11 -> print(attacker['monster']['name'], 'utilise', attack['name'],
12         ':',
13         (Pdb))
```

Nous pouvons alors reprendre notre attirail de commandes et essayer de comprendre le problème en inspectant les différentes valeurs.

```
1 (Pdb) pp attacker
2 {'monster': {'attacks': ['charge', 'jet-de-flotte'], 'name':
3   'Pythard'},
4   'pv': 100}
5 (Pdb) pp attack
6 {'damages': 50, 'name': 'Jet de flotte'}
7 (Pdb) pp target
8 {'monster': {'attacks': ['charge', 'tonnerre'], 'name':
9   'Pythachu'}, 'pv': 100}
```

Rien qui ne saute forcément aux yeux pour l'instant, donc on continue l'exécution avec `next`.

```
1 (Pdb) next
2 > /.../battle.py(44)apply_attack()
3 -> target['monster']['name'], 'perd', attack['damage'], 'PV')
4 (Pdb) next
5 KeyError: 'damage'
6 > /.../battle.py(44)apply_attack()
```

```
7 -> target['monster']['name'], 'perd', attack['damage'], 'PV')
```

Là on voit bien l'erreur `KeyError` qui se produit et la ligne fautive est pointée. On se rend alors compte qu'on a utilisé dans notre JSON la clé `'damages'` plutôt que `'damage'` pour l'attaque *jet-de-flotte*. Encore une fois l'erreur venait donc de nos données.

On peut directement quitter le programme pour aller corriger notre fichier `data.json`.

Par acquit de conscience, on le relance ensuite dans les mêmes conditions pour vérifier que tout se passe bien.

```
1 Attaque de Pythard: (charge/jet-de-flotte) jet-de-flotte
2 > /.../battle.py(43)apply_attack()
3 -> print(attacker['monster']['name'], 'utilise', attack['name'],
      ': ',
4 (Pdb)
```

On est à nouveau interrompu par notre *breakpoint* et on avance alors pas-à-pas pour nous assurer du bon fonctionnement.

```
1 (Pdb) next
2 > /.../battle.py(44)apply_attack()
3 -> target['monster']['name'], 'perd', attack['damage'], 'PV'
4 (Pdb) next
5 > /.../battle.py(43)apply_attack()
6 -> print(attacker['monster']['name'], 'utilise', attack['name'],
      ': ',
7 (Pdb) continue
8 Pythard utilise Jet de flotte : Pythachu perd 50 PV
9 Attaque de Pythachu: (charge/tonnerre)
```

Cette fois-ci c'est bon, le problème semble bien résolu. Mais le point d'arrêt reste toujours présent et continuera de nous interrompre. Il nous suffira de retirer la ligne `breakpoint()` dans le programme pour le supprimer.



La fonction `breakpoint` peut aussi directement s'utiliser depuis des fonctions de test, et ainsi être invoquée lors de l'exécution des tests.

Pour plus d'informations sur les commandes comprises par Pdb, je vous invite à consulter [sa page de documentation](#) .

VII.7.5. Déboguer avec votre IDE

Jongler entre l'éditeur de texte d'un côté et le débogueur de l'autre n'est pas toujours évident. Certains éditeurs permettent d'intégrer directement le débogueur dans leur interface. C'est le cas de PyCharm par exemple.

Il est ainsi possible de cliquer à gauche des lignes de code pour placer des points d'arrêt. Ils sont illustrés par un rond rouge dans la marge. Cliquer sur un tel rond permet de supprimer un point d'arrêt déjà posé.

VII. Aller plus loin

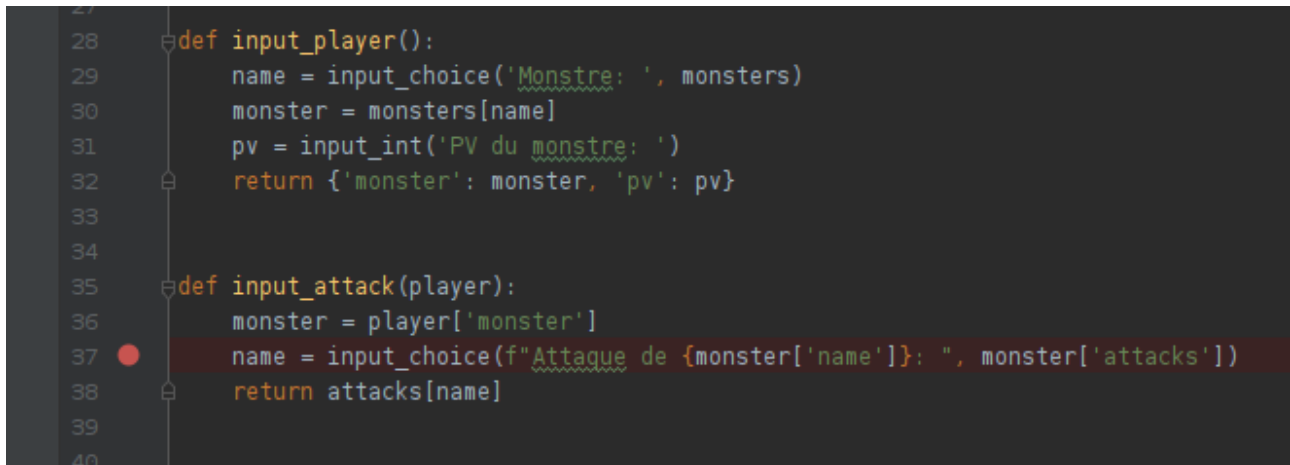


FIGURE VII.7.1. – Point d’arrêt dans PyCharm.

Il faut ensuite exécuter le programme en mode *debug* (*Run > Debug*) pour les prendre en compte. L’exécution se déroule alors normalement nous demandant d’entrer les différentes saisies, puis le programme se met en pause et bascule sur l’interface de débogage quand le point d’arrêt est rencontré.

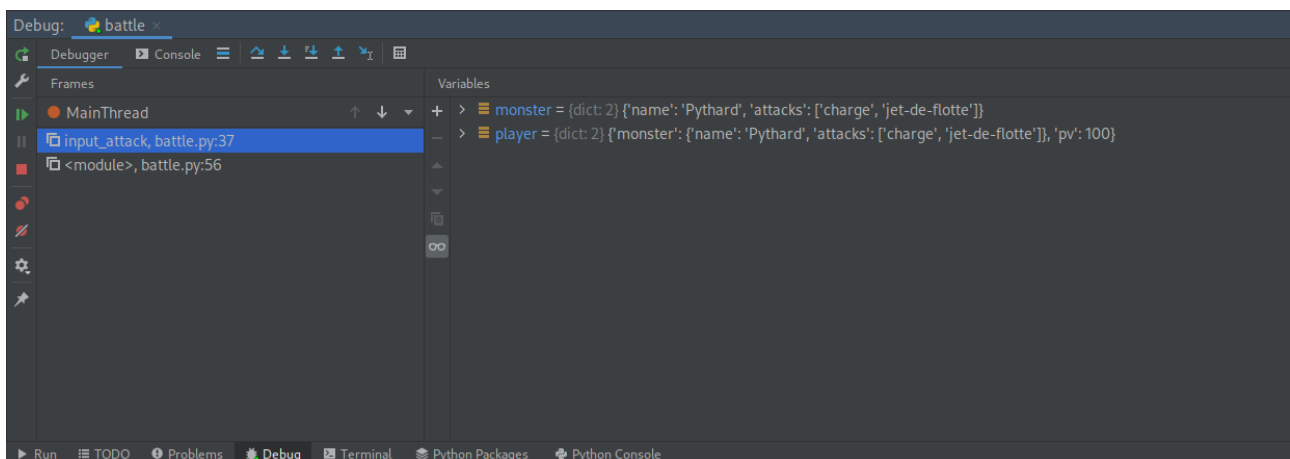


FIGURE VII.7.2. – Interface de débogage.

Là nous pouvons directement avoir un aperçu des variables présentes dans la fonction, et les boutons reprennent les actions que nous avons pu voir avec Pdb : passer à l’instruction suivante, reprendre l’exécution du programme, redémarrer depuis le début, etc.

Pour plus d’informations sur le débogage avec PyCharm je vous invite à consulter [cette page d’aide](#) [↗](#) (en anglais).

Huitième partie

La bibliothèque standard

Introduction

La bibliothèque standard regorge de nombreux modules répondant à diverses tâches et il est intéressant d'au moins connaître leur existence.

Cette partie a donc pour but de faire un tour d'ensemble de cette bibliothèque.

VIII.1. Tour d’horizon de la bibliothèque standard

Introduction

J’ai déjà évoqué à plusieurs reprises la bibliothèque standard de Python (ou *stdlib* pour *standard library*), il s’agit de l’ensemble des modules et fonctions qui sont inclus de base avec Python. Chaque langage vient avec sa bibliothèque standard mais celle de Python est particulièrement fournie.

Nous en avons déjà parcouru une partie au cours des chapitres précédents et je ne pourrai pas être exhaustif ici non plus. Sachez qu’elle comprend par exemple des modules pour gérer les nombres fractionnaires, les dates, les chemins de fichier, mais aussi les archives ZIP, les emails, le protocole HTTP, etc.

Un réflexe à avoir lorsque vous recherchez une fonctionnalité particulière en Python et de d’abord regarder si celle-ci est présente dans un module de la bibliothèque standard, et pour cela [la documentation est votre amie](#) [↗](#).

VIII.1.1. Fonctions natives

Je ne reviendrai pas sur l’ensemble des fonctions natives (*built-ins*) car beaucoup ont déjà été présentées dans les chapitres précédents, notamment [celui rappelant les différents types](#) [↗](#) et [celui dédié aux outils sur les boucles](#) [↗](#).

Mais quelques autres de ces fonctions méritent qu’on en parle un peu.

VIII.1.1.0.1. Manipulation de caractères

Les fonctions `ord` et `chr` par exemple permettent de manipuler les caractères et leurs codes numériques.

Jusqu’ici on n’a jamais dissocié caractères et chaînes de caractères, puisque les caractères sont simplement des chaînes de taille 1.

Mais en pratique, une chaîne de caractères s’apparente plutôt à une séquence de code numériques (des nombres entiers) où chaque code identifie un caractère particulier selon la spécification unicode.

Ainsi, la fonction `ord` permet simplement de récupérer le code associé à un caractère, et la fonction `chr` le caractère associé à un code.

```
1 >>> ord('x')
2 120
3 >>> chr(120)
4 'x'
```

```
5 >>> ord('♣')
6 9835
7 >>> chr(9835)
8 '♣'
```

Ces fonctions peuvent permettre de jongler un peu avec la table unicode pour réaliser des opérations particulières en exploitant les caractéristiques de cette table.

Par exemple pour récupérer n'importe quelle carte à jouer en connaissant [la manière dont elles sont stockées](#) ↗ :

```
1 >>> card_base = ord('')
2 >>> chr(card_base + 0x20 + 0x05) # 5 de carreau
3 ''
4 >>> chr(card_base + 0x10 + 0x0B) # Valet de pic
5 ''
```

`ord` échoue naturellement si on lui passe une chaîne de plusieurs caractères, et `chr` si on lui donne un code en dehors des bornes définies par unicode.

```
1 >>> ord('salut')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: ord() expected a character, but string of length 5 found
5 >>> chr(1000000000)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   ValueError: chr() arg not in range(0x110000)
```

VIII.1.1.0.2. Formattage des valeurs

La fonction `format` permet d'obtenir la représentation formatée de n'importe quelle valeur, sous forme d'une chaîne de caractères.

Vous ne la connaissez pas mais c'est elle qui intervient dans le mécanisme des chaînes de formatage (*f-string*) pour transformer les valeurs et leur appliquer le format demandé.

Elle prend ainsi en arguments la valeur et le format (les options de formatage) à lui appliquer.

```
1 >>> format(42, '05X')
2 '0002A'
3 >>> format(123.4, 'e')
4 '1.234000e+02'
5 >>> format('salut', '>10')
6 '      salut'
```

Appelée sans format, elle opère juste la conversion en chaîne de caractères de la valeur donnée et devient ainsi équivalente à `str`.

```
1 >>> format(25)
2 '25'
```

VIII.1.1.0.3. Évaluation dynamique



La fonction qui suit peut introduire de grosses failles de sécurité dans vos programmes et doit donc être utilisée avec parcimonie : seulement sur des données qui sont sûres, jamais sur des données reçues de l'utilisateur ou d'un tiers.

Python est un langage dynamique et permet en cela d'exécuter du code à la volée au sein du programme.

C'est l'objectif de la fonction `eval` qui prend en argument une chaîne de caractères représentant une expression Python, l'interprète et en renvoie le résultat.

```
1 >>> eval('1 + 3')
2 4
3 >>> x = 5
4 >>> eval('x * 8')
5 40
```

Cela offre donc la possibilité d'exécuter du code dynamiquement et donc de dépasser les fonctionnalités de base du langage. Par exemple pour créer en un coup une imbrication de 20 listes.

```
1 >>> eval('['*20 + 'None' + ']'*20)
2 [[[[[[[[[[[[[[[[[[[None]]]]]]]]]]]]]]]]]]]
```

Toutes ces fonctions natives peuvent être retrouvées sur [la page de documentation dédiée ↗](#).

VIII.1.1.1. Module operator

Les opérateurs font en quelque sorte partie des *built-ins* même si on y pense moins. Après tout, il s'agit aussi de fonctions natives de Python.

Mais les opérateurs sont des symboles et on ne peut pas les manipuler en tant que tels. En revanche, le module `operator` fournit pour chaque opérateur de Python un équivalent sous forme de fonction.

On y trouve ainsi des fonctions `add`, `sub`, `pow` ou encore `eq`.

```
1 >>> import operator
2 >>> operator.add(3, 5)
3 8
4 >>> operator.sub(10, 1)
```

```

5 9
6 >>> operator.pow(2, 3)
7 8
8 >>> operator.eq('a', 'a')
9 True
10 >>> operator.eq('a', 'b')
11 False

```

Quelques subtilités à noter :

- Il y a deux fonctions de division (`truediv` et `floordiv`) pour les deux opérateurs correspondant (respectivement `/` et `//`).

```

1 >>> operator.truediv(10, 4)
2 2.5
3 >>> operator.floordiv(10, 4)
4 2

```

- `operator.concat` (concaténation) est équivalent à `operator.add`, ces deux opérations se représentant par l'opérateur `+`, mais s'attend à ce que ses arguments soient des séquences.

```

1 >>> operator.concat('foo', 'bar')
2 'foobar'
3 >>> operator.add('foo', 'bar')
4 'foobar'
5 >>> operator.concat(3, 5)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: 'int' object can't be concatenated

```

- Les opérateurs `&` et `|` deviennent `and_` et `or_`, suffixés d'un `_` pour ne pas générer de conflit avec les mots-clés `and` et `or`. De même que `not` devient `not_`.

```

1 >>> operator.and_(3, 1)
2 1
3 >>> operator.or_(3, 1)
4 3
5 >>> operator.not_(False)
6 True

```

- Pour chaque fonction `xxx` d'un opérateur arithmétique on trouve une fonction `ixxx` pour l'opérateur en-place (par-exemple `iadd` pour `+=`).

```

1 >>> values = []
2 >>> operator.iadd(values, [42])
3 [42]

```

```

4 >>> values
5 [42]

```

- Les opérateurs `foo[key]`, `foo[key] = value` et `del foo[key]` sont appelés `getitem`, `setitem` et `delitem`. `getitem` renvoie la valeur demandée, `setitem` et `delitem` renvoient `None`.

```

1 >>> operator.setitem(values, 0, 21)
2 >>> operator.getitem(values, 0)
3 21
4 >>> operator.delitem(values, 0)
5 >>> values
6 []

```

- On trouve une fonction spéciale `itemgetter` qui permet de générer un opérateur renvoyant la valeur associée à une clé dans un conteneur.

```

1 >>> get_3rd = operator.itemgetter(3)
2 >>> get_3rd('abcdef')
3 'd'
4 >>> get_3rd([3, 4, 5, 6])
5 6
6 >>> get_3rd(range(10))
7 3
8 >>> get_foo = operator.itemgetter('foo')
9 >>> get_foo({'foo': -12})
10 -12

```

VIII.1.2. Gestion des nombres

On a vite fait de passer sur les nombres, car on peut croire que l'on en a fait le tour une fois que l'on a vu les types `int`, `float` et `complex` qui semblent en effet couvrir toutes les catégories de nombres que l'on connaît.

Si c'est vrai pour ce qui est des nombres entiers, les types dédiés aux réels et aux complexes n'en sont que des approximations.

VIII.1.2.1. Nombres décimaux

En effet, nous avons vu que les `float` étaient stockés par l'ordinateur sous forme binaire et étaient donc souvent des approximations des nombres décimaux que nous connaissons, ce qui pouvait mener à des erreurs d'arrondis.

```

1 >>> 0.1 + 0.1 + 0.1
2 0.30000000000000004

```

VIII. La bibliothèque standard

Un autre type existe néanmoins pour représenter de façon précise un nombre décimal, il s'agit du type `Decimal` du module `decimal`.

```
1 >>> from decimal import Decimal
2 >>> Decimal('0.1')
3 Decimal('0.1')
4 >>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
5 Decimal('0.3')
```

Un `Decimal` s'instancie avec une chaîne de caractère représentant notre nombre décimal et se comporte ensuite comme n'importe quel nombre : toutes les opérations usuelles peuvent s'y appliquer.

```
1 >>> Decimal('1.5') * Decimal('3.7')
2 Decimal('5.55')
3 >>> Decimal('-4.2') - Decimal('1.1')
4 Decimal('-5.3')
```

Les décimaux sont aussi compatibles avec les entiers, une opération entre des nombres des deux types renverra toujours un décimal.

```
1 >>> Decimal('0.1') + 3
2 Decimal('3.1')
3 >>> Decimal('0.1') * 4
4 Decimal('0.4')
5 >>> Decimal('0.1') ** 2
6 Decimal('0.01')
```

?

Vous vous demandez pourquoi on instancie un `Decimal` avec une chaîne de caractères plutôt qu'un `float` ?

Il est en fait possible de créer un décimal à partir d'un flottant, mais ce flottant comprenant dès le départ une erreur d'arrondi, celle-ci se répercutera sur le décimal.

```
1 >>> Decimal(0.1)
2 Decimal('0.100000000000000005551115123125782702118158340454101_
   5625')
```

On notera par ailleurs qu'il est possible de créer un décimal à partir d'un nombre entier, ce dernier ne comportant pas d'approximation.

```
1 >>> Decimal(1) / Decimal(10)
2 Decimal('0.1')
```

À tout moment, il est possible de convertir un décimal en entier ou flottant à l'aide d'un appel à

`int` ou `float`.

```
1 >>> int(Decimal('1.4') * Decimal('1.5'))
2 2
3 >>> float(Decimal('1.4') * Decimal('1.5'))
4 2.1
```

Les nombres décimaux sont pratiques pour manipuler des valeurs qui ne doivent pas subir d'arrondis involontaires, comme des valeurs monétaires, mais sont moins performants à manipuler que les flottants qui sont directement gérés par le processeur.

Enfin, les décimaux sont tout de même soumis à une précision limitée qui ne leur permet alors pas de représenter tous les nombres décimaux possibles.

Avec une précision par défaut de 28 décimales, on remarque ainsi qu'il y a une perte de précision quand il y a une trop grande distance entre le chiffre le plus à gauche et celui le plus à droite, qui se ressent lors des opérations suivantes.

```
1 >>> Decimal('1.0000000000000000000000000001') * 2
2 Decimal('2.0000000000000000000000000002')
3 >>> Decimal('1.0000000000000000000000000001') * 2
4 Decimal('2.0000000000000000000000000000')
```

La précision des décimaux peut cependant être connue et réglée à l'aide des fonctions `getcontext` et `setcontext` tel que décrit dans [la documentation du module](#) `decimal`.

```
1 >>> from decimal import getcontext
2 >>> ctx = getcontext()
3 >>> ctx.prec = 30
4 >>> Decimal('1.0000000000000000000000000001') * 2
5 Decimal('2.0000000000000000000000000002')
6 >>> ctx.prec = 1
7 >>> Decimal('1.01') + Decimal('1.01')
8 Decimal('2')
```

VIII.1.2.2. Nombres rationnels

Mais quelle que soit la précision choisie, celle-ci sera toujours finie (pour des raisons de performances), et on ne pourra donc pas représenter un nombre avec une infinité ou un trop grand nombre de décimales.

On ne peut pas représenter de façon exacte le nombre $\frac{1}{3}$ avec un `Decimal`.

En revanche, il existe un autre type pour représenter les nombres rationnels : le type `Fraction` du module `fractions`.

i

Pour rappel, un nombre rationnel est un nombre qui peut s'écrire comme le quotient (la fraction) entre deux nombres entiers, tels que $\frac{1}{3}$ (`1/3`), $\frac{15}{10}$ (`1.5`) ou encore $\frac{8}{2}$ (`4`).

Un objet `Fraction` s'instancie avec le numérateur et le dénominateur de la fraction et s'utilise

ensuite comme n'importe quel nombre.

```
1 >>> from fractions import Fraction
2 >>> Fraction(1, 3) + Fraction(1, 3)
3 Fraction(2, 3)
4 >>> Fraction(1, 3) * Fraction(3, 1)
5 Fraction(1, 1)
```

Les fractions sont elles aussi compatibles avec les entiers, et convertibles en `int` ou `float`.

```
1 >>> Fraction(1, 3) * 4
2 Fraction(4, 3)
3 >>> int(Fraction(4, 3))
4 1
5 >>> float(Fraction(4, 3))
6 1.3333333333333333
```

Ce type offre donc une précision infinie pour les nombres rationnels, mais avec un certain coût en performances. Ne l'utilisez donc que si vous avez besoin d'une précision exacte sur vos nombres, comme pour un solveur d'équations.

VIII.1.2.3. Hiérarchie des nombres

Les types de nombres sont généralement compatibles entre eux : il est possible d'exécuter une opération entre un entier et un flottant, comme entre un rationnel et un complexe. Mais qu'attendre du résultat d'une telle opération ?

On le sait, une opération entre un entier et un flottant renvoie un flottant, car c'est lui qui est le plus à même de stocker le résultat. En effet, $2 * 3.4$ ne pourra pas être représenté dans un entier.

Il existe en fait une hiérarchie entre les types numériques qui définit quel type doit être renvoyé lors d'une telle opération. Il s'agira toujours du type le plus haut dans la hiérarchie.

Cette hiérarchie reprend les notions d'ensembles de nombres en mathématiques : il y a les nombres complexes tout en haut, puis les réels, les rationnels, les relatifs et enfin les entiers naturels.

En Python, les complexes sont représentés par le type `complex`, les réels par `float`, les rationnels par `Fraction`, et les entiers relatifs et naturels par `int`.

Cela explique qu'une opération entre une fraction et un complexe renverra toujours un complexe.

```
1 >>> Fraction(1, 3) + 2j
2 (0.3333333333333333+2j)
```

Mais ce ne sont pas les seuls types de nombres que vous pourriez être amenés à manipuler, et certaines bibliothèques pourraient venir avec leurs propres types.

Pour autant, ces types se conformeraient à la hiérarchie présentée au-dessus car ils y feraient référence en utilisant les types abstraits (`Number`—nombre, `Complex`—complexe, `Real`—réel, `Rational`—rationnel, `Integral`—entier) définis dans le module `numbers`.

VIII. La bibliothèque standard

Les types abstraits ainsi définis permettent de savoir à quelle classe appartient à un nombre, à l'aide d'appels à `isinstance`.

```
1 >>> import numbers
2 >>> isinstance(4, numbers.Integral) # Les int sont des entiers
3 True
4 >>> isinstance(4, numbers.Real) # Mais ce sont aussi des réels
5 True
6 >>> isinstance(4, numbers.Number) # Ou tout simplement des nombres
   au sens large
7 True
8 >>> isinstance(4.2, numbers.Real) # Les flottants sont des réels
9 True
10 >>> isinstance(4.2, numbers.Rational) # Mais ne sont pas des
    rationnels
11 False
12 >>> isinstance(Fraction(1, 3), numbers.Rational) # Contrairement
    aux fractions
13 True
```

Les décimaux sont un cas un peu à part car ils ne s'inscrivent pas dans la hiérarchie, ils se situent quelque part entre les entiers et les rationnels. Aussi, les décimaux ne sont pas considérés comme des instances de `numbers.Real` ou `numbers.Rational`.

```
1 >>> isinstance(Decimal('0.1'), numbers.Number)
2 True
3 >>> isinstance(Decimal('0.1'), numbers.Real)
4 False
5 >>> isinstance(Decimal('0.1'), numbers.Rational)
6 False
```

Cela implique que les décimaux ne sont pas compatibles avec les autres types de la hiérarchie, et ne le sont en fait qu'avec les entiers.

VIII.1.2.4. Bibliothèques mathématiques

On a vu qu'il existait en Python le module `math` qui regroupe l'essentiel des fonctions mathématiques sur les nombres réels, et dont on peut retrouver la liste sur [la page de documentation dédiée](#) [↗](#).

On le sait moins, mais il existe aussi un module `cmath` pour des fonctions équivalentes dans le domaine des complexes.

```
1 >>> import math
2 >>> math.sqrt(2) # Racine carrée de 2
3 1.4142135623730951
4 >>> math.sqrt(-1)
5 Traceback (most recent call last):
```

```
6 File "<stdin>", line 1, in <module>
7 ValueError: math domain error
8 >>> import cmath
9 >>> cmath.sqrt(2)
10 (1.4142135623730951+0j)
11 >>> cmath.sqrt(-1)
12 1j
```

Le module étend donc le domaine de définition de certaines fonctions de `math` pour permettre de les appliquer à des nombres complexes. C'est le cas des fonctions trigonométriques ou exponentielles par exemple.

```
1 >>> cmath.cos(1+2j)
2 (2.0327230070196656-3.0518977991518j)
3 >>> cmath.exp(1j * cmath.pi)
4 (-1+1.2246467991473532e-16j)
```

Toutes ces fonctions sont à retrouver dans la [documentation du module `cmath`](#) .

Mais à propos de nombres, on trouve aussi le module `statistics` qui comme son nom l'indique fournit des outils de statistiques. On trouvera ainsi des fonctions pour calculer la moyenne (`mean`), la médiane (`median`), la variance (`variance`) ou encore l'écart type (`stdev`) d'une série de données.

```
1 >>> data = [1, 2, 2, 3, 4, 5, 5, 6, 7]
2 >>> statistics.mean(data)
3 3.888888888888889
4 >>> statistics.median(data)
5 4
6 >>> statistics.variance(data)
7 4.111111111111111
8 >>> statistics.stdev(data)
9 2.0275875100994063
```

Pour plus d'informations sur les outils fournis par ce module, je vous invite à vous reporter sur [sa documentation](#) .

VIII.1.3. Chemins et fichiers

On a déjà croisé la `pathlib` plus tôt pour tester l'existence d'un fichier. Mais ce module de la bibliothèque standard va bien au-delà et propose une ribambelle d'outils pour travailler avec les chemins et les fichiers.

Le module `pathlib` définit principalement le type `Path` ainsi que d'autres types qui en dépendent suivant l'implémentation. Ainsi, quand vous instanciez un objet `Path` vous obtiendrez une instance d'un autre type suivant votre système d'exploitation (`WindowsPath` pour Windows et `PosixPath` pour les autres systèmes).

```
1 >>> from pathlib import Path
2 >>> Path()
3 PosixPath('.')

```

VIII.1.3.1. Usage des chemins

On le voit, on peut instancier un `Path` sans argument, le chemin correspond alors au répertoire courant (c'est ce que signifie le point). Mais on peut aussi passer un chemin (relatif comme absolu) en argument pour obtenir un objet `Path` correspondant.

```
1 >>> Path('/')
2 PosixPath('/')
3 >>> Path('../subdir/file.py')
4 PosixPath('../subdir/file.py')

```

L'intérêt des objets `Path` est qu'ils sont composables entre-eux, à l'aide de l'opérateur `/` (qui représente la séparation entre répertoires).

```
1 >>> Path('a') / Path('b') / Path('c')
2 PosixPath('a/b/c')

```

Un raccourci permet d'ailleurs de composer des chemins directement avec des chaînes de caractères.

```
1 >>> Path('a') / 'b'
2 PosixPath('a/b')

```

Et les chemins de type `Path` sont bien sûr convertibles en chaînes de caractères via un appel explicite à `str`, ou en chaîne d'octets avec `bytes`

```
1 >>> path = Path('a')
2 >>> str(path)
3 'a'
4 >>> str(path / 'b')
5 'a/b'
6 >>> bytes(path)
7 b'a'

```

Ces objets peuvent aussi directement être utilisés pour certaines opérations qui attendent des chemins.

```
1 >>> with open(path, 'w') as f:
2 ...     f.write('hello')

```

```
3 ...  
4 5
```

VIII.1.3.2. Propriétés des chemins

Les objets `Path` sont pourvus de nombreux attributs et méthodes et j'aimerais vous en présenter les plus importants.

VIII.1.3.2.1. `parts`

Premièrement il est possible d'accéder à la décomposition d'un chemin à l'aide de son attribut `parts`. On obtient ainsi un tuple des répertoires / fichiers qui composent notre chemin.

```
1 >>> path.parts  
2 ('a',)  
3 >>> Path('../subdir/file.py').parts  
4 ('..', 'subdir', 'file.py')
```

VIII.1.3.2.2. `name`

L'attribut `name` renvoie la dernière partie du chemin, soit le nom du fichier cible.

```
1 >>> path.name  
2 'a'  
3 >>> Path('../subdir/file.py').name  
4 'file.py'
```

VIII.1.3.2.3. `suffix`, `stem` et `suffixes`

`suffix` renvoie le suffixe d'un chemin, plus communément appelé l'extension du fichier. Si aucune extension n'est présente, `suffix` renvoie une chaîne vide.

```
1 >>> path.suffix  
2 ''  
3 >>> Path('../subdir/file.py').suffix  
4 '.py'
```

À l'inverse, l'attribut `stem` renvoie le nom du fichier dépourvu du suffixe.

```
1 >>> path.stem  
2 'a'  
3 >>> Path('../subdir/file.py').stem  
4 'file'
```

VIII. La bibliothèque standard

Si un chemin contient plusieurs extensions (`.tar.gz` par exemple), seule la dernière extension sera renvoyée par `suffix` (et retirée de `stem`). L'attribut `suffixes` permet alors de récupérer la liste de toutes les extensions.

```
1 >>> Path('photos.tar.gz').suffix
2 '.gz'
3 >>> Path('photos.tar.gz').stem
4 'photos.tar'
5 >>> Path('photos.tar.gz').suffixes
6 ['.tar', '.gz']
```

VIII.1.3.2.4. parent et parents

On peut accéder au parent d'un chemin (son répertoire parent) via l'attribut `parent`. `parent` est en quelque sorte l'inverse de `name`.

```
1 >>> path.parent
2 PosixPath('.')
3 >>> Path('../subdir/file.py').parent
4 PosixPath('../subdir')
```

L'attribut `parents` permet aussi d'accéder à l'ensemble des parents d'un chemin. `path.parents[0]` correspondra ainsi à `path.parent`, `path.parents[1]` à `path.parent.parent`, etc.

```
1 >>> Path('../subdir/file.py').parents[0]
2 PosixPath('../subdir')
3 >>> Path('../subdir/file.py').parents[1]
4 PosixPath('.')
5 >>> Path('../subdir/file.py').parents[2]
6 PosixPath('.')
```



Attention, l'attribut `parents` ne renvoie pas une liste mais un type particulier de séquence. On peut bien sûr le convertir en liste avec un appel à `list`.

```
1 >>> Path('../subdir/file.py').parents
2 <PosixPath.parents>
3 >>> list(Path('../subdir/file.py').parents)
4 [PosixPath('../subdir'), PosixPath('.'), PosixPath('.')]
```

VIII.1.3.2.5. is_absolute

La méthode `is_absolute` est un prédicat pour savoir si un chemin est absolu (débuté par la racine du système de fichiers) ou non.

```
1 >>> Path('dir/hello.txt').is_absolute()
2 False
3 >>> Path('/home/antoine/dir/hello.txt').is_absolute()
4 True
```

VIII.1.3.2.6. `relative_to`

La méthode `relative_to` permet de convertir un chemin pour l'obtenir relativement à un autre. Elle offre ainsi un moyen de convertir un chemin absolu vers un chemin relatif. Par exemple le chemin `/home/antoine/dir/hello.txt` donne `dir/hello.txt` relativement à `/home/antoine`.

```
1 >>> Path('/home/antoine/dir/hello.txt').relative_to(
2     '/home/antoine')
3 PosixPath('dir/hello.txt')
```

Mais on peut aussi le calculer à partir de chemins relatifs.

```
1 >>> Path('dir/hello.txt').relative_to('dir')
2 PosixPath('hello.txt')
```

Dans le cas où aucune correspondance n'est trouvée et qu'il n'est donc pas possible de construire un chemin relatif entre les deux, la méthode lève une exception `ValueError`.

De même si on mélange chemins absolus et relatifs.

```
1 >>> Path('dir/hello.txt').relative_to('dir2')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "/usr/lib/python3.9/pathlib.py", line 939, in relative_to
5     raise ValueError("{!r} is not in the subpath of {!r}"
6 ValueError: 'dir/hello.txt' is not in the subpath of 'dir2' OR one
   path is relative and the other is absolute.
```

VIII.1.3.3. Méthodes concrètes

Toutes les méthodes précédentes permettent de manipuler les chemins de façon abstraite, déconnectée du système de fichiers. Mais d'autres méthodes servent à réaliser des opérations concrètes en s'appuyant sur le système.

VIII.1.3.3.1. `exists`

Nous l'avons déjà rencontrée, la méthode `exists` est un prédicat pour tester si le chemin pointe vers un fichier/répertoire qui existe ou non.

```
1 >>> Path('notfound').exists()
2 False
3 >>> Path('hello.txt').exists()
4 True
5 >>> Path('/').exists()
6 True
```

VIII.1.3.3.2. `is_dir` et `is_file`

Les méthodes `is_dir` et `is_file` permettent respectivement de tester si un chemin pointe vers un répertoire ou vers un fichier.

```
1 >>> Path('hello.txt').is_dir()
2 False
3 >>> Path('hello.txt').is_file()
4 True
5 >>> Path('/').is_dir()
6 True
7 >>> Path('/').is_file()
8 False
```

Ces méthodes renvoient `False` quand le chemin n'existe pas.

```
1 >>> Path('notfound').is_dir()
2 False
3 >>> Path('notfound').is_file()
4 False
```

VIII.1.3.3.3. `resolve`

La méthode `resolve` permet de résoudre un chemin, soit de trouver le chemin absolu correspondant.

```
1 >>> path.resolve()
2 PosixPath('/home/antoine/a')
3 >>> Path('hello.txt').resolve()
4 PosixPath('/home/antoine/hello.txt')
5 >>> Path('../subdir/file.py').resolve()
6 PosixPath('/home/subdir/file.py')
7 >>> Path('/').resolve()
8 PosixPath('/')
```

Elle peut s'utiliser avec un argument `strict` pour lever une erreur si le chemin en question n'existe pas.


```
1 >>> Path('notfound').resolve()
2 PosixPath('/home/antoine/notfound')
3 >>> Path('hello.txt').resolve(strict=True)
4 PosixPath('/home/antoine/hello.txt')
5 >>> Path('notfound').resolve(strict=True)
6 Traceback (most recent call last):
7   [... ]
8 FileNotFoundError: [Errno 2] No such file or directory:
   '/home/antoine/notfound'
```

VIII.1.3.3.4. `cwd`

`cwd` est une méthode du type `Path`, qui renvoie le chemin vers le répertoire courant. C'est ce chemin qui est utilisé pour les résolutions de `resolve`.

```
1 >>> Path.cwd()
2 PosixPath('/home/antoine')
```

VIII.1.3.4. Méthodes pour les répertoires

Certaines méthodes sont spécifiques aux chemins représentant des répertoires.

VIII.1.3.4.1. `mkdir` et `rmdir`

La méthode `mkdir` permet de créer un répertoire là où pointe le chemin.

```
1 >>> Path('subdir').exists()
2 False
3 >>> Path('subdir').mkdir()
4 >>> Path('subdir').exists()
5 True
6 >>> Path('subdir').is_dir()
7 True
```

La méthode lève une erreur `FileExistsError` si le répertoire (ou un fichier) existe déjà à ce chemin.

À l'inverse, la méthode `rmdir` permet de supprimer le répertoire pointé, et lève une erreur `FileNotFoundError` s'il n'existe pas.

```
1 >>> Path('subdir').rmdir()
2 >>> Path('subdir').rmdir()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "/usr/lib/python3.9/pathlib.py", line 1363, in rmdir
```

```
6 self._accessor.rmdir(self)
7 FileNotFoundError: [Errno 2] No such file or directory: 'subdir'
```

Un répertoire doit être vide pour pouvoir être supprimé par `rmdir`.

VIII.1.3.4.2. `iterdir`

Le principe des répertoires, c'est de contenir des fichiers. Ainsi les chemins possèdent une méthode `iterdir` qui renvoie un itérable pour parcourir les fichiers contenus dans le dossier.

```
1 >>> for p in Path('.').iterdir():
2     ...     print(p)
3     ...
4 hello.txt
5 subdir
6 game.py
```



Comme on le voit, les fichiers que l'on obtient ne sont pas particulièrement triés. On peut toujours faire appel à `sorted` si cela est nécessaire.



Le parcours n'est pas récursif, les fichiers contenus dans les sous-dossiers (`subdir` par exemple) ne sont donc pas explorés.

VIII.1.3.4.3. `glob`

`glob` est une autre méthode pour explorer les fichiers présents dans un dossier, qui permet de les rechercher selon un critère. En effet, `glob` prend une chaîne de caractères en argument qui décrit quels fichiers rechercher dans le répertoire.

Cette chaîne doit correspondre à un nom de fichier mais peut comprendre des `*` qui agissent comme des jokers et correspondent à n'importe quels caractères. Ainsi, `*.py` permet de trouver tous les fichiers `.py` d'un répertoire, et `glob('*')` est équivalent à `iterdir()`.

```
1 >>> for p in Path('.').glob '*.py':
2     ...     print(p)
3     ...
4 game.py
```

VIII.1.3.5. Méthodes pour les fichiers

Certaines autres méthodes sont spécifiques aux fichiers.

VIII.1.3.5.1. `touch` et `unlink`

`touch` est la méthode qui permet de créer le fichier pointé par le chemin.

```
1 >>> Path('newfile.txt').exists()
2 False
3 >>> Path('newfile.txt').touch()
4 >>> Path('newfile.txt').exists()
5 True
```

La méthode ne produit pas d'erreur si le fichier existe déjà (et elle modifiera sa date de dernière modification).

```
1 >>> Path('newfile.txt').touch()
```

Et dans l'autre sens, on trouve la méthode `unlink` pour supprimer un fichier.

```
1 >>> Path('newfile.txt').unlink()
2 >>> Path('newfile.txt').exists()
3 False
```

La méthode lève une exception `FileNotFoundError` si le fichier n'existe pas, mais depuis Python 3.8 il est possible de lui préciser un argument booléen `missing_ok` pour ne pas produire d'erreur.

```
1 >>> Path('newfile.txt').unlink()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   File "/usr/lib/python3.9/pathlib.py", line 1354, in unlink
5     self._accessor.unlink(self)
6   FileNotFoundError: [Errno 2] No such file or directory:
   'newfile.txt'
7 >>> Path('newfile.txt').unlink(missing_ok=True)
```

VIII.1.3.5.2. `open`

Nous l'avons déjà vu, la méthode `open` est semblable à la fonction *built-in* `open`, sauf qu'elle s'applique à un chemin. La méthode prend alors un mode en argument, `'r'` par défaut, et renvoie un gestionnaire de contexte sur le fichier.

```
1 >>> with path.open('w') as f_out:
2     ...     f_out.write('coucou')
3     ...
4 6
5 >>> with path.open() as f_in:
6     ...     print(f_in.read())
7     ...
8 coucou
```

VIII.1.3.5.3. `read_text` et `read_bytes`

Pour simplifier certaines opérations, il existe aussi des méthodes `read_text` et `read_bytes` pour lire dans une chaîne le contenu d'un fichier.

`read_text` renvoie une chaîne de caractères et `read_bytes` une chaîne d'octets.

```
1 >>> path.read_text()
2 'coucou'
3 >>> path.read_bytes()
4 b'coucou'
```

VIII.1.3.5.4. `write_text` et `write_bytes`

Et réciproquement, les méthodes `write_text` et `write_bytes` permettent de remplacer le contenu d'un fichier par la chaîne donnée en argument.

```
1 >>> path.write_text('bonne soirée')
2 12
3 >>> path.read_text()
4 'bonne soirée'
5 >>> path.write_bytes(b'\x01\x02\x03')
6 3
7 >>> path.read_bytes()
8 b'\x01\x02\x03'
```

Le fichier est automatiquement créé s'il n'existe pas.

```
1 >>> Path('notfound').write_text('abc')
2 3
3 >>> Path('notfound').read_text()
4 'abc'
```

Toutes les autres méthodes des objets `Path` sont à découvrir sur [la page de documentation du module `pathlib`](#) [↗](#).

VIII.1.4. Modules systèmes

Python dispose aussi de modules pour interagir avec le système, notamment les modules `sys`, `shutil` et `os`.

VIII.1.4.1. Module `sys`

`sys` est un module qui fournit différentes informations sur le système d'exploitation et l'interpréteur Python.

On trouve notamment des attributs `platform`, `version` et `version_info` pour connaître l'OS utilisé et la version de Python.

```
1 >>> import sys
2 >>> sys.platform
3 'linux'
4 >>> sys.version
5 '3.9.7 (default, Aug 31 2021, 13:28:12) \n[GCC 11.1.0]'
6 >>> sys.version_info
7 sys.version_info(major=3, minor=9, micro=7, releaselevel='final',
8 serial=0)
9 >>> sys.version_info.major, sys.version_info.minor
10 (3, 9)
```

On peut aussi accéder au chemin de l'exécutable Python (`executable`), ainsi qu'à la liste des arguments du programme (`argv`).

```
1 >>> sys.executable
2 '/usr/bin/python'
3 >>> sys.argv
4 ['']
```

Le module met à disposition les fichiers `stdin`, `stdout` et `stderr` qui sont liés respectivement à l'entrée standard, la sortie standard et la sortie d'erreur.

```
1 >>> sys.stdin.readline()
2 hello
3 'hello\n'
4 >>> sys.stdout.write('coucou\n')
5 coucou
6 7
7 >>> sys.stderr.write('error\n')
8 error
9 6
```

Le dictionnaire `modules` référence tous les modules importés au sein de l'interpréteur. C'est un mécanisme de cache au sein de Python pour éviter de charger plusieurs fois un même module.

```
1 >>> sys.modules
2 {'sys': <module 'sys' (built-in)>, 'builtins': <module 'builtins'
3 (built-in)>, ...}
4 >>> sys.modules['sys']
5 <module 'sys' (built-in)>
6 >>> sys.modules['sys'].platform
7 'linux'
```

Quand je vous parlais de récursivité, j'évoquais une limite au nombre de récursions autorisées par l'interpréteur Python. Cette limite peut être connue via un appel à la fonction `getrecursionlimit` du module `sys`.

```
1 >>> sys.getrecursionlimit()
2 1000
```

Enfin, nous l'avons déjà rencontrée, la fonction `exit` permet de couper le programme en cours d'exécution. Utilisée sans argument, la fonction coupe le programme normalement avec un code de retour de 0 (signifiant que tout s'est bien passé).

```
1 >>> sys.exit()
2 % echo $?
3 0
```

Avec un nombre en argument, c'est ce nombre qui sera utilisé comme code de retour (un code de retour différent de 0 signifie que le programme s'est terminé sur une erreur).

```
1 >>> sys.exit(12)
2 % echo $?
3 12
```

Avec une chaîne de caractères en argument, la chaîne sera écrite sur la sortie d'erreur et le code de retour sera 1.

```
1 >>> sys.exit('error')
2 error
3 % echo $?
4 1
```

L'ensemble de ces fonctions, et bien d'autres encore, peut être retrouvé sur [la page de documentation du module sys](#) [↗](#).

VIII.1.4.2. Module `shutil`

`shutil` peut venir en complément de `pathlib`, il propose des opérations de haut-niveau sur les fichiers et répertoires, notamment pour les copies, les déplacements et la suppression.

On trouve ainsi une fonction `copy` qui permet de copier un fichier à un autre endroit sur le système. La fonction prend en arguments le chemin source et sa destination, les chaînes de caractères et les objets `Path` sont acceptés.

Elle renvoie le chemin du fichier copié.

```
1 >>> from pathlib import Path
2 >>> import shutil
3 >>> shutil.copy(Path('hello.txt'), 'new.txt')
4 'new.txt'
5 >>> Path('new.txt').read_text()
6 'salut\n'
```

VIII. La bibliothèque standard

Il est aussi possible de préciser un répertoire en second argument pour copier le fichier (en conservant son nom) vers ce répertoire.

```
1 >>> shutil.copy('hello.txt', 'subdir')
2 'subdir/hello.txt'
3 >>> Path('subdir/hello.txt').read_text()
4 'salut\n'
```

Pour copier des arborescences de fichiers (fichiers et répertoires), `shutil` propose une fonction `copytree` sur le même principe que `copy`. La fonction copie récursivement le répertoire source et les fichiers qu'il contient vers la destination.

```
1 >>> shutil.copytree('subdir', 'newdir')
2 'newdir'
3 >>> list(Path('newdir').iterdir())
4 [PosixPath('newdir/hello.txt'), PosixPath('newdir/file.py')]
```

De la même manière, on trouve une fonction `move` pour déplacer un fichier ou un répertoire vers une destination.

```
1 >>> shutil.move('new.txt', 'moved.txt')
2 'moved.txt'
3 >>> Path('moved.txt').read_text()
4 'salut\n'
5 >>> Path('new.txt').exists()
6 False
7 >>> shutil.move('newdir', 'moveddir')
8 'moveddir'
9 >>> list(Path('moveddir').iterdir())
10 [PosixPath('moveddir/hello.txt'), PosixPath('moveddir/file.py')]
11 >>> Path('newdir').exists()
12 False
```

Et le module offre aussi une fonction `rmtree` pour supprimer récursivement un répertoire.

```
1 >>> shutil.rmtree('moveddir')
2 >>> Path('moveddir').exists()
3 False
```

Enfin, dans un tout autre genre, la fonction `get_terminal_size` permet de connaître la taille (en lignes de caractères et en colonnes) du terminal. La fonction renvoie un tuple nommé avec deux champs `columns` et `lines`.

```
1 >>> shutil.get_terminal_size()
2 os.terminal_size(columns=136, lines=66)
```

La [page de documentation de `shutil`](#) [complètera les informations au sujet de ce module.](#)

VIII.1.4.3. Module `os`

`os` est l'interface bas-niveau du système d'exploitation (*os* pour *operating system*), le module offre une multitude de fonctions pour communiquer avec lui.

On trouve notamment des fonctions pour manipuler les fichiers et répertoires telles que `mkdir`, `rmdir`, `unlink`, `open`, etc. Ces fonctions sont celles qui sont utilisées par la `pathlib` qui leur ajoute une interface plus haut-niveau pour manipuler ces données.

La plupart des fonctions exposées dans `os` sont d'ailleurs abstraites dans d'autres modules (`subprocess`, `shutil`) pour les rendre plus faciles à utiliser.



De la même manière, on trouve le module `os.path`, antérieur à la `pathlib`, pour gérer les chemins de fichiers avec des fonctions comme `exists`, `dirname`, `basename` ou encore `splitext`.

Le module propose aussi une fonction `chdir` (pour *change directory*) qui prend un chemin (relatif ou absolu) en argument et permet de changer le répertoire courant.

```
1 >>> Path.cwd()
2 PosixPath('/home/antoine')
3 >>> os.chdir('..')
4 >>> Path.cwd()
5 PosixPath('/home')
```



Attention, changer de répertoire courant affecte ensuite toutes les opérations utilisant des chemins relatifs, c'est une opération à réaliser avec précaution.

Parmi les autres outils présents dans le module, on trouve par exemple la fonction `cpu_count` qui permet de savoir combien de cœurs sont disponibles sur la machine.

```
1 >>> os.cpu_count()
2 8
```

VIII.1.4.3.1. Gestion de l'environnement

Un programme est toujours exécuté dans un certain environnement. Cet environnement consiste en un ensemble de variables définies par le système, sur lesquelles les programmes peuvent se baser pour certaines de leurs actions.

Il est ainsi courant de trouver des variables d'environnement telles que `SHELL` (le shell utilisé), `USER` (l'utilisateur courant), `LANG` (la langue de l'utilisateur), `HOME` (le dossier de l'utilisateur) ou `PWD` (le répertoire courant).

Depuis le shell, on peut spécifier des variables d'environnement supplémentaires pour un programme en plaçant `VAR=value` avant l'invocation du programme.


```
1 % OUTPUT=/tmp/out MAX_VALUE=256 python script.py
```

i

Il est coutume d'utiliser exclusivement des lettres capitales (ainsi que des chiffres et des *underscores*) dans les noms de variables d'environnement.

En Python, l'environnement est accessible via le dictionnaire `environ` du module `os`. Ce dictionnaire associe les valeurs des variables d'environnement à leurs noms.

```
1 >>> import os
2 >>> os.environ
3 environ({'...', 'OUTPUT': '/tmp/out', 'MAX_VALUE': '256'})
```

On le voit, les valeurs des variables d'environnement sont toujours des chaînes de caractères, il peut alors être nécessaire de les convertir.

```
1 >>> os.environ['MAX_VALUE']
2 '256'
3 >>> int(os.environ['MAX_VALUE'])
4 256
```

Le module dispose aussi d'une fonction `getenv` pour récupérer une variable d'environnement.

```
1 >>> os.getenv('OUTPUT')
2 '/tmp/out'
```

La fonction renvoie `None` si la variable d'environnement n'est pas définie, mais il est possible de lui spécifier un argument `default` pour choisir cette valeur par défaut.

```
1 >>> os.getenv('NOTFOUND')
2 >>> os.getenv('NOTFOUND', 'no')
3 'no'
```

Le dictionnaire `environ` est bien sûr éditable, ce qui permet de faire évoluer l'environnement du programme.

```
1 >>> os.environ['MAX_VALUE'] = str(int(os.environ['MAX_VALUE']) * 2)
2 >>> os.getenv('MAX_VALUE')
3 '512'
```

Afin de traiter l'environnement comme des chaînes d'octets, on trouve aussi le dictionnaire `environb` et la fonction `getenvb` qui remplissent le même rôle que `environ` et `getenv`.

```
1 >>> os.environb
2 environ({'...', b'OUTPUT': b'/tmp/out', b'MAX_VALUE': b'512'})
3 >>> os.getenvb(b'MAX_VALUE')
4 b'512'
```

Pour plus d'informations, vous pouvez consulter [la documentation du module `os`](#) .

VIII.2. Un peu d'aléatoire

Introduction

Actuellement notre jeu de combat se joue à deux joueurs. C'est très bien, mais que diriez vous de pouvoir y jouer en solo ?

Pour cela il va nous falloir réaliser une «intelligence artificielle», très basique. Et pour arriver à nos fins et rendre le tout moins prédictible, on va y insérer des comportements aléatoires.

Comment gérer de l'aléatoire sur une machine aussi déterministe qu'un ordinateur ? C'est ce que nous allons voir ici.

VIII.2.1. Le module random

La bibliothèque standard de Python comprend un module `random` dédié aux opérations aléatoires.

Nous sommes sur un ordinateur et l'aléatoire n'est pas réellement possible¹ mais il existe une astuce. Cette astuce ce sont les générateurs pseudo-aléatoires.

Ces générateurs sont des outils produisant des suites de nombres qui semblent aléatoirement tirés. Pour cela ils s'appuient sur des paramètres extérieurs tels que le temps ou le statut des périphériques afin d'initialiser leur état, puis sur des opérations mathématiques pour générer un nombre en fonction des précédents.

En pratique ça fonctionne bien, mais attention : deux générateurs qui seraient initialisés avec la même valeur produiraient exactement les mêmes nombres.

Certains langages vous demandent d'initialiser le générateur pseudo-aléatoire avant de commencer à faire des tirages, mais Python le fait pour nous lors de l'import du module `random`, et est donc directement utilisable.

```
1 >>> import random
```

Le module propose de nombreuses fonctions, mais nous n'allons nous intéresser qu'à certaines d'entre elles.

VIII.2.1.1. Nombres aléatoires

Premièrement, le plus simple, les fonctions pour tirer un nombre entier aléatoire, tel un lancer de dé. Il y en a deux, `randrange` et `randint`.

La première reçoit entre 1 et 3 arguments, comme la fonction `range`, formant donc un intervalle avec une valeur de début (0 si omise), de fin et un pas (1 si omis). Elle renvoie un nombre aléatoire compris dans cet intervalle (pour rappel, la valeur de fin est exclue de l'intervalle).

Voici par exemple des tirages de nombres entre 1 et 6 (inclus).

1. À moins d'utiliser un périphérique dédié.

```
1 >>> random.randrange(1, 7)
2 5
3 >>> random.randrange(1, 7)
4 4
5 >>> random.randrange(1, 7)
6 2
```

Ce qui est d'ailleurs strictement équivalent à :

```
1 >>> random.randrange(6) + 1
2 3
3 >>> random.randrange(6) + 1
4 2
```

(bien sûr, vous n'obtiendrez pas nécessairement les mêmes résultats que les exemples)

Si l'on ne souhaitait tirer que des valeurs de dé impaires, on pourrait ajouter un pas à notre appel.

```
1 >>> random.randrange(1, 7, 2)
2 5
3 >>> random.randrange(1, 7, 2)
4 1
5 >>> random.randrange(1, 7, 2)
6 3
```

La fonction `randint` est un peu similaire si ce n'est qu'elle prend deux arguments (ni plus ni moins) et qu'elle retourne un nombre de cet intervalle, bornes incluses. Ainsi, notre tirage de dé se ferait comme suit.

```
1 >>> random.randint(1, 6)
2 6
3 >>> random.randint(1, 6)
4 4
```

VIII.2.1.2. Opérations aléatoires

Mais tirer un nombre aléatoire ce n'est pas tout, et le module propose d'autres opérations aléatoires intéressantes.

Par exemple, la fonction `choice` permet de sélectionner aléatoirement un élément dans une liste.

```
1 >>> actions = ['manger', 'dormir', 'aller au ciné']
2 >>> random.choice(actions)
3 'manger'
```

```
4 >>> random.choice(actions)
5 'aller au ciné'
```

Je parle de liste, mais tout objet se comportant comme une liste² est aussi accepté, les `range` par exemple. Ainsi, `random.choice(range(1, 7))` est équivalent à `random.randrange(1, 7)`.

```
1 >>> random.choice(range(1, 7))
2 3
```

Si vous souhaitez tirer plusieurs valeurs sans remise, `choice` ne sera pas adaptée, vous risqueriez de tirer plusieurs fois la même.

```
1 >>> random.choice(actions)
2 'manger'
3 >>> random.choice(actions)
4 'manger'
```

Dans ce cas orientez-vous vers `sample`, qui prend en argument le nombre de valeurs à tirer en plus de la liste.

```
1 >>> random.sample(actions, 2)
2 ['dormir', 'manger']
```

Enfin, la fonction `shuffle` permet de simplement trier aléatoire la liste (elle modifie la liste reçue en paramètre).

```
1 >>> random.shuffle(actions)
2 >>> actions
3 ['aller au ciné', 'manger', 'dormir']
4 >>> random.shuffle(actions)
5 >>> actions
6 ['dormir', 'aller au ciné', 'manger']
```

C'est utile pour mélanger un paquet de cartes ou d'autres opérations du genre, et avoir ensuite un tirage sans remise.

```
1 >>> cards = ['as de pique', '3 de trèfle', '7 de carreau',
2             'dame de cœur']
3 >>> random.shuffle(cards)
4 >>> cards.pop()
5 '3 de trèfle'
6 >>> cards.pop()
```

2. C'est-à-dire ayant une taille et permettant d'accéder à n'importe quel élément à partir de son index.

```
6 '7 de carreau'
```

VIII.2.2. Distributions

VIII.2.2.1. Loïs de distribution

Voilà pour ce qui est des tirages dit discrets (on a un ensemble de valeurs connues et on veut tirer une valeur dans celles-ci) mais il est aussi possible de tirer des nombres dans des intervalles continus.

Par exemple, très simple, la fonction `random` va renvoyer un nombre flottant entre 0 et 1 (1 étant exclu de l'intervalle).

```
1 >>> random.random()
2 0.9294919627802888
3 >>> random.random()
4 0.47588843177000617
```

Le tirage de ce nombre est uniforme, grossièrement cela veut dire qu'on a autant de chances de tirer un nombre n'importe où dans l'intervalle.

Une fonction est spécifiquement dédiée au tirage uniforme entre deux nombres flottants, il s'agit de la fonction `uniform`.

```
1 >>> random.uniform(0, 10)
2 1.4017486291855232
3 >>> random.uniform(0, 10)
4 5.926447309804371
```

Suivant les arrondis, la borne supérieure peut être incluse ou non dans l'intervalle, mais cela a peu d'importance : il est pratiquement impossible de tomber sur ce nombre précis, puisqu'il y en a une infinité¹.

On a l'habitude de présenter une distribution par sa densité de probabilité, la fonction qui montre quelles zones de l'intervalle ont plus de chances d'être sollicitées.

Dans le cas d'une distribution uniforme, cette densité est constante.

1. Pas exactement puisque la représentation d'un flottant est finie, mais vous comprenez l'idée.

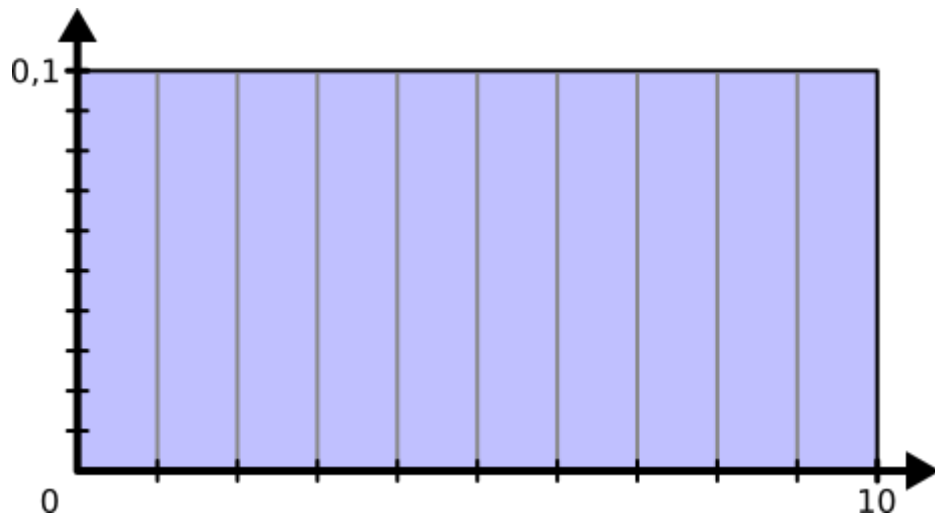


FIGURE VIII.2.1. – Densité de probabilité d'une distribution uniforme.

D'autres distributions sont possibles pour les tirages de nombres flottants.

Il y a par exemple la distribution triangulaire accessible via la fonction `triangular`, qui prend en argument les deux bornes de l'intervalle.

On parle de distribution triangulaire car sa densité représente un triangle entre les deux bornes.

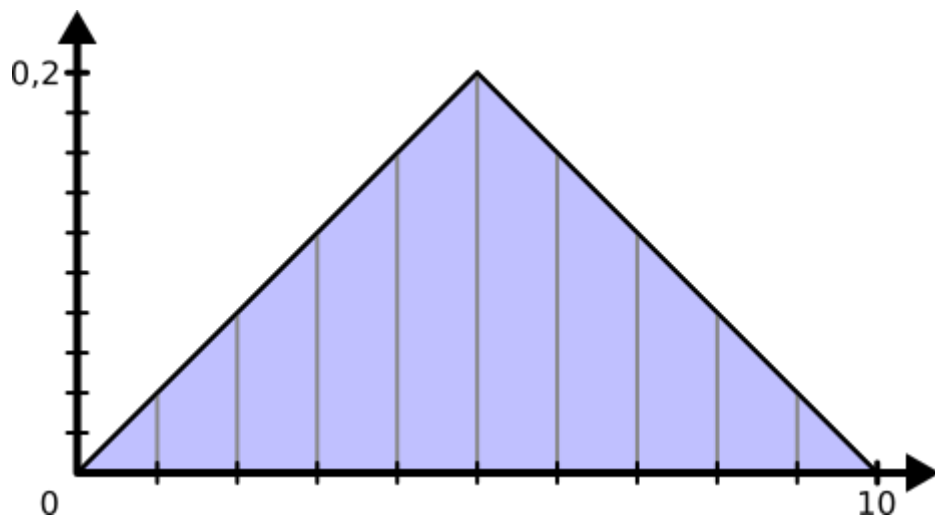


FIGURE VIII.2.2. – Densité de probabilité d'une distribution triangulaire.

Ainsi, les valeurs autour du sommet du triangle auront plus de probabilité d'être tirées que celles aux extrémités.

```
1 >>> random.triangular(0, 10)
2 4.0479535343895865
```

Un troisième argument optionnel, le mode, permet de spécifier la valeur du sommet du triangle (par défaut il s'agit du milieu de l'intervalle, 5 dans notre exemple).

```
1 >>> random.triangular(0, 10, 2)
2 2.4400405218007473
```

On trouve aussi la distribution normale, qui représente la distribution naturelle autour d'une moyenne avec un certain écart type. La moyenne et l'écart type sont les deux arguments de la fonction `normalvariate`.

```
1 >>> random.normalvariate(5, 1)
2 4.655500829738334
3 >>> random.normalvariate(5, 1)
4 5.808402224132684
```

Sa densité de probabilité prend la forme d'une cloche centrée autour de la moyenne. Plus on s'éloigne de la moyenne, moins les valeurs ont de chance d'être tirées.

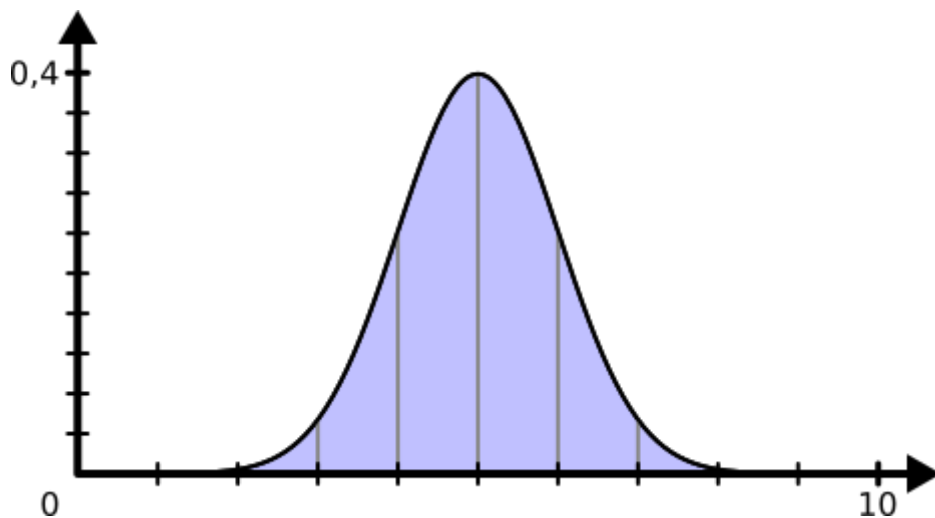


FIGURE VIII.2.3. – Densité de probabilité d'une distribution normale.

VIII.2.2.2. Pondération

Un autre point important à propos des tirages aléatoires concerne la pondération. En effet, les tirages discrets que nous avons effectués jusqu'ici étaient tous uniformes : chaque valeur avait autant de chance que les autres de tomber.

Avec `random.randint(1, 6)`, chaque valeur a une probabilité de $\frac{1}{6}$ d'être tirée. On peut d'ailleurs le vérifier en simulant un très grand nombre de tirages et en calculant le nombre d'occurrences de chaque valeur pour en déterminer la fréquence.

Si le tirage est bien uniforme, chaque valeur est censée être équitablement présente.

```
1 >>> from collections import Counter
2 >>> occurrences = Counter()
3 >>> N = 10000
4 >>> for _ in range(N):
5 ...     val = random.randint(1, 6)
6 ...     occurrences[val] += 1
7 ...
8 >>> for val, occ in sorted(occurrences.items()): # sorted pour
...     afficher selon l'ordre des clés
9 ...     print(f'{val}: {occ / N}')
```



```

10 ...
11 1: 0.1649
12 2: 0.1638
13 3: 0.1687
14 4: 0.1695
15 5: 0.1654
16 6: 0.1677

```

On voit que chaque fréquence est proche de 0,1666.

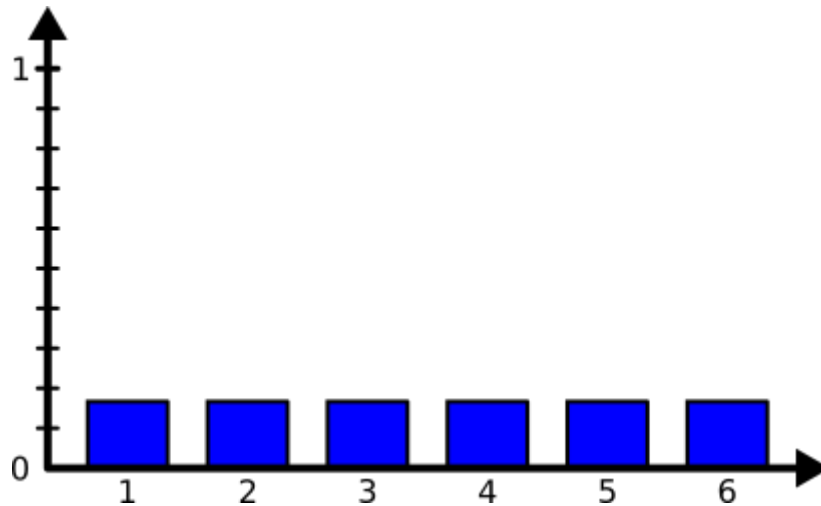


FIGURE VIII.2.4. – Distribution uniforme des valeurs.

Mais parfois on souhaiterait pouvoir pondérer notre tirage, affecter un poids différent à chaque valeur. Une manière de faire serait d'utiliser un `choice` et d'y mettre plusieurs fois les valeurs selon l'importance que l'on souhaite leur donner.

```

1 choices = [1, 2, 3, 4, 4, 5, 5, 6, 6, 6]

```

Ici, 6 a une probabilité de 0,3 ($\frac{3}{10}$) d'être tiré, 4 et 5 en ont une de 0,2 et les autres sont de 0,1.

```

1 >>> occurrences = Counter()
2 >>> for _ in range(N):
3 ...     val = random.choice(choices)
4 ...     occurrences[val] += 1
5 ...
6 >>> for val, occ in sorted(occurrences.items()):
7 ...     print(f'{val}: {occ / N}')
8 ...
9 1: 0.0982
10 2: 0.1021
11 3: 0.0968
12 4: 0.1982
13 5: 0.1985
14 6: 0.3062

```

VIII. La bibliothèque standard

Mais j'ai choisi un exemple facile, il est généralement assez compliqué de déterminer combien de valeurs on souhaite mettre en fonction de la probabilité que l'on veut leur donner, et cela peut amener à des listes de valeurs assez grandes.

Heureusement, Python a pensé à nous et propose une fonction qui prend directement en compte la pondération, il s'agit de la fonction `random.choices`.

Par défaut la fonction est semblable à `choice`, attribuant le même poids à chaque valeur, sauf qu'elle renvoie la valeur tirée sous forme d'une liste.

```
1 >>> random.choices(range(1, 7))
2 [6]
```

C'est parce qu'il est possible de lui demander de tirer plusieurs valeurs (avec remise) en utilisant le paramètre `k`.

```
1 >>> random.choices(range(1, 7), k=3)
2 [1, 1, 6]
```

Mais l'intérêt de cette fonction se situe dans son deuxième argument qui est une liste de poids, correspondant donc aux valeurs données en premier argument. Notre tirage de tout à l'heure pourrait se réécrire de la façon suivante :

```
1 >>> weights = [0.1, 0.1, 0.1, 0.2, 0.2, 0.3]
2 >>> random.choices(range(1, 7), weights)
3 [5]
```

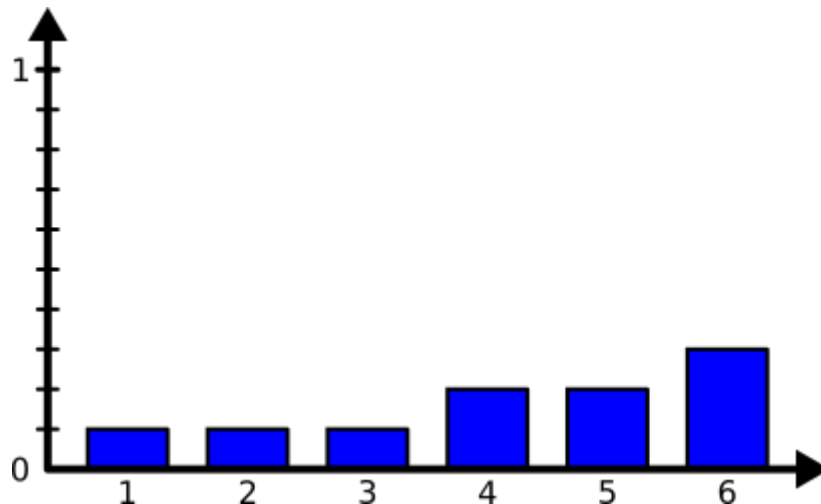


FIGURE VIII.2.5. – Pondération des valeurs.

Encore une fois, on peut le vérifier en calculant les fréquences d'apparition.

```
1 >>> occurrences = Counter()
2 >>> for _ in range(N):
```

```

3 ...     val = random.choices(range(1, 7), weights)[0] # Attention,
           choices renvoie une liste
4 ...     occurrences[val] += 1
5 ...
6 >>> for val, occ in sorted(occurrences.items()):
7 ...     print(f'{val}: {occ / N}')
8 ...
9 1: 0.0995
10 2: 0.1008
11 3: 0.1008
12 4: 0.2018
13 5: 0.2
14 6: 0.2971

```

J'ai utilisé ici des fréquences comme poids, mais il est possible d'utiliser n'importe quels nombres, Python calculera la fréquence en fonction de la somme des poids.

```

1 >>> weights = [1, 1, 1, 2, 2, 3]
2 >>> random.choices(range(1, 7), weights)
3 [2]

```

Enfin, il est aussi possible d'utiliser des poids cumulés pour le tirage. Dans ce cas, la fonction prend un paramètre `cum_weights` définissant ces poids.

Les poids cumulés peuvent être vus comme une règle graduée entre 0 et 1, chaque valeur se voyant attribuer une graduation. Un nombre est tiré entre 0 et 1, et c'est la valeur située juste à droite de cette graduation qui sera sélectionnée.

Notre tirage précédent peut alors s'écrire comme suit.

```

1 >>> cum_weights = [0.1, 0.2, 0.3, 0.5, 0.7, 1]
2 >>> random.choices(range(1, 7), cum_weights=cum_weights)
3 [5]

```

Je vous laisse calculer la fréquence des tirages pour le vérifier.

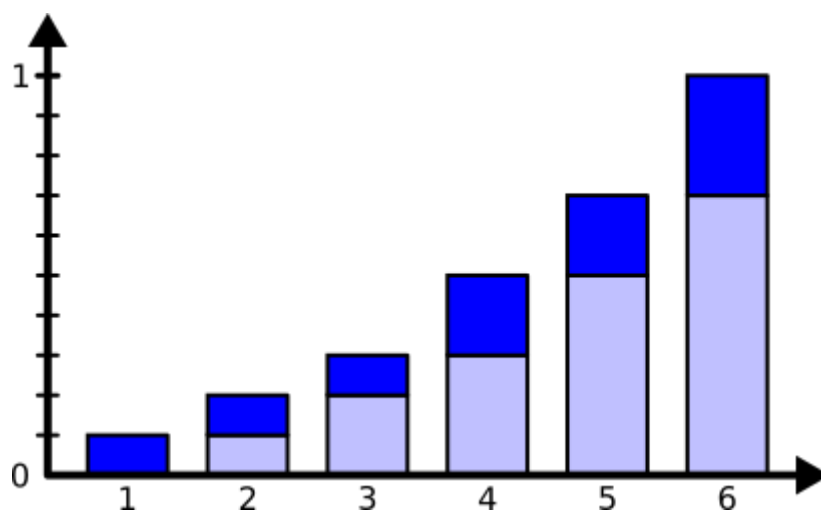


FIGURE VIII.2.6. – Poids cumulés.

VIII.3. Gestion du temps

Introduction

Différents modules de la bibliothèque standard permettent de gérer les dates et le temps, des données qui ne sont pas toujours faciles à manipuler en raison de différentes conventions (durée des mois, années bissextiles, fuseaux horaires, heure d'été, secondes intercalaires, etc.).

VIII.3.1. Module `time`

Il existe plusieurs manières de représenter le temps en informatique, selon que l'on parle d'instants ou de dates.

VIII.3.1.1. Les *timestamps*

La plus simple d'entre toutes, c'est le *timestamp* qui représente un instant selon le nombre de secondes écoulées depuis une date de référence appelée *epoch* (généralement le 1er janvier 1970 à minuit UTC, norme Unix). On y accède via la fonction `time` du module `time` qui nous renvoie un nombre flottant (incluant donc les fractions de secondes).

```
1 >>> import time
2 >>> time.time()
3 1633091908.9014919
4 >>> time.time()
5 1633091911.8008878
```

i

Le *timestamp* peut aussi être un nombre négatif pour représenter les instants antérieurs à cet *epoch*.

Si une plus grande précision est nécessaire, la fonction `time_ns` permet de récupérer le *timestamp* sous la forme d'un nombre entier de nanosecondes écoulées.

```
1 >>> time.time_ns()
2 1633093266497388151
```

Étant un nombre, le *timestamp* est très facile à manipuler, et n'est pas soumis aux problématiques sur la gestion des fuseaux horaires. Il est donc utile pour des problématiques d'horodatage (noter à quel instant s'est produit un événement).

Cependant, on ne peut pas le considérer comme portable car son interprétation dépend de la date utilisée comme *epoch*. Il n'est ainsi pas recommandé de transmettre un *timestamp* à un autre programme car celui-ci pourrait l'interpréter différemment.

De plus, le *timestamp* est un nombre soumis à un stockage limité : auparavant sur 32 bits (aujourd'hui sur 64) il ne permettait alors de ne représenter qu'un intervalle restreint de dates.

?

Peut-être avez-vous entendu parler du bug de l'an 2038 ? Il s'agit de la date où les *timestamps* Unix atteindront leur capacité maximale sur 32 bits, rendant leur usage impossible après cette date.

Mais d'ici-là, tout le monde devrait être passé aux *timestamps* 64 bits.

Aussi, on pourrait être tenté d'utiliser des *timestamps* pour mesurer des durées dans un programme, en calculant la différence entre les *timestamps*. C'est une mauvaise pratique car ceux-ci ne sont pas monotones : étant alignés sur l'horloge du système, le temps peut «revenir en arrière» si l'horloge est recalibrée.

Pour un tel cas d'usage, il faut alors plutôt faire appel à la fonction `monotonic` (ou `monotonic_ns`) qui est une horloge monotonique. Le nombre ainsi renvoyé est aussi un nombre de secondes (ou de nanosecondes) mais la date de référence est indéterminée, ils ne sont alors utiles que pour calculer des durées.

```
1 >>> start = time.monotonic()
2 >>> ... # différentes opérations
3 >>> time.monotonic() - start
4 14.564803410001332
```

VIII.3.1.2. Structure de temps

Une autre manière de représenter le temps est de stocker des données liées à une date : année, mois, jour, heure, minutes, secondes, etc. C'est ce que fait l'objet `struct_time` du module `time`.

On peut obtenir un objet `struct_time` en appelant la fonction `localtime` par exemple.

```
1 >>> time.localtime()
2 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1, tm_hour=15,
   tm_min=12, tm_sec=52, tm_wday=4, tm_yday=274, tm_isdst=1)
3 >>> date = time.localtime()
4 >>> date.tm_year
5 2021
6 >>> date.tm_hour
7 15
```

Il s'agit donc d'une représentation du temps plus exploitable, dont on peut explorer les différentes composantes. Mais un tel objet est alors dépendant du fuseau horaire (le fuseau local pour `localtime`) et des autres conventions sur les dates.

La fonction `gmtime` permet de récupérer le `struct_time` correspondant au temps courant dans le fuseau horaire UTC.

```
1 >>> time.gmtime()
2 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1, tm_hour=13,
  tm_min=15, tm_sec=3, tm_wday=4, tm_yday=274, tm_isdst=0)
```

VIII.3.1.3. Utilitaires du module

Le module `time` met aussi à disposition quelques utilitaires.

Ainsi, il est possible de mettre le programme en pause pendant une certaine durée (en secondes) à l'aide de la fonction `sleep`.

```
1 >>> time.sleep(3)
```

On trouve aussi certaines fonctions pour faire des conversions entre les types précédents. Ainsi la fonction `mktime` permet de transformer un objet `struct_time` (dans le fuseau courant) en un *timestamp*.

```
1 >>> time.mktime(date)
2 1633093972.0
```

Aussi, `localtime` et `gmtime` peuvent prendre un *timestamp* en argument et renvoyer la date associée (respectivement dans le fuseau local ou en UTC).

```
1 >>> time.localtime(1633093972.0)
2 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1, tm_hour=15,
  tm_min=12, tm_sec=52, tm_wday=4, tm_yday=274, tm_isdst=1)
3 >>> time.gmtime(1633093972.0)
4 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1, tm_hour=13,
  tm_min=12, tm_sec=52, tm_wday=4, tm_yday=274, tm_isdst=0)
```

Enfin, on trouve d'autres fonctions de calcul du temps dans le module, comme `process_time` (et `process_time_ns`) qui sert à calculer le nombre de secondes de travail effectif (excluant les pauses) du programme, ainsi que `perf_counter` (et `perf_counter_ns`) spécialement dédiée aux calculs de performance du programme avec une résolution adaptée (les dates de référence de ces différentes fonctions sont indéterminées).

```
1 >>> start = time.process_time()
2 >>> time.sleep(3)
3 >>> time.process_time() - start
4 0.00061256600000000088
5 >>> start = time.perf_counter()
6 >>> time.sleep(3)
7 >>> time.perf_counter() - start
8 3.0024995610001497
```

Et n'hésitez pas à jeter un œil à [la documentation du module `time`](#) pour aller plus loin.

VIII.3.2. Module `datetime`

Le module `datetime` fournit une interface haut-niveau pour gérer les temps et les dates, construit autour du module `time`, avec le type `datetime`.

Un objet `datetime` représente une date précise (avec année, mois, jour, heure, minutes, secondes et microsecondes), avec ou sans fuseau horaire.

Une date avec fuseau horaire représente donc un instant précis, on dit qu'elle est avisée. Une date sans fuseau est dite naïve car son interprétation dépend du fuseau horaire courant.

```
1 >>> datetime.datetime(2000, 4, 12, 8, 30, 55)
2 datetime.datetime(2000, 4, 12, 8, 30, 55)
```

La méthode `now` du type `datetime` permet de récupérer l'objet associé à l'instant courant (exprimé dans le fuseau local). Par défaut, elle renvoie une date naïve.

```
1 >>> dt = datetime.datetime.now()
2 >>> dt
3 datetime.datetime(2021, 10, 1, 16, 19, 43, 840744)
```

Il est possible de préciser un fuseau horaire en argument pour obtenir une date avisée selon ce fuseau, par exemple en utilisant `datetime.timezone.utc`.

```
1 >>> dt = datetime.datetime.now(datetime.timezone.utc)
2 >>> dt_utc
3 datetime.datetime(2021, 10, 1, 14, 19, 43, 840744,
   tzinfo=datetime.timezone.utc)
```

On voit que le fuseau horaire est stocké dans l'attribut `tzinfo` de l'objet `datetime`.



Le format `datetime` ne permet que de représenter un ensemble limité de dates : seules les années 1 à 9999 sont autorisées.

VIII.3.2.1. Conversions

Les `datetime` peuvent être convertis vers d'autres types de dates à l'aide de méthodes spécifiques :

- `datetime.fromtimestamp` permet de construire un objet `datetime` depuis un *timestamp*. Un fuseau optionnel peut être donné en argument.

```
1 >>> datetime.datetime.fromtimestamp(1633093972)
2 datetime.datetime(2021, 10, 1, 15, 12, 52)
```

```

3 >>> datetime.datetime.fromtimestamp(1633093972,
   datetime.timezone.utc)
4 datetime.datetime(2021, 10, 1, 13, 12, 52,
   tzinfo=datetime.timezone.utc)

```

- La méthode `timestamp` permet l'opération inverse (que l'objet `datetime` soit naïf ou avisé).

```

1 >>> dt.timestamp()
2 1633097983.840744
3 >>> dt_utc.timestamp()
4 1633097983.840744

```

- On peut aussi convertir des `datetime` vers des `struct_time` à l'aide de la méthode `timetuple`.

```

1 >>> dt.timetuple()
2 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1,
   tm_hour=16, tm_min=19, tm_sec=43, tm_wday=4, tm_yday=274,
   tm_isdst=-1)
3 >>> dt_utc.timetuple()
4 time.struct_time(tm_year=2021, tm_mon=10, tm_mday=1,
   tm_hour=14, tm_min=19, tm_sec=43, tm_wday=4, tm_yday=274,
   tm_isdst=-1)

```

Les conversions sont aussi possibles vers et depuis des chaînes de caractères, notamment en format ISO avec les méthodes `isoformat` et `fromisoformat`.

```

1 >>> dt.isoformat()
2 '2021-10-01T16:19:43.840744'
3 >>> dt_utc.isoformat()
4 '2021-10-01T14:19:43.840744+00:00'
5 >>> datetime.datetime.fromisoformat('2021-10-01T16:19:43.840744')
6 datetime.datetime(2021, 10, 1, 16, 19, 43, 840744)
7 >>> datetime.datetime.fromisoformat(
   '2021-10-01T14:19:43.840744+00:00')
8 datetime.datetime(2021, 10, 1, 14, 19, 43, 840744,
   tzinfo=datetime.timezone.utc)

```

Mais d'autres conversions en chaînes sont possibles, avec `strftime` par exemple. Cette méthode accepte une chaîne pour représenter le format de sortie, où différents codes de formatage sont disponibles comme :

- `%a` et `%A` pour le nom du jour de la semaine (forme abrégée ou forme longue)
- `%d` pour le numéro de jour dans le mois
- `%b` et `%B` pour le nom du mois (forme abrégée ou forme longue)
- `%m` pour le numéro du mois
- `%y` et `%Y` pour l'année (sur 2 ou 4 chiffres)

VIII. La bibliothèque standard

- %H, %M et %S respectivement pour les heures, minutes et secondes
- %z et %Z pour le fuseau horaire (en tant que décalage ou par son nom)

```
1 >>> dt.strftime('Le %A %d %B %Y à %Hh%M')
2 'Le vendredi 01 octobre 2021 à 16h19'
3 >>> dt_utc.strftime('Le %A %d %B %Y à %Hh%M (%Z)')
4 'Le vendredi 01 octobre 2021 à 14h19 (UTC)'
```



Il se peut que vous obteniez des noms anglais pour les jours et mois, cela est dû à la *locale* définie pour les conversions. Vous pouvez définir une *locale* française à l'aide des lignes suivantes :

```
1 import locale
2 locale.setlocale(locale.LC_ALL, 'fr_FR')
```

(fr_BE pour la Belgique et fr_CA pour le Canada sont aussi disponibles)

On notera que ces options de formatage sont aussi disponibles au sein des *fstrings* pour représenter des objets `datetime`.

```
1 >>> f'{dt:%d/%m/%Y %H:%M}'
2 '01/10/2021 16:19'
3 >>> f'{dt_utc:%d/%m/%Y %H:%M%z}'
4 '01/10/2021 14:19+0000'
```

L'opération inverse est elle aussi possible (mais plus compliquée) avec la méthode `strptime` : on spécifie le chaîne représentant la date et le format attendu en arguments, la méthode nous renvoie alors l'objet `datetime` correspondant.

```
1 >>> datetime.datetime.strptime('01/10/2021 14:19+0000',
2                                '%d/%m/%Y %H:%M%z')
3 datetime.datetime(2021, 10, 1, 14, 19,
4                   tzinfo=datetime.timezone.utc)
```

VIII.3.2.2. Durées

Il est possible de soustraire des objets `datetime` pour obtenir une durée, qui représente le nombre de jours et secondes qui séparent les deux dates.

```
1 >>> dt - datetime.datetime(2000, 4, 12, 8, 30, 55)
2 datetime.timedelta(days=7842, seconds=28128, microseconds=840744)
```

Ces durées se matérialisent par le type `timedelta`. Elles peuvent s'additionner et se soustraire entre-elles. Il est aussi possible de les multiplier par des nombres.

```
1 >>> datetime.timedelta(days=1) + datetime.timedelta(days=1)
2 datetime.timedelta(days=2)
3 >>> datetime.timedelta(days=1) - datetime.timedelta(days=1)
4 datetime.timedelta(0)
5 >>> datetime.timedelta(days=1) * 10
6 datetime.timedelta(days=10)
```

Et bien sûr, on peut additionner une durée à un `datetime` (naïf ou avisé) pour obtenir un nouveau `datetime`.

```
1 >>> dt + datetime.timedelta(days=1)
2 datetime.datetime(2021, 10, 2, 16, 19, 43, 840744)
3 >>> dt_utc + datetime.timedelta(days=1)
4 datetime.datetime(2021, 10, 2, 14, 19, 43, 840744,
   tzinfo=datetime.timezone.utc)
```

VIII.3.2.3. Fuseaux horaires

On l'a vu : les objets `datetime` peuvent contenir ou non des informations de fuseau horaire, selon l'usage que l'on veut en faire, et les deux types sont généralement gérés par les différentes fonctions.

Il est cependant à noter qu'on ne peut pas mélanger dates naïves et avisées au sein des mêmes opérations.

```
1 >>> dt_utc - dt
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: can't subtract offset-naive and offset-aware datetimes
```

Le module `datetime` ne fournit par défaut que le fuseau horaire UTC (*Temps Universel Coordonné*) avec `datetime.timezone.utc`.

Mais le type `timezone` permet de construire des fuseaux à décalage fixe par rapport à UTC, en prenant un `timedelta` en argument.

```
1 >>> tz = datetime.timezone(datetime.timedelta(seconds=3600))
2 >>> datetime.datetime.now(tz)
3 datetime.datetime(2021, 10, 1, 15, 19, 43, 840744,
   tzinfo=datetime.timezone(datetime.timedelta(seconds=3600)))
```

On notera aussi que la méthode `astimezone` permet de convertir une date vers un autre fuseau horaire. Les dates naïves sont considérées comme appartenant au fuseau local.

```

1 >>> dt.astimezone(tz)
2 datetime.datetime(2021, 10, 1, 15, 19, 43, 840744,
   tzinfo=datetime.timezone(datetime.timedelta(seconds=3600)))
3 >>> dt.astimezone(datetime.timezone.utc)
4 datetime.datetime(2021, 10, 1, 14, 19, 43, 840744,
   tzinfo=datetime.timezone.utc)
5 >>> dt_utc.astimezone(tz)
6 datetime.datetime(2021, 10, 1, 15, 19, 43, 840744,
   tzinfo=datetime.timezone(datetime.timedelta(seconds=3600)))

```

i

`astimezone` opère une conversion sur la date pour correspondre au fuseau horaire choisi. Pour simplement ajouter un fuseau horaire à une date sans faire de conversion, vous pouvez utiliser la méthode `replace` avec l'argument nommé `tzinfo`.

```

1 >>> dt.replace(tzinfo=datetime.timezone.utc)
2 datetime.datetime(2021, 10, 1, 16, 19, 43, 840744,
   tzinfo=datetime.timezone.utc)

```

Depuis Python 3.9, le module `zoneinfo` apporte une collection de fuseaux horaires pour traiter les fuseaux courants.

```

1 >>> from zoneinfo import ZoneInfo
2 >>> tz = ZoneInfo('Europe/Paris')
3 >>> dt.astimezone(tz)
4 datetime.datetime(2021, 10, 1, 16, 19, 43, 840744,
   tzinfo=zoneinfo.ZoneInfo(key='Europe/Paris'))

```

Pour plus d'informations sur ces modules, vous pouvez consulter les documentations de [date](#) [time](#) et [zoneinfo](#).

VIII.3.3. Module calendar

Le module `calendar` est un module qui sert principalement à afficher de simples calendriers dans le terminal.

```

1 >>> import calendar
2 >>> calendar.prmonth(2021, 10)
3     octobre 2021
4 lu ma me je ve sa di
5             1  2  3
6  4  5  6  7  8  9 10
7 11 12 13 14 15 16 17
8 18 19 20 21 22 23 24

```

```
9 | 25 26 27 28 29 30 31
```

Le module contient ainsi des fonctions `month` et `calendar`. La première prend une année et un mois en arguments et renvoie la représentation de ce mois. La seconde prend une année et renvoie la représentation de tous les mois de cette année.

Les tailles des lignes et des colonnes sont configurables à l'aide des différents paramètres de ces fonctions.

Les fonctions `prmonth` et `prcal` sont des raccourcis pour directement afficher ces représentations sur le terminal.

Le module apporte aussi différents attributs pour connaître les noms de jours et de mois :

- `day_name` est le tableau des noms de jours de la semaine
- `day_abbr` est celui des noms de jours abrégés
- `month_name` est le tableau des noms de mois
- `month_abbr` est celui des noms de mois abrégés

```
1 >>> calendar.day_name[0]
2 'lundi'
3 >>> calendar.day_abbr[1]
4 'mar.'
5 >>> calendar.month_name[3]
6 'mars'
7 >>> calendar.month_abbr[7]
8 'juil.'
```

Enfin, on trouve aussi dans ce module une fonction `timegm` qui permet de convertir un objet `struct_time` en *timestamp*.

```
1 >>> calendar.timegm(time.gmtime())
2 1633102464
```

D'autres fonctions sont encore disponibles dans le module, je vous laisse les découvrir [sur la page de documentation](#) [↗](#) .

Conclusion

La gestion du temps et des dates n'est pas une chose aisée, je vous invite d'ailleurs à consulter [ce tutoriel de @SpaceFox](#) [↗](#) pour en apprendre plus sur les subtilités.

VIII.4. Expressions rationnelles

Introduction

Dans ce chapitre nous allons découvrir les «expressions rationnelles» aussi connues sous le noms de *regex* (de l'anglais «*regular expressions*» parfois traduit en «expressions régulières») et comment les utiliser en Python.

VIII.4.1. Problématique

On sait demander à Python de résoudre des problèmes simples sur des chaînes de caractères comme :

- récupère-moi les N premiers caractères de la chaîne (`my_string[:N]`) ;
- teste si telle chaîne commence par tel préfixe (`my_string.startswith(prefix)`) ;
- découpe-moi cette chaîne en morceaux selon les espaces (`my_string.split(' ')`) ;
- etc.

Mais comment pourrions-nous procéder pour des problèmes plus complexes comme «est-ce que ma chaîne de caractères représente un nombre» ?¹

Une solution évidente serait de tenter une conversion `float(my_string)` et voir si elle réussit ou elle échoue.²

Mais intéressons-nous ici à une autre solution qui consisterait à analyser notre chaîne caractère par caractère afin d'identifier si oui ou non elle correspond à un nombre. La chaîne pourrait commencer par un `+` ou un `-`, suivraient une série de chiffres potentiellement suivis d'un `.` et d'une nouvelle série de chiffres.

```
1 def is_number(my_string):
2     # On traite notre chaîne comme un itérateur pour simplifier
      les traitements
3     it = iter(my_string)
4     first_char = next(it, '')
5
6     # On ignore le préfixe + ou -
7     if first_char in ('+', '-'):
8         first_char = next(it, '')
9
10    # On vérifie que la chaîne commence par un chiffre
11    if not first_char.isdigit():
```

1. Nous ne nous intéresserons ici qu'aux notations simples pour des nombres décimaux comme `42`, `+12.5` ou encore `-18000`, exit les nombres complexes ou les notations sans partie entière telles que `.3` ou à base d'exposants comme `1e10`.

2. Ce qui ne remplit pas à 100% la demande puisque l'expression reconnaît les formes `.3`, `1e10` et même `inf` qui ne nous intéressent pas ici.

```

12         return False
13
14     for char in it:
15         if char == '.':
16             # Si on tombe sur un point, on sort de la boucle pour
17             # traiter la partie décimale
18             # On vérifie cependant que la partie décimale commence
19             # par un chiffre
20             next_char = next(it, '')
21             if not next_char.isdigit():
22                 return False
23             break
24         elif not char.isdigit():
25             # Si le caractère n'est pas un chiffre, la chaîne ne
26             # peut pas représenter un nombre
27             return False
28
29     # On recommence pour la partie décimale (optionnelle)
30     for char in it:
31         if not char.isdigit():
32             return False
33
34     # On est arrivé jusqu'au bout, la chaîne représente un nombre
35     return True

```

```

1  >>> is_number('123')
2  True
3  >>> is_number('123.45')
4  True
5  >>> is_number('-123.45')
6  True
7  >>> is_number('+12000')
8  True
9  >>> is_number('abc')
10 False
11 >>> is_number('12c4')
12 False
13 >>> is_number('.5')
14 False
15 >>> is_number('10.')
16 False
17 >>> is_number('.')
18 False
19 >>> is_number('')
20 False

```

Cette solution est un peu fastidieuse mais nous verrons par la suite qu'il y a plus simple grâce aux *regex*.

VIII.4.2. Une histoire d'automates

La solution que nous venons de réaliser s'apparente à un automate fini, un modèle de calcul qui parcourt des données séquentiellement (notre chaîne caractère par caractère) afin d'identifier des motifs, le tout sans utiliser de mémoire.

L'ensemble des motifs (ou mots) qui peuvent être identifiés par un automate forme ce qu'on appelle un langage. Dans notre exemple le langage est formé des représentations de nombres de la forme `123` et `4.56` pouvant être préfixés d'un `+` ou d'un `-`.

Il est d'usage de représenter un automate sous la forme d'un graphe montrant les relations entre les états.

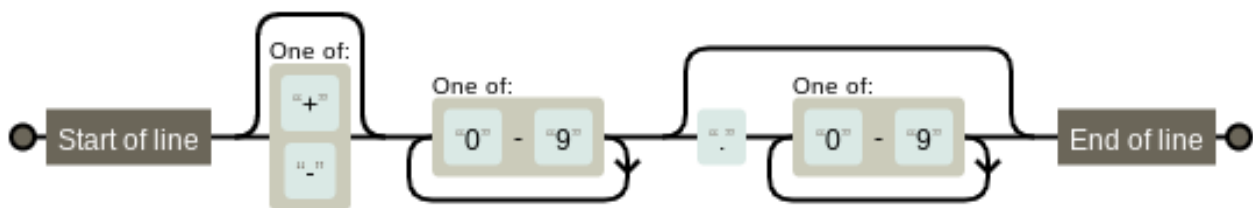


FIGURE VIII.4.1. – Automate `is_number`—image générée par [regexper](#)

Un état correspond à l'avancée dans la chaîne de caractères, en consommant les caractères qui correspondent au motif.

On part de l'état initial (à gauche) et on avance vers la droite tant qu'un chemin correspond au caractère lu dans notre chaîne (plusieurs chemins sont possibles). Si l'on atteint l'état final (à droite) alors c'est que le motif est reconnu dans la chaîne.

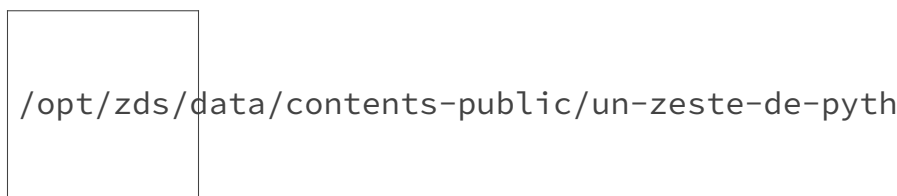


FIGURE VIII.4.2. – Animation de l'automate pour tester la chaîne `-123.45`

Il existe plusieurs types d'automates, les automates finis étant les plus simples d'entre eux. On dit d'un langage formé de mots reconnaissables par un automate fini qu'il est rationnel, d'où le terme d'expression rationnelle.

Le graphe ci-dessus représente donc une expression rationnelle pour reconnaître les chaînes représentant des nombres. Mais nous allons tout de suite voir une manière plus formelle de la décrire.

VIII.4.3. Module `re`

En effet le graphe qui précède est bien joli, mais comment l'intégrer à notre programme pour pouvoir l'utiliser ?

Ce graphe n'est qu'une représentation de l'expression rationnelle comme il en existe d'autres (notre fonction `is_number` en est une elle aussi).

Le plus souvent, on va représenter ces expressions sous la forme de chaînes de caractères, où des caractères spéciaux permettront de décrire des motifs (comme des boucles). Il existe

plusieurs standards pour cela, les plus connus étant POSIX et PCRE (*Perl-Compatible Regular Expressions*).

Le standard POSIX est celui que l'on retrouvera dans des outils tels que `grep` ou `sed`. En Python, c'est plutôt le standard PCRE qui est utilisé avec le module `re`.

Ce module regroupe les opérations permettant de travailler avec des expressions rationnelles, offrant différentes fonctions pour plusieurs usages (rechercher un motif, découper selon un motif, etc.).

VIII.4.3.1. Utilisation

On va y aller pas à pas pour construire une expression correspondant à notre besoin. Nous allons tout d'abord importer le module `re` et nous intéresser à la fonction `re.fullmatch`. C'est une fonction qui reçoit l'expression rationnelle (en premier argument) et le texte à analyser (en second) et qui renvoie un objet résultat ou `None` suivant si le texte correspond à l'expression ou non.

L'expression rationnelle peut être une chaîne de caractères toute simple (par exemple `'123'`) et la fonction va alors simplement vérifier que les caractères correspondent un à un.

```
1 >>> import re
2 >>> re.fullmatch('123', '123')
3 <re.Match object; span=(0, 3), match='123'>
4 >>> re.fullmatch('123', '124')
```

On considère dans ce cas que la *regex* se compose de motifs (1, 2, 3) qui ne peuvent chacun identifier qu'un seul caractère.



On voit dans l'objet `re.Match` renvoyé par la fonction la zone qui a été identifiée dans le texte (la valeur `span` qui indique que le motif a été identifié entre les caractères 0 et 3) et l'extrait correspondant dans le texte (`match`, le texte complet dans notre cas).

Mais l'expression peut aussi contenir des caractères particuliers pour exprimer des motifs plus évolués. Ces motifs pouvant correspondre à plusieurs caractères dans notre texte. Par exemple le caractère `.` utilisé dans une *regex* signifie «n'importe quel caractère» (comme un joker).

```
1 >>> re.fullmatch('12.', '123')
2 <re.Match object; span=(0, 3), match='123'>
3 >>> re.fullmatch('12.', '124')
4 <re.Match object; span=(0, 3), match='124'>
5 >>> re.fullmatch('12.', '134')
```

Un autre caractère particulier est le `+` qui indique que le motif qui précède peut être répété indéfiniment. La *regex* `'a+'` permet ainsi de reconnaître les suites de caractères `a` (minuscule, on note au passage que les *regex* sont sensibles à la casse par défaut).


```

1 >>> re.fullmatch('a+', 'a')
2 <re.Match object; span=(0, 1), match='a'>
3 >>> re.fullmatch('a+', 'aaaa')
4 <re.Match object; span=(0, 4), match='aaaa'>
5 >>> re.fullmatch('a+', 'aaab')
6 >>> re.fullmatch('a+', 'A')

```

i

Il est parfaitement possible de combiner nos motifs spéciaux, ainsi `.+` identifie une suite de n'importe quels caractères : `'123'`, `'aaa'`, `'abcd'`, etc.

```

1 >>> re.fullmatch('.', '123')
2 <re.Match object; span=(0, 3), match='123'>
3 >>> re.fullmatch('.', 'aaa')
4 <re.Match object; span=(0, 3), match='aaa'>
5 >>> re.fullmatch('.', 'abcd')
6 <re.Match object; span=(0, 4), match='abcd'>

```

Dans le même genre que `+` on trouve aussi `?` pour indiquer un motif optionnel. Un motif suivi d'un `?` peut donc être présent zéro ou une fois.

```

1 >>> re.fullmatch('a?b', 'ab')
2 <re.Match object; span=(0, 2), match='ab'>
3 >>> re.fullmatch('a?b', 'b')
4 <re.Match object; span=(0, 1), match='b'>
5 >>> re.fullmatch('a?b', 'a')

```

On peut utiliser des parenthèses comme en mathématiques pour gérer les priorités : `(ab)?` correspondra ainsi à la chaîne `'ab'` ou à la chaîne vide, tandis que `ab?` correspond à `'a'` ou `'ab'`.

Dans notre cas initial, on cherche à pouvoir identifier des suites de chiffres. Pour cela il va nous falloir utiliser des classes de caractères : ce sont des motifs qui peuvent correspondre à plusieurs caractères bien précis (ici des chiffres).

On définit une classe de caractère à l'aide d'une paire de crochets à l'intérieur de laquelle on fait figurer tous les caractères possibles. Par exemple `[0123456789]` correspond à n'importe quel chiffre.

Pour simplifier, il est possible d'utiliser un `-` pour définir un intervalle de caractères à l'intérieur de la classe : la syntaxe précédente devient alors équivalente à `[0-9]`.

```

1 >>> re.fullmatch('[0-9]', '5')
2 <re.Match object; span=(0, 1), match='5'>
3 >>> re.fullmatch('[0-9]+', '123')
4 <re.Match object; span=(0, 3), match='123'>

```

VIII.4.3.2. `is_number`

Nous avons maintenant toutes les clefs en main pour recoder notre fonction `is_number...` ou presque!

En effet, dans notre nombre nous voulons pouvoir identifier un caractère `.`, mais nous savons que ce caractère est un motif particulier dans une *regex* qui fait office de joker.

Comment alors faire en sorte de n'identifier que le caractère `.` et lui seul? Il nous faut pour cela l'échapper, en le faisant précéder d'un *antislash* (`\`).

```
1 >>> re.fullmatch('\.', '.')
2 <re.Match object; span=(0, 1), match='.'>
3 >>> re.fullmatch('\.', 'a')
```

Reprenons maintenant le graphe de notre automate et décomposons-le.

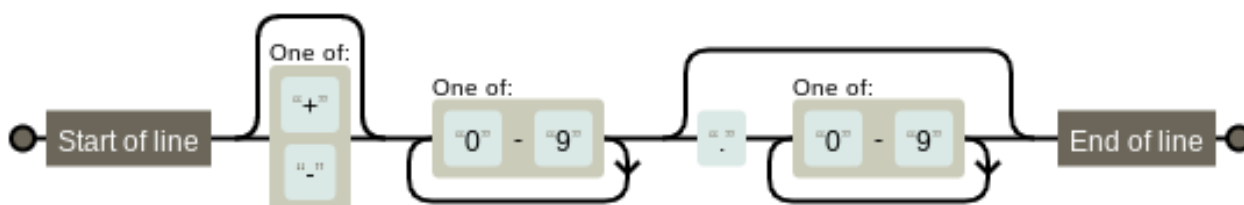


FIGURE VIII.4.3. – Automate `is_number`—image générée par [regexper](#)

Il commence par un état *Start of line*, c'est-à-dire le début de la ligne. `re.fullmatch` s'occupe déjà de rechercher un motif au début du texte donné, donc nous n'avons pas à en tenir compte ici.

L'état suivant est optionnel puisqu'il existe un chemin qui le contourne, il teste si le caractère est un `+` ou un `-`.

Cela correspond donc au motif `[+-]?` (à l'intérieur d'une classe de caractère, le `+` perd son statut de caractère spécial).

On voit que l'état suivant forme une boucle : il y a en effet un chemin qui part de la droite de l'état pour revenir à sa gauche, qui permet de le répéter indéfiniment.

Cette boucle correspond au symbole `+` que nous avons vu plus haut, qui signifie «au moins une fois».

L'état en lui-même détaille que le caractère doit être entre `0` et `9`, soit `[0-9]`. La *regex* correspondant à ce motif est donc `[0-9]+`.

Les deux états qui suivent peuvent-être court-circuités pour arriver directement à la fin, cela veut dire qu'ils forment un groupe optionnel `(...)?`.

Le premier état est un simple point (`\.`) et le second est une nouvelle suite de chiffres (`[0-9]+`).

Le groupe s'exprime donc sous la forme `(\.[0-9]+)?`.

Enfin, l'état *End of line* est lui aussi déjà géré par la fonction `fullmatch`.

En mettant tous ces extraits bout à bout, on forme la *regex* finale qui identifie nos nombres : `[+-]?[0-9]+(\.[0-9]+)?`.

```
1 >>> pattern = '[+-]?[0-9]+(\.[0-9]+)?'
2 >>> re.fullmatch(pattern, '123.456')
```

```

3 <re.Match object; span=(0, 7), match='123.456'>
4 >>> re.fullmatch(pattern, '-42')
5 <re.Match object; span=(0, 3), match='-42'>
6 >>> re.fullmatch(pattern, '100')
7 <re.Match object; span=(0, 3), match='100'>
8 >>> re.fullmatch(pattern, '0.0')
9 <re.Match object; span=(0, 3), match='0.0'>
10 >>> re.fullmatch(pattern, '.123')
11 >>> re.fullmatch(pattern, '123.')
12 >>> re.fullmatch(pattern, '.')
13 >>> re.fullmatch(pattern, 'abc')

```

La fonction `is_number` peut donc simplement être réécrite comme suit.

```

1 import re
2
3 def is_number(my_string):
4     result = re.fullmatch('[+-]?[0-9]+(\.[0-9]+)?', my_string)
5     return result is not None

```

VIII.4.3.3. Autres fonctions du module

D'autres fonctions sont aussi proposées par le module `re` pour réaliser d'autres opérations.

VIII.4.3.3.1. `re.search`

`re.search` est une fonction similaire à `re.fullmatch` à la différence qu'elle permet de trouver un motif n'importe où dans la chaîne.

```

1 >>> re.search('[0-9]+', 'abc123def')
2 <re.Match object; span=(3, 6), match='123'>

```

On remarque que les valeurs `span` et `match` du résultat correspondent à la zone où notre motif a été identifié dans le texte. Cette valeur `match` est d'ailleurs récupérable en accédant au premier élément (`[0]`) de l'objet résultat.

```

1 >>> result = re.search('[0-9]+', 'abc123def')
2 >>> result[0]
3 '123'

```



Nous verrons par la suite que ce résultat peut en effet contenir plusieurs éléments.

Sachez qu'il existe les caractères spéciaux `^` et `$` pour reproduire le comportement de `fullmatch` avec `search` : un motif débutant par `^` signifie que le motif doit être trouvé au début du texte et un motif finissant par `$` signifie que le motif doit être trouvé à la fin.

```

1 >>> re.search('[0-9]+', 'abc123')
2 >>> re.search('[0-9]+', '123abc')
3 <re.Match object; span=(0, 3), match='123'>
4 >>> re.search('[0-9]+$', '123abc')
5 >>> re.search('[0-9]+$', 'abc123')
6 <re.Match object; span=(3, 6), match='123'>

```

En combinant les deux, `re.search('^[0-9]+$', ...)` est alors équivalent à `re.fullmatch('...', ...)`.

```

1 >>> re.search('[0-9]+$', 'abc123def')
2 >>> re.search('[0-9]+$', '123')
3 <re.Match object; span=(0, 3), match='123'>

```



On note qu'il existe aussi la fonction `re.match` qui recherche un motif au début du texte. Elle est ainsi équivalente à `re.search` avec un `^` systématique.

VIII.4.3.3.2. `re.findall`

Cette fonction est un peu plus intéressante : elle permet de trouver toutes les occurrences d'un motif dans le texte. Elle renvoie la liste des extraits de texte ainsi trouvés.

```

1 >>> re.findall('[0-9]+',
2     "Nous sommes le 31 mars 2022 et il fait 10°C")
3 ['31', '2022', '10']

```

Si le motif n'est jamais trouvé, la fonction renvoie simplement une liste vide.

```

1 >>> re.findall('[0-9]+', "C'est bientôt le week-end")
2 []

```

Dans la même veine, on trouve la fonction `re.finditer` qui ne renvoie pas une liste mais un itérateur pour parcourir les résultats. Elle évite ainsi de parcourir le texte en entier dès le début et de construire une liste.

```

1 >>> for result in re.finditer('[0-9]+',
2     "Nous sommes le 31 mars 2022 et il fait 10°C"):
3     print(result)
4 <re.Match object; span=(15, 17), match='31'>
5 <re.Match object; span=(23, 27), match='2022'>
6 <re.Match object; span=(39, 41), match='10'>

```

VIII.4.3.3.3. `re.sub`

Cette fonction permet d'opérer des remplacements (ou comme son nom l'indique des substitutions) sur un texte, remplaçant chaque occurrence du motif par la valeur précisée.

Elle prend donc en arguments la *regex*, la valeur par laquelle remplacer le motif, et le texte sur lequel opérer. Et elle renvoie le texte après substitution.

```
1 >>> re.sub('[0-9]+', '?',  
2     "Nous sommes le 31 mars 2022 et il fait 10°C")  
"Nous sommes le ? mars ? et il fait ?°C"
```

Si le motif n'est pas trouvé, alors le texte est renvoyé inchangé.

```
1 >>> re.sub('[0-9]+', '?', "C'est bientôt le week-end")  
2 "C'est bientôt le week-end"
```

VIII.4.3.3.4. `re.split`

`re.split` est plus ou moins équivalente à la méthode `split` des chaînes de caractères, qui permet de découper la chaîne selon un séparateur, sauf qu'ici le séparateur est spécifié sous la forme d'une *regex*.

```
1 >>> re.split('[ ,?!:]+', 'Alors : ça décoiffe, hein ?')  
2 ['Alors', 'ça', 'décoiffe', 'hein', '']
```

i

On constate qu'une chaîne vide est renvoyée dans le résultat si le texte termine par un séparateur. Mais on peut facilement la filtrer si elle ne nous intéresse pas.

```
1 >>> [s for s in re.split('[ ,?!:]+',  
2     'Alors : ça décoiffe, hein ?') if s]  
['Alors', 'ça', 'décoiffe', 'hein']
```

VIII.4.3.3.5. `re.compile`

On notera enfin la présence de la fonction `re.compile` qui permet de créer un objet *regex*. Cette fonction reçoit l'expression rationnelle sous forme d'une chaîne et renvoie un objet avec des méthodes `fullmatch`, `search`, `finditer`, `split`, etc.

Cela peut être plus pratique si l'on est amené à réutiliser plusieurs fois une même expression.

```
1 >>> pattern = re.compile('[0-9]+')  
2 >>> pattern.findall('3 + 5 = 8')  
3 ['3', '5', '8']
```

```

4 >>> pattern.sub('?', '3 + 5 = 8')
5 '? + ? = ?'

```

VIII.4.4. Syntaxe des regex

Maintenant que nous connaissons les fonctions du module, voyons voir quelques autres éléments de syntaxe des *regex*.

VIII.4.4.1. Chaînes brutes (*raw strings*)

Il est d'usage, pour représenter des expressions rationnelles, de ne pas utiliser des chaînes de caractères telles quelles mais d'utiliser ce qu'on appelle des chaînes brutes (ou *raw strings*). On les reconnaît au caractère `r` qui les préfixe.

```

1 >>> r'abc'
2 'abc'

```

Celles-ci ne forment pas un type particulier, on voit d'ailleurs que l'objet évalué est une chaîne de caractère tout à fait normale. Non la différence se trouve au niveau de l'analyse de l'entrée par l'interpréteur, la façon dont il interprète les caractères écrits pour former l'objet `str`.

On le sait, les chaînes de caractères permettent d'utiliser des séquences d'échappement telles que `\t` ou `\n` pour représenter des caractères spéciaux.

```

1 >>> print('abc\tdef\nghi')
2 abc      def
3 ghi

```

Ce comportement est rendu possible par l'interpréteur qui quand il lit la séquence de caractères `\t` dans le code la transforme en caractère «tabulation».

Mais il ne le fait pas pour les chaînes brutes, qui conservent alors toutes les séquences d'échappement sans les interpréter comme des caractères spéciaux.

```

1 >>> print(r'abc\tdef\nghi')
2 abc\tdef\nghi

```

Pour les *regex*, on préfère ainsi utiliser des chaînes brutes pour ne pas générer de conflits avec des motifs qui pourraient être interprétés comme des séquences d'échappement.

```

1 >>> re.fullmatch(r'[0-9]+', '1234')
2 <re.Match object; span=(0, 4), match='1234'>

```

VIII.4.4.2. Syntaxe des motifs

On a déjà vu de nombreux motifs dans le début du chapitre, mais laissez-moi ici vous les présenter de façon plus détaillée.

VIII.4.4.2.1. Échappement (\)

L'antislash utilisé devant un caractère spécial du motif permet de lui faire faire son aspect spécial et de l'utiliser comme un caractère normal. `\+` identifie le caractère `+`.

```
1 >>> re.match(r'\.\.\+$', '.*+$')
2 <re.Match object; span=(0, 3), match='.*+$'>
3 >>> re.match(r'\.\.\+$', 'toto')
4 >>> re.match(r'.+$', 'toto')
5 <re.Match object; span=(0, 4), match='toto'>
```

VIII.4.4.2.2. Joker (.)

`.` est le caractère joker, il correspond à n'importe quel caractère du texte (hors retours à la ligne). Il correspond toujours à un et un seul caractère.

```
1 >>> re.match(r'.', 'a')
2 <re.Match object; span=(0, 1), match='a'>
3 >>> re.match(r'.', '@')
4 <re.Match object; span=(0, 1), match='@'>
5 >>> re.match(r'.', '')
6 >>> re.match(r'.', 'ab')
7 <re.Match object; span=(0, 1), match='a'>
```

Par défaut, le caractère de retour à la ligne (`\n`) n'est pas reconnu par ce motif mais on verra avec l'option `DOTALL` comment y remédier.

```
1 >>> re.match(r'.', '\n')
```

VIII.4.4.2.3. Classes de caractères ([...])

Les crochets identifient les classes de caractères, une classe pouvant alors correspondre à n'importe lequel des caractères qu'elle contient. `[abc]` pourra correspondre aux caractères `a`, `b` ou `c` (toujours un et un seul).

Il est possible de préciser dans cette classe des intervalles de chiffres ou de lettres à l'aide d'un tiret (`-`). `[0-9]` identifie ainsi un chiffre et `[0-0A-Za-z]` un caractère alphanumérique.

Pour contenir le caractère `-` en lui-même, il est possible de l'échapper (le précéder d'un `\`) ou le placer au tout début ou à la fin de la classe : `[0-91-Za-z_-]` identifie un caractère alphanumérique, un caractère de soulignement (`_`) ou un tiret (`-`).

Un `^` placé en début de classe fait office de négation, ainsi la classe `[^0-9]` reconnaît les caractères qui ne sont pas des chiffres.

Les autres symboles que nous avons pu voir perdent leur signification spéciale à l'intérieur d'une classe de caractères. Seul le caractère `]` a besoin d'être échappé pour éviter de fermer la classe prématurément.

VIII.4.4.2.4. Quantificateurs (`?`, `+`, `*`, `{...}`)

Les quantificateurs sont différents symboles qui s'appliquent au motif qui précède afin d'en préciser la quantité attendue.

- `?` rend le motif optionnel. Il s'agit alors d'un quantificateur 0 ou 1 fois.
- `+` permet de répéter le motif. Il s'agit alors d'un quantificateur 1 fois ou plus.
- `*` est un quantificateur 0 ou plus, il combine alors `?` et `+`.

Les accolades (`{...}`) permettent d'appliquer un quantificateur personnalisé au motif qui précède. On précise à l'intérieur de ces accolades le nombre de répétitions voulues, ou l'intervalle de répétitions acceptées (sous forme de deux nombres séparés d'une virgule).

Par exemple `x{3}` identifie la chaîne `xxx` et `x{2,4}` correspond aux chaînes `xx`, `xxx` et `xxxx`.

Il est possible d'omettre l'une ou l'autre des bornes de l'intervalle. `{,n}` sera alors équivalent à `{0,n}` et `{n,}` signifiera un motif répété au moins `n` fois.

VIII.4.4.2.5. Groupes (`(...)`)

Les parenthèses permettent de prioriser une sous-expression mais aussi de former un groupe de capture. Lors d'un appel valide à `re.fullmatch` par exemple, l'objet `re.Match` renvoyé permet d'accéder aux différentes valeurs des groupes capturés.

Chaque groupe est identifié par un nombre correspondant à sa position dans l'expression, et le groupe 0 correspond à la chaîne entière.

```
1 >>> match = re.fullmatch('([0-9]+)\+([0-9]+)=([0-9]+)', '13+25=38')
2 >>> match[0]
3 '13+25=38'
4 >>> match[1]
5 '13'
6 >>> match[2]
7 '25'
8 >>> match[3]
9 '38'
```

L'objet `re.Match` possède aussi une méthode `groups` pour renvoyer tous les groupes capturés dans le texte.

```
1 >>> match.groups()
2 ('13', '25', '38')
```



Pour bénéficier de la priorisation des parenthèses sans créer de groupe de capture, il est possible d'utiliser un `?:` à l'intérieur des parenthèses (`(?:...)`), Python comprendra alors que ces parenthèses ne correspondent pas à un groupe.



```

1 >>> re.fullmatch('(ab)+', 'ababab')
2 <re.Match object; span=(0, 6), match='ababab'>
3 >>> _.groups()
4 ('ab',)
5 >>> re.fullmatch('(?:ab)+', 'ababab')
6 <re.Match object; span=(0, 6), match='ababab'>
7 >>> _.groups()
8 ()

```

VIII.4.4.2.6. Unions (|)

Les quantificateurs nous permettent de représenter un choix entre plusieurs alternatives suivant le nombre de fois qu'un motif est répété. Cette notion de choix est au cœur des automates finis puisqu'ils représentent les différents chemins qui partent d'un même nœud.

Pour représenter un choix simple, on utilise l'opérateur d'union (|), celui-ci offrant deux possibilités pour évaluer la chaîne : soit le motif de gauche, soit celui de droite. Ainsi l'expression `ab|cd` correspond aux deux chaînes `'ab'` et `'cd'`.

```

1 >>> re.fullmatch(r'ab|cd', 'ab')
2 <re.Match object; span=(0, 2), match='ab'>
3 >>> re.fullmatch(r'ab|cd', 'cd')
4 <re.Match object; span=(0, 2), match='cd'>
5 >>> re.fullmatch(r'ab|cd', 'abcd')

```

L'opérateur d'union a une priorité plus faible que l'ensemble des autres opérateurs, à l'exception des parenthèses qui permettent donc de prioriser une union.

L'expression `a(b|c)d` correspond alors aux chaînes `'abd'` et `'acd'`.

```

1 >>> re.fullmatch(r'a(b|c)d', 'abd')
2 <re.Match object; span=(0, 3), match='abd'>
3 >>> re.fullmatch(r'a(b|c)d', 'acd')
4 <re.Match object; span=(0, 3), match='acd'>
5 >>> re.fullmatch(r'a(b|c)d', 'ab')

```

Un quantificateur peut évidemment être appliqué à une union, deux choix possibles seront alors à opérer à chaque répétition du motif. `(ab|ba)+` représente une chaîne comprenant une suite de mots `ab` ou `ba`.

```

1 >>> re.fullmatch('(ab|ba)+', 'ababab')
2 <re.Match object; span=(0, 6), match='ababab'>
3 >>> re.fullmatch('(ab|ba)+', 'baba')
4 <re.Match object; span=(0, 4), match='baba'>
5 >>> re.fullmatch('(ab|ba)+', 'abba')

```

```

6 <re.Match object; span=(0, 4), match='abba'>
7 >>> re.fullmatch('(ab|ba)+', 'abbb')
```

Enfin, il est possible d'utiliser plusieurs `|` successifs pour représenter un choix entre plus de deux motifs. `ab|bc|cd` identifie le motif `ab`, `bc` ou `cd`.

```

1 <re.Match object; span=(0, 2), match='ab'>
2 >>> re.fullmatch('ab|bc|cd', 'bc')
3 <re.Match object; span=(0, 2), match='bc'>
4 >>> re.fullmatch('ab|bc|cd', 'cd')
5 <re.Match object; span=(0, 2), match='cd'>
6 >>> re.fullmatch('ab|bc|cd', 'ac')
```



On note que les unions permettent de représenter différemment des motifs que l'on connaissait déjà. Par exemple `X|XY` est équivalent à `XY?` et `a|b|c` est équivalent à `[abc]`.

VIII.4.4.2.7. Marqueurs d'extrémités (^ et \$)

Les caractères `^` et `$` permettent respectivement d'identifier le début et la fin du texte (ou de la ligne suivant le mode, voir les options plus bas).

Ces marqueurs n'ont pas d'intérêt avec `re.fullmatch` qui les ajoute implicitement mais s'avèrent utiles pour les autres fonctions du module. Un motif débutant par `^` indique qu'il doit se trouver au début du texte, tandis qu'un motif se terminant par `$` indique qu'il doit se trouver à la fin du texte.

```

1 >>> re.search(r'^a', 'bac')
2 >>> re.search(r'^a', 'abc')
3 <re.Match object; span=(0, 1), match='a'>
4 >>> re.search(r'a$', 'bac')
5 >>> re.search(r'a$', 'bca')
6 <re.Match object; span=(2, 3), match='a'>
```

Ces marqueurs sont moins prioritaires que l'union, il est donc parfaitement possible par exemple de représenter l'ensemble des chaînes qui commencent par «zeste» ou terminent par «savoir» avec `^zeste|savoir$`.

```

1 >>> re.search(r'^zeste|savoir$', 'zeste de savoir')
2 <re.Match object; span=(0, 5), match='zeste'>
3 >>> re.search(r'^zeste|savoir$', 'concentré de savoir')
4 <re.Match object; span=(13, 19), match='savoir'>
5 >>> re.search(r'^zeste|savoir$', 'zeste de citron')
6 <re.Match object; span=(0, 5), match='zeste'>
7 >>> re.search(r'^zeste|savoir$', 'concentré de citron')
```



On remarque que lorsque les deux motifs d'une union correspondent au texte, c'est celui de gauche qui l'emporte («zeste de savoir» *matche* sur `^zeste` avant `savoir$`).

VIII.4.4.2.8. Séquences spéciales

On trouve aussi quelques séquences d'échappement particulières pour représenter facilement certaines classes de caractères.

Ainsi, `\d` identifie un chiffre (à la manière de `[0-9]` mais en plus large car identifie tous les caractères reconnus comme tels par le standard Unicode).

```
1 >>> re.fullmatch(r'\d+', '123')
2 <re.Match object; span=(0, 3), match='123'>
3 >>> re.fullmatch(r'\d+', 'abc')
4 >>> re.fullmatch(r'\d+', '')
5 <re.Match object; span=(0, 3), match=''>
```

À l'inverse, `\D` identifie ce qui n'est pas un chiffre.

```
1 >>> re.fullmatch(r'\D+', '123')
2 >>> re.fullmatch(r'\D+', 'abc')
3 <re.Match object; span=(0, 3), match='abc'>
```

La séquence `\w` correspond aux caractères alphanumériques unicodes (chiffres, lettres et caractères de soulignement comme `_`). Là encore, `\W` (notez la majuscule) identifie le motif inverse, soit les caractères non alphanumériques.

```
1 >>> re.fullmatch(r'\w+', 'Ab_12')
2 <re.Match object; span=(0, 5), match='Ab_12'>
3 >>> re.fullmatch(r'\w+', 'Àâ_')
4 <re.Match object; span=(0, 5), match='Àâ_'>
5 >>> re.fullmatch(r'\w+', '!.?')
6 >>> re.fullmatch(r'\W+', '!.?')
7 <re.Match object; span=(0, 2), match='!.?'>
```

La séquence `\s` identifie un caractère d'espacement, et `\S` un autre caractère.

```
1 >>> re.fullmatch(r'\s', ' ')
2 <re.Match object; span=(0, 1), match=' '>
3 >>> re.fullmatch(r'\s', '\n')
4 <re.Match object; span=(0, 1), match='\n'>
5 >>> re.fullmatch(r'\s', '\t')
6 <re.Match object; span=(0, 1), match='\t'>
7 >>> re.fullmatch(r'\s', 'x')
8 >>> re.fullmatch(r'\S', 'x')
```

```
9 <re.Match object; span=(0, 1), match='x'>
```

D'autres motifs et séquences d'échappement ne sont pas abordés ici et je vous invite à les retrouver dans [la documentation du mode `re`](#) [↗](#).

VIII.4.4.3. Options

Les fonctions de recherche du module `re` acceptent un argument `flags` qui permet de préciser des options sur la recherche, que je vais vous décrire ici.

VIII.4.4.3.1. `re.IGNORECASE` (ou `re.I`)

Cette option permet simplement d'ignorer la casse des caractères de la chaîne à analyser, ainsi le motif ne fera pas de différence entre caractères en minuscules ou en capitales.

```
1 >>> re.match('[a-z]+', 'ToTo', re.IGNORECASE)
2 <re.Match object; span=(0, 4), match='ToTo'>
3 >>> re.match('[a-z]+', 'ToTo')
```

VIII.4.4.3.2. `re.ASCII` (`re.A`)

Par défaut les *regex* en Python expriment des motifs unicode, c'est-à-dire qu'elles gèrent les caractères accentués et spéciaux.

Comme on l'a vu, le motif `\w` permet par exemple de reconnaître des chiffres et des lettres quelle que soit leur forme (différents alphabets, différentes diacritiques).

Mais il est possible de restreindre ces motifs à la seule table des caractères ASCII (cf [le tableau dans le chapitre dédié aux *bytes*](#) [↗](#)) avec l'option `ASCII` et ainsi n'accepter par exemple que les lettres de l'alphabet latin.

```
1 >>> re.match('\w+', 'été', re.ASCII)
2 >>> re.match('\w+', 'ete', re.ASCII)
3 <re.Match object; span=(0, 3), match='ete'>
4 >>> re.match('\w+', 'été')
5 <re.Match object; span=(0, 3), match='été'>
```

VIII.4.4.3.3. `re.DOTALL` (`re.S`)

On a vu précédemment que le motif joker (`*`) ne reconnaissait pas le caractère de retour à la ligne dans le mode par défaut. Il est possible de changer ce comportement à l'aide de l'option `DOTALL`.

```
1 >>> re.match(r'.', '\n', re.DOTALL)
2 <re.Match object; span=(0, 1), match='\n'>
3 >>> re.match(r'.', '\n')
```

VIII.4.4.3.4. `re.MULTILINE` (`re.M`)

Enfin, l'option `MULTILINE` est une option qui permet de gérer différemment les textes sur plusieurs lignes.

Par défaut, une chaîne de caractères contenant des retours à la ligne (`\n`) est gérée comme les autres chaînes, sans traitement particulier pour les sauts de ligne.

Cette option permet de différencier les lignes les unes des autres et d'avoir un traitement différencié. Ainsi les marqueurs `^` et `$` n'identifieront plus seulement le début et la fin du texte mais aussi le début et la fin de chaque ligne.

```
1 >>> re.findall(r'^.+$', 'abc\ndef\nghi', re.MULTILINE)
2 ['abc', 'def', 'ghi']
3 >>> re.findall(r'^.+$', 'abc\ndef\nghi')
4 []
```



Le traitement n'est pas le même qu'avec l'option `DOTALL` qui elle ne reconnaît simplement pas les sauts de ligne comme des caractères spéciaux.

```
1 >>> re.findall(r'^.+$', 'abc\ndef\nghi', re.DOTALL)
2 ['abc\ndef\nghi']
```

VIII.4.4.3.5. Composition d'options

Les options ne sont pas exclusives et peuvent être composées les unes avec les autres.

On utilise pour cela la notation d'union afin d'assembler différentes options entre elles.

```
1 >>> re.findall(r'^[a-z]\w+', 'abc\ndef\nghi', re.ASCII |
2 re.MULTILINE | re.IGNORECASE)
3 ['abc', 'DEF', 'gh']
```

Ainsi le code qui précède permet de faire une recherche ascii multiligne ignorant la casse.

On pourra bien sûr enregistrer ces options dans une variable si on est amenés à les réutiliser.

```
1 >>> flags = re.ASCII | re.IGNORECASE
2 >>> re.fullmatch(r'zds_\w+', 'zds_foo', flags)
3 <re.Match object; span=(0, 7), match='zds_foo'>
4 >>> re.fullmatch(r'zds_\w+', 'ZDS_BAR', flags)
5 <re.Match object; span=(0, 7), match='ZDS_BAR'>
6 >>> re.fullmatch(r'zds_\w+', 'zds_été', flags)
```



L'ordre des opérandes autour des `|` n'a pas d'importance, puisqu'il s'agit d'une union de tous les éléments.



On remarque d'ailleurs que l'ordre n'est pas conservé dans le résultat de l'union.

```
1 >>> re.MULTILINE | re.ASCII
2 re.ASCII | re.MULTILINE
```

VIII.4.5. Limitations

De par leur construction (automates finis) les expressions rationnelles sont normalement assez limitées en raison de l'absence de mémorisation : elles ne permettent de reconnaître que des langages rationnels.

Il s'agit du type de langage le plus simple dans la [hiérarchie de Chomsky](#) [↗](#), on ne peut pas les utiliser pour décrire des structures récursives par exemple.

Mais le moteur de *regex* de Python permet d'aller au-delà de certaines limitations (au prix de l'efficacité et de la lisibilité) en fournissant des fonctionnalités supplémentaires :

- Le *look-ahead* qui permet de regarder ce qui suit une expression.

```
1 >>> # trouve toutes les lettres suivies d'un "b"
2 >>> re.findall(r'\w(?:=b)', 'ab cd eb')
3 ['a', 'e']
4 >>> # ou celles qui ne sont pas suivies d'une espace
5 >>> re.findall(r'\w(?:! )', 'ab cd eb')
6 ['a', 'c', 'e', 'b']
```

- Le *look-behind* pour regarder ce qui précède.

```
1 >>> # trouve toutes les lettres précédées d'un "a"
2 >>> re.findall(r'(?<=a)\w', 'ab de ac')
3 ['b', 'c']
4 >>> # ou celles qui ne sont pas précédées d'une espace
5 >>> re.findall(r'(?<! )\w', 'ab de ac')
6 ['a', 'b', 'e', 'c']
```

- Les *back-references* pour référencer une expression déjà capturée.

```
1 >>> # trouve les motifs doublés
2 >>> re.findall(r'(\w+)(\1)', 'toto tutu tati')
3 [('to', 'to'), ('tu', 'tu')]
4 >>> reconnaît N occurrences de "a" suivies
   d'un "b" et de N nouvelles occurrences de "a"
5 >>> re.fullmatch(r'(a+)b(\1)', 'aba')
6 <re.Match object; span=(0, 3), match='aba'>
7 >>> re.fullmatch(r'(a+)b(\1)', 'aaabaaa')
8 <re.Match object; span=(0, 7), match='aaabaaa'>
9 >>> re.fullmatch(r'(a+)b(\1)', 'abaaa')
```

VIII. La bibliothèque standard

Cependant, même avec ces fonctionnalités supplémentaires certaines choses restent impossibles. Par exemple on ne peut pas écrire de motif pour reconnaître N occurrences de «a» suivies de N occurrences de «b».

De même qu'une expression arithmétique (`3 * (1 + 2 * 5)`), par sa nature récursive, ne peut pas être reconnue par une *regex*, même étendue.

On notera enfin que les fonctionnalités étendues présentées ici ne sont pas standards et ne seront pas reconnues par les moteurs de *regex* «purs»¹, je vous recommande donc de les éviter autant que possible (ainsi que pour des questions de lisibilité et de performances) et de préférer des algorithmes plus classiques pour résoudre vos problèmes complexes.

Conclusion

Pour des informations plus complètes sur les *regex* en Python, je vous renvoie bien sûr à [la documentation du module `re`](#) [↗](#).

1. Par exemple la bibliothèque [`re2`](#) [↗](#) qui propose une implémentation optimale d'un moteur d'expressions rationnelles (à l'aide d'automates finis justement) ne comprend pas ces extensions (et c'est ce qui lui permet d'être optimale).

VIII.5. TP : Monstre sauvage

Introduction

Comme promis, on va reprendre notre jeu pour le transformer en jeu solo. Maintenant on sélectionnera un monstre et l'ordinateur en choisira un autre. À chaque tour, il sélectionnera aussi quelle attaque il souhaite nous infliger.

VIII.5.1. L'aléatoire à la rescousse !

On va donc intégrer quelques doses d'aléatoire dans notre jeu, à différents niveaux :

- Pour le choix de monstre, l'ordinateur réalisera un choix aléatoire, de même pour son nombre de PV.
- Pour connaître l'ordre d'attaque entre les deux monstres, on pourra faire un tirage aléatoire (savoir qui commence).
- À chaque tour, l'ordinateur sélectionnera une attaque aléatoire pour son monstre. Ce choix pourra être pondéré selon les dégâts infligés par chaque attaque.

Pour plus de généricité, on aimerait ne pas avoir à gérer l'ordinateur comme un cas spécifique, et donc ne pas faire de différence de traitement entre nos deux joueurs.

Pour cela, je vous propose de modifier la structure des joueurs (le dictionnaire tel que renvoyé par la fonction `get_player`) pour y ajouter une fonction (un *callback*) associée à la clé `'chose_attack_func'`, qui pourra être appelée depuis la boucle de jeu pour demander au joueur de sélectionner une attaque.

Dans le cas d'un joueur humain, cette fonction fera appel à `input`, et dans le cas de l'ordinateur elle opérera une sélection aléatoire. Mais la boucle de jeu n'en saura rien, ce sera totalement abstrait pour elle.

```
1 attack = player['chose_attack_func'](player)
2 apply_attack(player, opponent)
```

En bonus, on pourrait ajouter un choix pour permettre au 2ème joueur d'être un humain ou un robot, voire que les deux joueurs soient des ordinateurs pour les observer combattre. 🍊

VIII.5.1.1. Solution

Je vous propose la solution suivante, n'hésitez pas à regarder plus en détails le mécanisme de *callback*. J'ai aussi utilisé une interface commune entre les modules `players` et `ia`, avec une fonction `get_player` prenant un identifiant en argument et renvoyant un dictionnaire décrivant le joueur.

Pour la sélection du nombre de PV par l'ordinateur, j'ai utilisé une distribution normale, mais tout autre tirage serait correct.

Enfin, pour alléger le code, j'ai supprimé ce qui était relatif à la sauvegarde du jeu car ça n'est plus utile ici.

👁 Contenu masqué n°17

VIII.5.2. Animations

Un tout petit exercice avant de finir.

L'ordinateur est un peu trop rapide à jouer, on a à peine le temps de voir ce qui se passe. On serait alors tenté d'ajouter un simple `time.sleep(1)` pour ralentir l'exécution, mais on se demanderait alors ce qui se passe.

Une autre idée serait d'ajouter des animations pendant les choix de l'ordinateur, afin de voir qu'il se passe quelque chose sans que ça ne se passe trop vite.

Et pour cela, on va simplement utiliser les fonctions `print` et `time.sleep`.

Par exemple comment représenter une barre de progression en animation ? On peut afficher un caractère, attendre, afficher un autre caractère, etc.

Pour cela, on va appeler `print` avec l'argument `end=''` (pour ne pas afficher de retour à la ligne) dans une boucle. En sortie de boucle, on s'occupera de revenir à la ligne pour finaliser la barre.

```
1 import time
2
3 for _ in range(10):
4     print('-', end='')
5     time.sleep(0.1)
6
7 print()
```

Si vous exécutez ce code, vous verrez probablement la barre complète s'afficher d'un seul coup au bout d'une seconde, sans aucune animation.

Cela est dû au mécanisme de *flush* (mémoire tampon) dont je vous avais parlé : en l'absence de retour à la ligne, `print` a simplement placé le texte en mémoire tampon mais n'a rien écrit réellement sur le terminal. On corrige ça en ajoutant l'argument `flush=True` à l'appel.

```
1 import time
2
3 for _ in range(10):
4     print('-', end='', flush=True)
5     time.sleep(0.1)
6
7 print()
```

i

Pour aller plus loin, on peut aussi utiliser le caractère spécial `\b` qui permet de revenir en arrière sur la ligne et donc d'effacer le dernier caractère imprimé.

VIII.5.2.1. Solution

Rien de bien méchant, je présente ici le fichier `tp/ia.py` uniquement qui est le seul à changer.

👁 Contenu masqué n°18

Si les animations dans le terminal vous intéressent et que vous souhaitez aller plus loin, je vous conseille de regarder du côté [du module `curses`](#) de Python, qui permet de dessiner dans le terminal plus simplement qu'avec `print` et `'\b'`.

La bibliothèque tierce [`prompt_toolkit`](#) peut aussi être un bon point d'entrée.

Contenu masqué

Contenu masqué n°17

1

Listing 77 – `tp/__init__.py`

```
1 from . import game
2
3
4 if __name__ == '__main__':
5     game.main()
```

Listing 78 – `tp/__main__.py`

```
1 monsters = {
2     'pythachu': {
3         'name': 'Pythachu',
4         'attacks': ['tonnerre', 'charge'],
5     },
6     'pythard': {
7         'name': 'Pythard',
8         'attacks': ['jet-de-flotte', 'charge'],
9     },
10    'ponytha': {
11        'name': 'Ponytha',
12        'attacks': ['brûlure', 'charge'],
13    },
14 }
15
16 attacks = {
17     'charge': {'damages': 20},
18     'tonnerre': {'damages': 50},
```

```

19     'jet-de-flotte': {'damages': 40},
20     'brûlure': {'damages': 40},
21 }

```

Listing 79 – tp/definitions.py

```

1  import random
2
3  from .definitions import attacks
4  from .prompt import get_choice_input
5  from .players import get_player as get_real_player
6  from .ia import get_player as get_ia_player
7
8
9  def apply_attack(attack, opponent):
10     opponent['pv'] -= attack['damages']
11     if opponent['pv'] < 0:
12         opponent['pv'] = 0
13
14
15  def game_turn(player, opponent):
16     # Si le joueur est KO, il n'attaque pas
17     if player['pv'] <= 0:
18         return
19
20     attack = player['chose_attack_func'](player)
21     apply_attack(attack, opponent)
22
23     print(
24         player['monster']['name'],
25         'attaque',
26         opponent['monster']['name'],
27         'qui perd',
28         attack['damages'],
29         'PV, il lui en reste',
30         opponent['pv'],
31     )
32
33
34  def get_winner(player1, player2):
35     if player1['pv'] > player2['pv']:
36         return player1
37     else:
38         return player2
39
40
41  def main():
42     players = [get_real_player(1), get_ia_player(2)]

```

```

43     random.shuffle(players)
44     player1, player2 = players
45
46     print()
47     print(player1['monster']['name'], 'affronte',
48           player2['monster']['name'])
49     print()
50     while player1['pv'] > 0 and player2['pv'] > 0:
51         game_turn(player1, player2)
52         game_turn(player2, player1)
53
54     winner = get_winner(player1, player2)
55     print('Le joueur', winner['id'], 'remporte le combat avec',
56           winner['monster']['name'])

```

Listing 80 – tp/game.py

```

1  import random
2
3  from .definitions import attacks, monsters
4
5
6  def chose_monster():
7      values = list(monsters.values())
8      monster = random.choice(values)
9      return monster
10
11
12  def chose_attack(player):
13      monster = player['monster']
14      weights = [attacks[name]['damages'] for name in
15                monster['attacks']]
16      att_name = random.choices(monster['attacks'],
17                                weights=weights)[0]
18      print(f"Le joueur {player['id']} utilise {att_name}")
19      return attacks[att_name]
20
21
22  def get_player(player_id):
23      monster = chose_monster()
24      pv = int(random.normalvariate(100, 10))
25      print(f"
26            Le joueur {player_id} choisit {monster['name']} ({pv} PV)"
27            )
28
29      return {
30          'id': player_id,

```

```
27     'monster': monster,
28     'pv': pv,
29     'chose_attack_func': chose_attack,
30 }
```

Listing 81 – tp/ia.py

```
1  from .definitions import attacks, monsters
2  from .prompt import get_choice_input
3
4
5  def chose_attack(player):
6      print('Joueur', player['id'], 'quelle attaque utilisez-vous ?')
7      for name in player['monster']['attacks']:
8          print('-', name.capitalize(), -attacks[name]['damages'],
9                'PV')
10
11     return get_choice_input(attacks, 'Attaque invalide')
12
13  def get_player(player_id):
14      print('Monstres disponibles :')
15      for monster in monsters.values():
16          print('-', monster['name'])
17
18      print('Joueur', player_id, 'quel monstre choisissez-vous ?')
19      monster = get_choice_input(monsters, 'Monstre invalide')
20      pv = int(input('Quel est son nombre de PV ? '))
21      return {
22          'id': player_id,
23          'monster': monster,
24          'pv': pv,
25          'chose_attack_func': chose_attack,
26      }
```

Listing 82 – tp/players.py

```
1  def get_choice_input(choices, error_message):
2      entry = input('> ').lower()
3      while entry not in choices:
4          print(error_message)
5          entry = input('> ').lower()
6      return choices[entry]
```

Listing 83 – tp/prompt.py

[Retourner au texte.](#)

Contenu masqué n°18

```

1 import random
2 import time
3
4 from .definitions import attacks, monsters
5
6
7 def wait(steps, step_duration=0.1):
8     print('[', end='', flush=True)
9     for _ in range(steps):
10         print('>', end='', flush=True)
11         time.sleep(step_duration)
12         print('\b#', end='', flush=True)
13     print(']')
14
15
16 def chose_monster():
17     values = list(monsters.values())
18     monster = random.choice(values)
19     wait(10)
20     return monster
21
22
23 def chose_attack(player):
24     monster = player['monster']
25     weights = [attacks[name]['damages'] for name in
26               monster['attacks']]
27     att_name = random.choices(monster['attacks'],
28                               weights=weights)[0]
29
30     wait(10)
31     print(f"Le joueur {player['id']} utilise {att_name}")
32     return attacks[att_name]
33
34 def get_player(player_id):
35     monster = chose_monster()
36     pv = int(random.normalvariate(100, 10))
37     print(f"
38           "Le joueur {player_id} choisit {monster['name']} ({pv} PV)"
39           )
40
41     return {
42         'id': player_id,
43         'monster': monster,
44         'pv': pv,
45         'chose_attack_func': chose_attack,
46     }

```

Listing 84 – `tp/ia.py`

[Retourner au texte.](#)

VIII.6. Installer des modules complémentaires

Introduction

La bibliothèque standard de Python a beau être très complète (voici par exemple [la liste de tous ses modules](#) [↗](#)), elle ne couvre pas tous les usages possibles.

Il est ainsi parfois nécessaire, quand on développe un programme, de réutiliser le code de quelqu'un d'autre, de faire appel à une bibliothèque logicielle externe.

Généralement cette bibliothèque prendra la forme d'un paquet Python, que l'on pourra installer pour le rendre disponible comme tout autre module. Il nous suffira alors de l'importer depuis notre code pour pouvoir l'utiliser.

Voyons donc maintenant comment installer de tels modules complémentaires.

VIII.6.1. Installation

Il existe plusieurs manières d'installer des modules complémentaires en Python.

D'abord, ils peuvent être installés au niveau du système d'exploitation, notamment si celui-ci met à disposition un gestionnaire de paquets. Sous Ubuntu on trouve ainsi un paquet `python3-numpy` dans `apt` pour installer la bibliothèque Python `numpy` par exemple.

Sous Windows, on trouvera parfois des fichiers `.exe` permettant d'installer des modules particuliers.

Ces installations se font au niveau du système, les bibliothèques deviennent alors disponibles depuis n'importe où sur l'ordinateur.

On trouve aussi des suites logicielles, telles que Anaconda (voir [ce tutoriel sur Zeste de Savoir](#) [↗](#)), qui viennent directement avec un ensemble de paquets tiers pour un usage particulier (ici des bibliothèques dédiées au calcul scientifique) et ainsi en simplifier l'installation.

Mais ces solutions sont assez dépendantes du système utilisé, et il devient difficile de simplement dire «mon code a besoin du module *potjevleesch* pour fonctionner» si son installation est différente sur chaque machine.

Heureusement, Python fournit un outil pour simplifier et unifier tout cela : `pip` !

VIII.6.2. Pip, le gestionnaire de paquets Python

Pip est un gestionnaire de paquets spécialement dédié à l'installation de modules complémentaires pour Python. Il est normalement inclus dans toute installation récente de Python. Vous pouvez vous en assurer en essayant d'exécuter la commande `python -m pip` (ou `python3 -m pip`, voire `py -m pip` sur Windows) depuis un terminal (pas depuis une console Python).



Si toutefois ce n'était pas le cas, regardez si un paquet `python-pip` ou `python3-pip` existe dans le gestionnaire de paquets de votre système que vous pourriez installer. Sinon, vous pouvez exécuter la commande `python3 -m ensurepip --default-pip` (`py -m ensurepip --default-pip` sous Windows) pour demander à Python d'installer le nécessaire.

Pip peut donc être invoqué via la commande *shell* `python -m pip` (`py -m pip` sous Windows), ou par le simple raccourci `pip`.

L'outil comprend plusieurs commandes, notamment la commande `install` pour installer un paquet, suivie du nom du paquet à installer. Ce nom sera généralement inscrit sur les sites officiels des bibliothèques que vous souhaitez installer.

```

1 % pip install Pillow
2 Collecting Pillow
3   Downloading Pillow-9.0.1-cp310-cp310-manylinux_2_17_x86_64.manyl
      inux2014_x86_64.whl (4.3
      MB)
4   |████████████████████████████████████████| 4.3 MB 1.7 MB/s
5 Installing collected packages: Pillow
6 Successfully installed Pillow-9.0.1

```

Le paquet sera alors installé par défaut pour l'utilisateur courant (ou pour tout le système si la commande est exécutée avec les droits d'administration).



Attention aux paquets frauduleux. Il peut arriver que certains paquets imitent le nom de paquets connus pour diffuser du code malicieux. Assurez-vous donc de toujours utiliser le nom clairement défini sur le site officiel.

Il est aussi possible de spécifier une version précise du paquet à l'aide de la syntaxe `paquet==version`, par exemple `pip install Pillow==9.0.1`. C'est la méthode conseillée dans votre répertoire de travail pour être sûr de la version utilisée.

On peut même préciser plusieurs noms de paquets à installer derrière `pip install`.

`pip install` accepte aussi une option `-r` suivie d'un nom de fichier, ce fichier devant contenir les dépendances à installer dans une syntaxe comprise par `pip install` :

```

1 Pillow==9.0.1
2 pyglet

```

Listing 85 – `requirements.txt`

```

1 % pip install -r requirements.txt
2 Collecting Pillow

```

```

3   Using cached Pillow-9.0.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.3 MB)
4   Collecting pygame==1.5.22
5     Downloading pygame-1.5.22-py3-none-any.whl (1.1 MB)
6     |██████████████████████████████████████| 1.1 MB 2.8 MB/s
7   Installing collected packages: pygame, Pillow
8   Successfully installed Pillow-9.0.1 pygame-1.5.22

```

À l'inverse, on peut supprimer un paquet installé à l'aide de la commande `uninstall`. La désinstallation demandera une confirmation (y) pour supprimer le paquet.

```

1 % pip uninstall Pillow
2 Found existing installation: Pillow 9.0.1
3 Uninstalling Pillow-9.0.1:
4   Would remove:
5     ...
6 Proceed (Y/n)? y
7   Successfully uninstalled Pillow-9.0.1

```

Il est aussi possible de lister les paquets installés avec `pip list`.

```

1 % pip list
2 Package      Version
3 -----
4 Pillow       9.0.1
5 pip          21.2.4
6 pygame       1.5.22
7 setuptools   58.1.0

```

`pip freeze` permet quant à lui d'extraire la liste des paquets installés dans une syntaxe comprise par `pip install`.

```

1 % pip freeze
2 Pillow==9.0.1
3 pygame==1.5.22

```



Il est ainsi courant d'utiliser la liste renvoyée par `pip freeze` pour former un fichier `requirements.txt` qui pourra ensuite permettre de réinstaller à l'identique ces mêmes dépendances dans un autre environnement à l'aide d'un `pip install -r requirements.txt`. Cela permet d'avoir un environnement reproductible d'une machine à l'autre.

Par défaut, `pip` fait appel à l'index *PyPI* (*Python Package Index*) pour trouver les paquets à installer, c'est le dépôt officiel des paquets Python, qui peut être consulté à cette URL : <https://pypi.org/>

Il est possible sur le site de faire des recherches sur l'index ou d'explorer les projets pour retrouver

des paquets. Chaque paquet vient avec une page de description et un ensemble de métadonnées. Voici par exemple [la page du paquet Pillow](#) .

Pour plus d'informations au sujet de *Pip*, vous pouvez vous reporter au [guide officiel](#) .

VIII.6.3. Environnements virtuels

En utilisant *Pip* comme nous venons de le faire, les paquets sont installés au niveau du système ou de l'utilisateur. C'est accommodant parce que le paquet est alors disponible partout et utilisable par tous les projets, mais cela peut parfois poser problème.

En effet, une seule version d'un paquet peut être disponible à la fois, ce qui fait que tous les projets doivent partager cette même version. Impossible alors pour un projet de bénéficier des évolutions récentes d'un module complémentaire si un autre projet dépend d'une version plus ancienne.

Pour résoudre ce problème, on va chercher à cloisonner nos applications, afin qu'elles gardent leurs dépendances (les modules complémentaires qu'elles doivent installer) auprès d'elles plutôt que de les installer sur tout le système. Et cela se fait à l'aide des environnements virtuels.

Un environnement virtuel n'est ni plus ni moins qu'un répertoire cloisonné de bibliothèques Python. On peut donc avoir autant d'environnements virtuels que l'on veut sur le système, qui contiendront chacun leurs bibliothèques dans les versions qu'ils veulent.

Mais bien sûr, dans un même environnement, une bibliothèque ne pourra être installée qu'en un seul exemplaire (donc une seule version).

Pour créer un nouvel environnement, on utilise la commande *shell* `python -m venv` suivie d'un nom, souvent `env` ou `venv`. Ce nom correspond au nom du répertoire qui sera créé pour l'environnement, depuis le répertoire courant donc.

On fera en sorte d'utiliser le répertoire du projet comme répertoire courant.

```
1 % python -m venv env
```

Cette commande a donc créé un dossier `env` dans le répertoire courant. Pour l'instant cet environnement est juste créé, mais pour l'utiliser nous devons l'activer.

Cela se fait à l'aide de la commande `source XXX/bin/activate` sous Linux/Mac ou `XXX\Scripts\Activate.ps1` sous Windows. Avec `XXX` remplacé par le nom du répertoire de l'environnement, `env` dans notre cas.

```
1 % source env/bin/activate
2 (env) %
```

On voit que le *prompt* de notre *shell* est maintenant préfixé d'un `(env)` pour signifier que nous sommes à l'intérieur de l'environnement.

Toutes les commandes que nous exécuterons maintenant (notamment les `pip install`) le seront à l'intérieur de cet environnement et n'affecteront pas le reste du système.



Il est nécessaire d'activer l'environnement virtuel chaque fois que vous ouvrez un nouveau terminal pour pouvoir l'utiliser.

VIII. La bibliothèque standard

Une fois votre travail terminé, si vous souhaitez sortir de l'environnement virtuel, vous pouvez utiliser la commande `deactivate`.

```
1 (env) % deactivate
2 %
```

Vous pouvez consulter [la page de documentation du module `venv`](#) pour de plus amples informations à son sujet.

Neuvième partie

Annexes

Introduction

IX.1. Glossaire et mots clés

IX.1.1. Glossaire

IX.1.1.0.1. annotation de type

Information facultative sur le **type** d'une **variable**, d'une **fonction** ou d'un **paramètre**. Utilisée par des outils tels que *mypy* pour vérifier la cohérence du code.

IX.1.1.0.2. argument

Valeur envoyée à une **fonction**, qui sera assignée à un **paramètre**. Les arguments peuvent être positionnels (une simple valeur dans la liste des arguments) ou nommés (préfixés du nom du paramètre : `end='\n'`).

IX.1.1.0.3. assertion

Prédicat évalué avec le **mot-clé** `assert` qui lève une **exception** s'il est faux.

IX.1.1.0.4. assignation

Affectation d'une **valeur** à une **variable**.

```
1 variable = 42
```

IX.1.1.0.5. attribut

Champ contenu dans un **objet**, données relatives à l'objet (`obj.attr`).

IX.1.1.0.6. bibliothèque standard (*stdlib*)

Ensemble des **modules**, **paquets** et **fonctions natives** embarquées avec Python par défaut.

IX.1.1.0.7. bloc

Élément de syntaxe qui réunit plusieurs lignes de code dans une même entité, introduit par une **instruction** particulière (**boucle**, **condition**, etc.) suivie d'un `:`.

IX.1.1.0.8. booléen

Type de **valeur** à deux états, *vrai* (`True`) ou *faux* (`False`).

IX.1.1.0.9. boucle

Bloc de code répété un certain nombre de fois (pour **itérer** avec une boucle `for` ou selon une **condition booléenne** avec un `while`).

```
1 while condition:
2     ...
3
4 for item in iterable:
5     ...
```

IX.1.1.0.10. boucle infinie

Boucle dont la condition de fin n'est jamais atteinte, qui ne s'arrête jamais.

IX.1.1.0.11. bytes

Valeur semblable aux **chaînes de caractères** pour représenter des **séquences** d'octets (des nombres entre 0 et 255). `b'a\x01b\x02'` est de type *bytes*.

IX.1.1.0.12. callable

Objet que l'on peut appeler, tel qu'une **fonction**.

```
1 >>> min(3, 4)
2 3
3 >>> int('123')
4 123
```

IX.1.1.0.13. chaîne de caractères (string)

Valeur représentant du texte, une **séquence** de caractères (`'abcdef'` par exemple).

IX.1.1.0.14. chemin (path)

Adresse d'un **fichier** sur le système d'exploitation.

IX.1.1.0.15. clé

Identifiant d'une **valeur** dans un **dictionnaire**. Seuls les types de données *hashables* peuvent être utilisés en tant que clés.

IX.1.1.0.16. condition

Bloc de code exécuté selon la **valeur** d'une **expression booléenne**.


```

1 if condition:
2     ...

```

IX.1.1.0.17. conteneur

Objet contenant des éléments (des **valeurs**), auxquels on peut généralement accéder par **itération** ou via l'opérateur `container[key]`.

IX.1.1.0.18. débogueur

Outil permettant de déceler pas-à-pas les bugs dans le code d'un programme.

IX.1.1.0.19. décorateur

Élément de syntaxe permettant de modifier le comportement d'une **fonction**, introduit par un `@` suivi du nom du décorateur avant la définition de la fonction (`@cache` par exemple).

```

1 @cache
2 def addition(a, b):
3     return a + b

```

IX.1.1.0.20. dictionnaire

Table d'association, pour associer des **valeurs** à des **clés**.

```

1 {'a': 'foo', 2: 3}

```

IX.1.1.0.21. docstring

Chaîne de caractères en en-tête d'une **fonction** pour documenter son comportement.

```

1 def addition(a, b):
2     "Addition entre deux nombres"
3     return a + b

```

IX.1.1.0.22. EAFP

Easier to Ask Forgiveness than Permission (il est plus simple de demander pardon que demander la permission), mécanisme de traitement des erreurs qui préconise de laisser se produire les **exceptions** pour les attraper ensuite (*demander pardon*).

EAFP s'oppose à **LYBL**.

IX.1.1.0.23. éditeur de texte

Logiciel permettant de modifier des **fichiers** texte (ou fichiers de code) sur le système d'exploitation.

IX.1.1.0.24. encodage

Norme de codage des caractères dans une **chaîne de caractères**, associe chaque caractère (lettres, chiffres, symbole, accents, etc.) à un nombre.

IX.1.1.0.25. ensemble (set)

Conteneur non-ordonné composé de **valeurs** uniques et **hashables**.

```
1 {'a', 'b', 'c'}
```

IX.1.1.0.26. entrée standard

Flux de données en entrée du programme, le terminal par défaut, sollicité par la **fonction** `input`. Correspond à `sys.stdin`.

IX.1.1.0.27. environnement virtuel

Répertoire cloisonné de **paquets** Python.

IX.1.1.0.28. exception

Comportement permettant de remonter des erreurs dans le programme afin de les traiter.

IX.1.1.0.29. expression

Ensemble d'opérations Python qui produisent une **valeur**.

```
1 >>> (1 + 2 * 3) / 5 + round(1/3, 2)
2 1.73
```

IX.1.1.0.30. fichier

Document sur le système d'exploitation (adressé par un **chemin**), représenté en Python par un **objet** qui permet d'interagir avec lui (lire son contenu, écrire dans le fichier, etc.).

IX.1.1.0.31. fonction

Opération recevant des **arguments** et renvoyant une nouvelle **valeur** en fonction de ceux-ci (fonctions mathématiques par exemple : `round`, `abs`).

```

1 >>> round(3.5)
2 4
3 >>> abs(-2)
4 2

```

IX.1.1.0.32. fonction native (*builtin*)

Fonction disponible directement dans l'interpréteur, sans **import**.

IX.1.1.0.33. fonction récursive

Fonction qui se rappelle elle-même pour mettre en place un mécanisme de répétition.

```

1 def my_len(s):
2     if s:
3         return 1 + len(s[1:])
4     return 0

```

IX.1.1.0.34. formatage

Action d'obtenir une **représentation** d'une **valeur** dans un format voulu.

IX.1.1.0.35. *f-string*

Chaîne de **formatage**, élément de syntaxe permettant de composer facilement des **chaines de caractères**.

```

1 >>> f"1 + 3 = {1+3}"
2 '1 + 3 = 4'

```

IX.1.1.0.36. gestionnaire de contexte

Bloc permettant de gérer des ressources (telles que des **fichiers**).

```

1 with open('file.txt') as finput:
2     ...

```

IX.1.1.0.37. *hashable*

Valeur qui peut être utilisée en tant que **clé** de **dictionnaire** ou contenue dans un **ensemble**. Le *hash* est un «code-barre» généré à partir de la valeur, qui permet de la retrouver : le *hash* d'un objet ne doit pas changer et deux valeurs égales doivent avoir le même *hash*. Les **types immutables** natifs de Python sont *hashables* tandis que les **mutables** ne le sont pas.

IX.1.1.0.38. IDLE

Interactive DeveLopment Environment, l'environnement de développement fourni avec Python.

IX.1.1.0.39. import

Instruction qui permet de charger le code d'un **module** Python.

```
1 import math
```

IX.1.1.0.40. instruction

Élément de syntaxe de Python au sens large, souvent équivalent à une ligne de code.

IX.1.1.0.41. intension

Manière de créer des **listes** / **ensembles** / **dictionnaires** par **itération**.

```
1 >>> [i**2 for i in range(5)]
2 [0, 1, 4, 9, 16]
3 >>> {i**2 for i in range(5)}
4 {0, 1, 4, 9, 16}
5 >>> {i**2: i for i in range(5)}
6 {0: 0, 1: 1, 4: 2, 9: 3, 16: 4}
```

IX.1.1.0.42. interpréteur interactif / REPL

Mode de l'interpréteur de Python qui permet d'entrer les **instructions** et de les exécuter directement, en affichant les **valeurs** des **expressions**.

REPL pour *Read-Eval-Print-Loop*, soit *boucle qui lit, évalue et affiche*.

IX.1.1.0.43. introspection

Caractéristique d'un programme qui est capable de s'inspecter lui-même (parcourir les **attributs** de ses objets, explorer les **méthodes**, etc.).

IX.1.1.0.44. itérable

Valeur sur laquelle on peut **itérer** à l'aide d'une **boucle** `for`, appliquer un traitement sur chacun des éléments.

```
1 for item in [3, 2, 5, 8]:
2     ...
```

IX.1.1.0.45. itérateur

Curseur le long d'un **itérable**, utilisé par les **boucles** `for` pour les parcourir.

IX.1.1.0.46. itération / itérer

Action de parcourir les éléments d'un **itérable** avec un **itérateur**.

IX.1.1.0.47. LBYL

Look Before You Leap (*réfléchis avant d'agir*), mécanisme de traitement des erreurs qui préconise d'empêcher les erreurs en vérifiant les conditions de réussite au préalable.

LBYL s'oppose à *EAFP*.

IX.1.1.0.48. liste

Séquence mutable d'éléments de **types** variables.

```
1 [1, 2, 3, 4]
2 ['a', 42, 1.5, [0]]
```

IX.1.1.0.49. littéral

Élément de syntaxe de base qui possède une **valeur**, comme les **chaînes de caractères**, les **listes** ou les **dictionnaires**.

IX.1.1.0.50. méthode

Fonction intégrée à un **objet**, opération spécifique à un **type**.

```
1 >>> [1, 2, 3].pop()
2 3
```

IX.1.1.0.51. module

Fichier de code Python, que l'on peut charger à l'aide d'un **import**.

IX.1.1.0.52. mot-clé

Élément de syntaxe formé de lettres correspondant à une **instruction** ou un **opérateur** du langage. Les mots-clés ne peuvent pas être utilisés comme noms de **variables**.

IX.1.1.0.53. mutable / immutable

Une **valeur** mutable est une valeur modifiable, que l'on peut altérer (les **listes** par exemple) contrairement à une valeur immutable (comme les **tuples**).

```
1 >>> values = [1, 2, 3]
2 >>> values[0] = 4
3 >>> values
4 [4, 2, 3]
```

```

5 >>> values = (1, 2, 3)
6 >>> values[0] = 4
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   TypeError: 'tuple' object does not support item assignment

```

IX.1.1.0.54. nombre complexe

Représentation d'un nombre complexe en Python, formé de deux **flottants** (partie réelle et partie imaginaire suffixée d'un **j**) comme `3.2+5j`.

IX.1.1.0.55. nombre entier

Représentation d'un nombre entier relatif (positif ou négatif) en Python. Suite de chiffres potentiellement précédée d'un `+` ou d'un `-`, comme `101` ou `-42`.

IX.1.1.0.56. nombre flottant

Représentation d'un nombre réel en Python, formé d'une partie entière et d'une partie décimale, comme `10.0`, `8.3` ou `5e2`.

IX.1.1.0.57. objet / valeur

Résultat d'une **expression**, qui peut être stocké dans une **variable**. Toute valeur Python est un objet, et peut posséder des **attributs** et **méthodes**.

IX.1.1.0.58. opérateur

Élément de syntaxe (**mot-clé** ou caractères spéciaux) représentant une opération en Python, comme `+` pour l'addition.

IX.1.1.0.59. paquet (*package*)

Niveau d'indirection au-dessus du **module** : un module qui contient des modules. S'utilise aussi pour parler des bibliothèques tierces, installables via **pip** (gestionnaire de paquets).

IX.1.1.0.60. paramètre

Variable d'une **fonction** dont la **valeur** sera automatiquement **assignée** selon un **argument**. Un paramètre peut choisir d'accepter les arguments positionnels ou nommés, et posséder une valeur par défaut.

```

1 def func(param1, param2=None):
2     pass

```

IX.1.1.0.61. PEP

Python Enhancement Proposal, soit *Proposition d'amélioration pour Python*, c'est par là que passent les demandes de fonctionnalités au langage, avant d'être acceptées ou non.

IX.1.1.0.62. prédicat

Expression booléenne, utilisée dans une condition.

IX.1.1.0.63. pythonique

Qualificatif de ce qui est idiomatique en Python, qui correspond à la philosophie du langage. Voir [cet article](#) à propos du code pythonique.

IX.1.1.0.64. représentation

Chaîne de caractères obtenue à partir d'une **valeur**, qui permet d'en décrire le contenu.

IX.1.1.0.65. retour

Valeur renvoyée par une **fonction**. Donne sa valeur à l'**expression** d'appel de la fonction.

```
1 >>> x = abs(-1)
2 >>> x
3 1
```

IX.1.1.0.66. séquence

Conteneur dont les clés sont des nombres entiers de 0 à N-1 (avec N la taille du conteneur).

IX.1.1.0.67. scope

Espace de noms, là où sont déclarées les **variables**.

IX.1.1.0.68. slicing

Découpage d'une séquence selon un intervalle.

```
1 >>> 'abcdefghi'[1:-1:2]
2 'bdfh'
3 >>> [1, 2, 3, 4][1:]
4 [2, 3, 4]
```

IX.1.1.0.69. sortie d'erreur

Flux de données en sortie du programme dédié aux erreurs, le terminal par défaut. Correspond à `sys.stderr`.

IX.1.1.0.70. sortie standard

Flux de données en sortie du programme où sont affichés les messages (par appel à `print` par exemple), le terminal par défaut. Correspond à `sys.stdout`.

IX.1.1.0.71. test

Fonction composée d'**assertions** pour vérifier le bon comportement d'un code.

IX.1.1.0.72. tuple

Séquence **immutable** d'éléments de **types** variables.

```
1 (1, 2, 3, 4)
2 ('a', 42, 1.5, [0])
```

IX.1.1.0.73. tuple nommé

tuple dont les éléments peuvent aussi être accédés via des **attributs**.

```
1 from collections import namedtuple
2 Point = namedtuple('Point', ('x', 'y'))
3 p = Point(3, 5)
4 print(p.x, p.y)
```

IX.1.1.0.74. type

Toute **valeur** en Python possède un **type**, qui décrit les opérations et **méthodes** qui lui sont applicables.

IX.1.1.0.75. variable

Étiquette posée sur une **valeur** par **assignation**. Plusieurs variables peuvent correspondre à la même valeur.

```
1 variable = 42
```

IX.1.1.0.76. zen

Zen of Python, ou **PEP 20**, sorte de poème qui décrit la philosophie du langage : <https://www.python.org/dev/peps/pep-0020/> ↗ (traduction ↗).

IX.1.2. Tableau des mots-clés

Voici le tableau de l'ensemble des mots-clés de Python :

Faanel	im	pass
Ncbr	ex	irraise
Trcl	fi	re
an	cont	fclatrya
as	de	gl
se	de	ncwith
aseli	for	yield

À cela on pourrait aussi ajouter les mots-clés `match` et `case`, qui sont bien des mots-clés mais pas des noms réservés (vous pouvez nommer une variable `match` ou `case` sans soucis).

- **False**: Valeur **False** du type booléen.
- **None**: Valeur **None**, qui représente l'absence de valeur.
- **True**: Valeur **True** du type booléen.
- **and**: Opération booléenne ET (conjonction).

```
1 >>> True and False
2 False
```

- **as**: Permet une assignation si couplé à un autre mot-clé. (`import`, `with`, `except`).

```
1 import math as m
```

```
1 with open('file') as f:
2     ...
```

```
1 try:
2     ...
3 except ValueError as e:
4     ...
```

- **assert**: Assertion, échoue si l'expression donnée est fausse (les assertions ne sont pas exécutées si Python est lancé en mode optimisé -O).

```
1 assert 5 == 4 + 1
```

- `async`: Introduit une fonction asynchrone (`async def`)¹³.

- **await**: Attend un résultat asynchrone (depuis une fonction asynchrone)¹³.
- **break**: Permet de sortir immédiatement d'une boucle. En cas de boucles imbriquées, le mot-clé affecte la boucle intérieure uniquement.

```
1 while condition:
2     ...
3     break
```

- **case**: Introduit un motif de filtrage dans un bloc **match**²³.
- **class**: Définit une classe en programmation orientée objet⁴.
- **continue**: Permet de passer à l'itération suivante de la boucle. En cas de boucles imbriquées, le mot-clé affecte la boucle intérieure uniquement.

```
1 while condition:
2     ...
3     continue
```

- **def**: Définit une fonction.

```
1 def func(a, b):
2     ...
```

- **del**: Supprime une variable ou un élément d'un conteneur.

```
1 del var
```

```
1 del container[key]
```

- **elif**: Condition *sinon-si* dans un bloc conditionnel.

```
1 if condition:
2     ...
3 elif other_condition:
4     ...
```

- **else**: Condition *sinon* dans un bloc conditionnel, ou deuxième clause d'une expression conditionnelle.

```
1 if condition:
2     ...
3 else:
4     ...
```

```
1 true_val if condition else false_val
```

Peut aussi se placer après un bloc `for/while` (réagir en cas de sortie de boucle prématurée) ou `try` (réagir si tout s'est bien passé).

- `except`: Attrape une exception après un bloc `try`.

```
1 try:
2     ...
3 except ValueError:
4     ...
```

- `finally`: Exécute des instructions dans tous les cas après un bloc `try`.

```
1 try:
2     ...
3 finally:
4     ...
```

- `for`: Introduit une boucle d'itération. Peut aussi introduire une intension (liste, ensemble, etc.).

```
1 for item in iterable:
2     ...
```

```
1 [... for item in iterable]
```

- `from`: Réalise un import dans l'espace de nom courant, conjointement avec `import` (`from ... import`).

```
1 from collections import Counter
```

- `global`: Déclare une variable comme étant globale.

```
1 global var
```

- `if`: Introduit un bloc conditionnel avec une condition *si*. Peut aussi introduire une expression conditionnelle ou une condition de filtrage dans une intension.

```
1 if condition:
2     ...
```

```
1 true_val if condition else false_val
```

```
1 [... for item in iterable if condition]
```

- `import`: Réalise un import, utilisé seul (import simple) ou conjointement avec `from` (`from ... import`).

```
1 import math
```

```
1 from collections import Counter
```

- `in`: Opérateur d'appartenance, teste si une valeur est présente dans un conteneur.

```
1 value in container
```

- `not in`: Opérateur de non-appartenance, teste si une valeur est absente d'un conteneur.

```
1 value not in container
```

- `is`: Opérateur d'identité.

```
1 value is None
```

- `is not`: Opérateur de non-identité.

```
1 value is not None
```

- `lambda`: Introduit une fonction lambda.

```
1 lambda x: x**2
```

- `match`: Introduit un bloc de filtrage par motif²³.
- `nonlocal`: Déclare une variable comme non-locale⁵.
- `not`: Opération booléenne NON (négation).

```
1 >>> not True
2 False
```

- `or`: Opération booléenne OU (disjonction).

```

1 >>> True or False
2 True

```

- `pass`: Ne fait rien, ne renvoie rien (utile quand un bloc indenté est attendu).

```

1 if True:
2     pass

```

- `raise`: Lève une exception.

```

1 raise ValueError()

```

- `return`: Renvoie une valeur depuis une fonction (la fonction se termine au premier `return`).

```

1 def f(a, b):
2     return ...

```

- `try`: Introduit un bloc de traitement d'exception.

```

1 try:
2     ...
3 except:
4     ...

```

- `while`: Introduit une boucle sur une condition.

```

1 while condition:
2     ...

```

- `with`: Introduit un gestionnaire de contexte (pour ouvrir un fichier par exemple).

```

1 with open('file') as f:
2     ...

```

- `yield`: Produit une valeur depuis un générateur³.

1. Introduit en [Python 3.5](#) .

2. Introduit en [Python 3.10](#) .

3. Non abordé dans ce cours.

4. Non abordé, mais c'est l'objet du cours [sur la programmation objet en Python](#) .

5. Non abordé, mais introduit dans [ce tutoriel sur les scopes](#) .

IX.1.3. Tableau des opérateurs

IX.1.3.1. Opérateurs simples (expressions)

En plus des mots-clés précédents, on trouve aussi les opérateurs suivants. Ces opérateurs sont constitués de caractères spéciaux et ne sont donc pas des noms.

+	%	&	!=	:	=
-	*	*		<	<
*	x	(^	.	>
/	x	.	~	t	<=
/	/	x	[=	>=

— **+** : Addition / concaténation, ou opérateur unaire positif.

```

1 >>> 3 + 5
2 8
3 >>> 'abc' + 'def'
4 'abcdef'
5 >>> +42
6 42

```

— **-** : Soustraction / différence ou opérateur unaire négatif.

```

1 >>> 3 - 5
2 -2
3 >>> -42
4 -42

```

```

1 >>> {1, 2, 3} - {2, 3, 4}
2 {1}

```

— ***** : Multiplication, concaténation multiplicative, ou opérateur *splat*.

```

1 >>> 3 * 5
2 15
3 >>> 'cou' * 2
4 'coucou'
5 >>> [3] * 4
6 [3, 3, 3, 3]

```

```
1 func(*[1, 2, 3])
```

— `/` : Division ou séparateur de chemins.

```
1 >>> 3 / 5
2 0.6
```

```
1 >>> Path('a') / Path('b')
2 PosixPath('a/b')
```

— `//` : Division entière (euclidienne).

```
1 >>> 10 // 3
2 3
```

— `%` : Modulo (reste de division) ou formatage de chaîne.

```
1 >>> 10 % 3
2 1
```

```
1 >>> 'salut %s' % 'toto'
2 'salut toto'
```

— `**` : Exponentiation (puissance) ou *double-splat*

```
1 >>> 5 ** 3
2 125
```

```
1 func(**{'arg': 42})
```

— `x(...)` : Appel de fonction (ou *callable*), instantiation de type.

```
1 >>> round(3.5)
2 4
3 >>> list()
4 []
```

— `x.attr` : Accès à un attribut.

```

1 >>> Path('a/b').name
2 'b'

```

— `x[...]` : Accès à un élément. Permet aussi le *slicing*.

```

1 >>> squares[3]
2 9
3 >>> squares[4:8]
4 [16, 25, 36, 49]

```

— `&` : Conjonction (*ET*) bit-à-bit ou intersection d'ensembles.

```

1 >>> bin(0b101 & 0b110)
2 '0b100'

```

```

1 >>> {1, 2, 3} & {2, 3, 4}
2 {2, 3}

```

— `|` : Disjonction (*OU*) bit-à-bit ou union d'ensembles.

```

1 >>> bin(0b101 | 0b110)
2 '0b111'

```

```

1 >>> {1, 2, 3} | {2, 3, 4}
2 {1, 2, 3, 4}

```

— `^` : *XOR* bit-à-bit ou différence symétrique d'ensembles.

```

1 >>> bin(0b101 ^ 0b110)
2 '0b11'

```

```

1 >>> {1, 2, 3} ^ {2, 3, 4}
2 {1, 4}

```

— `~` : Négation (*NON*) bit-à-bit, opérateur unaire.

```

1 >>> bin(~0b101)
2 '-0b110'

```

— `==` : Test d'égalité.


```

1 >>> 5 == 4 + 1
2 True

```

— `!=` : Test de différence.

```

1 >>> 5 != 4 + 1
2 False

```

— `<` : Test d'infériorité stricte.

```

1 >>> 3 < 5
2 True

```

— `>` : Test de supériorité stricte.

```

1 >>> 3 > 5
2 False

```

— `<=` : Test d'infériorité.

```

1 >>> 3 <= 5
2 True
3 >>> 3 <= 3
4 True

```

— `>=` : Test de supériorité.

```

1 >>> 3 >= 5
2 False
3 >>> 3 >= 3
4 True

```

— `:=` : Expression d'assignation².

```

1 >>> if x:= 5:
2 ...     print(x+1)
3 ...
4 6

```

— `<` : Décalage de bits à gauche.

```

1 >>> bin(0b101 << 2)
2 '0b10100'

```

— `>` : Décalage de bits à droite.

```

1 >>> bin(0b10101 >> 2)
2 '0b101'

```

— `@` : Multiplication matricielle¹³.

— `,` : La virgule est un peu à part, c'est un séparateur (arguments, listes, etc.) mais aussi l'opérateur qui permet de créer des tuples.

```

1 >>> 1,
2 (1,)
3 >>> 3, 4, 5
4 (3, 4, 5)

```

IX.1.3.2. Opérateurs d'assignation

=	*	%		=>	>
+=	/	=*	^	@	=
-	/	&	=	<	<
=					

Les opérations d'assignation suivent toutes le même principe, `var = expression`.

```

1 >>> x = 42
2 >>> x
3 42

```

L'opérateur utilisé applique simplement l'opération cible (+ pour += etc.) entre la variable initiale et l'expression.

```

1 >>> x += 2 # x = x + 2
2 >>> x
3 44
4 >>> x //= 3 # x = x // 3
5 >>> x
6 14

```

-
1. Introduit en [Python 3.5](#) [↗](#).
 2. Introduit en [Python 3.8](#) [↗](#).
 3. Non abordé dans ce cours.



Attention, certaines assignations peuvent s'opérer en-place sur l'objet. `a += b` n'est donc pas strictement équivalent à `a = a + b`.

```
1 >>> values = [1, 2]
2 >>> copy = values
3 >>> values += [3]
4 >>> copy
5 [1, 2, 3]
```

Les opérations d'assignation permettent aussi d'assigner les éléments des conteneurs.

```
1 values[0] = 42
2 dic[key] = value
```

IX.1.4. Priorité des opérateurs

IX.1.4.1. Ordre d'évaluation des expressions

Les expressions en Python sont toujours évaluées de la gauche vers la droite, à l'exception près des assignations où la partie droite sera évaluée avant la partie gauche.

Ainsi, dans les exemples fictifs suivants, `expr1` sera toujours évaluée avant `expr2`, elle-même avant `expr3`, etc. jusqu'à `expr5`.

```
1 expr1, expr2, expr3, expr4
2 (expr1, expr2, expr3, expr4)
3 {expr1: expr2, expr3: expr4}
4 expr1 + expr2 * (expr3 - expr4)
5 expr1(expr2, expr3, *expr4, **expr5)
6 expr3, expr4 = expr1, expr2
```

IX.1.4.2. Tableau des priorités

Aussi, dans une même expression, les opérations seront exécutées par ordre de priorité. Dans le tableau suivant, les opérations de rang inférieur seront exécutées prioritairement à celles de rang supérieur (`**` est évalué avant `/`).

Opé-	Des-
Rang	Rang
infé-	supé-
rieur	rieur

		Ex- pres- sions entre pa- ren- thèses, cro- (.che)s
1		[.(listes) {.ou } ac- co- lades (dic- tion- naires, en- sembles)
		Ap- pels, ac- cès x(aux), x[élé-], x.attrib
2		et at- tri- buts
		Ex- pres- sions await
3		await
		**Exponentiations ¹
4		+, Opé- - ra- , teurs ~ unaires
5		

6	Opé- ra- teurs bi- *, naires @, de /, mul- //ti- %pli- ca- tions/di- vi- sions
7	Opé- ra- teurs bi- +, naires - d'ad- di- tion/sous- trac- tion
8	Dé- <, ca- > lages de bits
9	Conjonc- tions bit- à- & bit, in- ter- sec- tions
10	<i>XOR</i> bit- à- bit, dif- fé- rences sy- mé- triques

		Dis- junc- tions
11		bit- à- bit, unions
		inc
		Com- not- rai- is- sons, is- tests
12		not- ap- <, par- <=e- >, nance >=et !=d'iden- =tité
		Né- ga- tions
13		not- boo- léennes
		Conjunc- tions
14		and boo- léennes
		Dis- junc- tions
15		or boo- léennes
		Ex- pres- sions condi- ti- on- nelles
		Fonc- tions
17		lambda lambda
		Ex- pres- sions
18	:	=d'as- si- gna- tion

	Sé-
	pa-
19,	ra-
	teurs,
	tuples

Les exemples de cette section sont tirés de la page de documentation [Référence sur les expressions](#) [↗](#), sur laquelle vous trouverez plus d’informations au sujet des expressions et des priorités des opérateurs.

IX.1.5. Autres éléments de syntaxe

Certains éléments font partie de la syntaxe Python sans être inclus dans les tableaux précédents, les voilà détaillés ici.

Élé-	Des-
ments	crip-
de	tion
scrip-	tion
tion	taxe
	Chaînes
	de
	ca-
	'rac',
	b'ères',
	f'ou .',
	et bytes
	lit-
	té-
	rales
42,	Nombres
	en-
10,	tiers
100_000,	lit-
0b1,	té-
0o7,	raux
0xF	Nombres
	flot-
1.5,	tants
	lit-
1e10	té-
	raux

1. On note cependant que l’opérateur `**` est moins prioritaire qu’un opérateur unaire sur son opérande de droite. Ainsi `10**-2` s’évalue comme `10**(-2)` (mais `-10**2` s’évalue bien comme `-(10**2)`).

	Nombres
	ima-
3j	gi-
-	naires
1.8j	(com-
2e5j	plexes)
	lit-
	té-
	raux
	Listes
[item1,	item1,
item2]	item2]
	rales
	En-
{item1,	sembles
item2}	lit-
	té-
	raux
	Dic-
{k1,	non-
v1,	naires
k2,	lit-
v2,	té-
	raux
va-	va-
riable	riable
	El-
..lip-	lip-
	sis
	In-
	tro-
:	duc-
(if	tion
..d'un	
:)	bloc
	de
	code
#	Com-
com-	men-
ment	ment
	raux

1. Inclut aussi des double-guillemets, *triple-quotes* et les différents préfixes.

IX.2. Notes diverses

Introduction

J'aimerais ici vous présenter des petits sujets variés, indépendants les uns des autres, qui sont utiles mais n'ont pas forcément leur place dans les autres chapitres du cours.

IX.2.1. En-têtes de fichiers

Il est possible d'ajouter des lignes d'en-tête à nos fichiers Python. Il s'agit d'instructions spéciales que l'on place au tout début du fichier, avant les premières lignes de code.

i

Ces en-têtes sont facultatives et concernent des cas particuliers qui sont décrits au long du cours. Elles pourront simplement vous être utiles si par la suite vous vous trouvez dans l'un des cas concernés.

Aussi, pour simplifier les exemples donnés dans le cours, je n'y ferai jamais figurer ces en-têtes.

IX.2.1.0.1. Shebang

La première dont je veux vous parler est ce qu'on appelle le *shebang*. C'est une instruction qui permet à certains systèmes (Linux notamment) de reconnaître un fichier exécutable comme un programme Python (ou plus précisément de savoir avec quoi lancer cet exécutable).

Celle-ci n'est utile que pour le ou les fichiers principaux d'un projet Python, ceux qui seront amenés à être exécutés directement.

Le *shebang* est une ligne qui prend la forme suivante, vous verrez parfois `python3` à la place de `python`.

```
1 #!/usr/bin/env python
```

Elle définit quel programme utiliser pour exécuter le fichier. Ici on fait appel à la commande `env` pour localiser le programme `python` et c'est ce dernier qui exécutera notre fichier.

On trouve parfois aussi `#!/usr/bin/python` qui stipule directement le chemin du programme Python mais est moins portable d'un environnement à un autre.

Cela permet ensuite pour un fichier Python `programme.py` disposant des droits d'exécution (`chmod +x programme.py`) d'être exécuté à l'aide d'un simple `./programme.py` depuis le répertoire courant.

```
1 % ./programme.py
2 Hello World!
```

IX.2.1.0.2. Encodage

La seconde est la déclaration de l'encodage du fichier, qui permet à Python de savoir comment le décoder. En effet notre ordinateur est rudimentaire et ne sait pas ce qu'est du texte, de son point de vue il ne manipule que des nombres.

Un encodage c'est une règle qui lui décrit comment convertir chaque caractère utilisé dans le fichier (notamment les caractères spéciaux et les lettres accentuées) en nombres.

Aujourd'hui l'encodage le plus courant est UTF-8, et c'est celui que je vous recommande d'utiliser pour vos fichiers. Il est utilisé par défaut par Python, ainsi que dans IDLE et Geany.

Mais certains systèmes d'exploitation (Windows pour ne pas le citer) pourraient ne pas l'utiliser par défaut, et si c'est le cas de votre éditeur de texte, alors il faudra préciser à Python quel encodage utiliser pour lire le fichier.

Cela se fait à l'aide d'une ligne telle que :

```
1 # coding: xxx
```

Où `xxx` serait remplacé par l'encodage utilisé dans le fichier (`utf-8`, `latin-1`, `windows_1252`, etc.).

IX.3. Quelques modules complémentaires bien utiles

Introduction

Voyons maintenant quelques bibliothèques tierces en Python assez connues, que vous pourriez être amenés à utiliser pour répondre à différents besoins.

IX.3.1. Requests

Requests est une bibliothèque Python de référence, qui permet de réaliser des requêtes HTTP (interroger des sites web). Elle est connue pour sa facilité d'utilisation.

On l'installe via `pip install requests`.

Par exemple on peut l'utiliser pour interroger l'API de Zeste de Savoir, ici pour obtenir la liste des tags du site.

```
1 >>> import requests
2 >>> resp = requests.get('https://zestedesavoir.com/api/tags')
3 >>> resp.status_code
4 200
5 >>> resp.text
6 '{"count":5071,"next":"https://zestedesavoir.com/api/tags/?page=2",
  "previous":null,"results":[{"id":1,"title":"exercice"}, {"id":2,
  "title":"java"}, {"id":3,"title":"langages
  oo"}, {"id":4,"title":"urgent"}, {"id":5,"title":"bug"}, {"id":6,
  "title":"suggestion"}, {"id":7,"title":"transmis"}, {"id":8,"tit
  le":"blog"}, {"id":9,"title":"régression"}, {"id":10,"title":"fr
  ont"}]}'
7 >>> resp.json()
8 {'count': 5071,
9  'next': 'https://zestedesavoir.com/api/tags/?page=2',
10 'previous': None,
11 'results': [{'id': 1, 'title': 'exercice'},
12              {'id': 2, 'title': 'java'},
13              {'id': 3, 'title': 'langages oo'},
14              {'id': 4, 'title': 'urgent'},
15              {'id': 5, 'title': 'bug'},
16              {'id': 6, 'title': 'suggestion'},
17              {'id': 7, 'title': 'transmis'},
18              {'id': 8, 'title': 'blog'},
19              {'id': 9, 'title': 'régression'},
```

20	<code>{'id': 10, 'title': 'front'}}</code>
----	--

Pour plus d'informations au sujet de *Requests*, je vous invite à consulter [sa documentation](#) [↗](#).

IX.3.2. Numpy

Numpy est une bibliothèque dédiée au calcul numérique, offrant de bonnes performances pour ces opérations. Elle est très utilisée dans le domaine scientifique (*data science*).

Elle s'installe facilement avec *Pip* : `pip install numpy`.

Ensuite c'est une bibliothèque qui permet notamment de manipuler des tableaux de données (de même type), et d'effectuer des opérations en lots sur les valeurs de ces tableaux.

```

1  >>> import numpy as np
2  >>> array = np.array([[1, 2, 3], [4, 5, 6]])
3  >>> array.size # nombre de valeurs
4  6
5  >>> array.ndim # nombre de dimensions
6  2
7  >>> array.shape # taille des dimensions
8  (2, 3)
9  >>> array.dtype # type des valeurs
10 dtype('int64')
11 >>> array
12 array([[1, 2, 3],
13        [4, 5, 6]])
14 >>> array + 1
15 array([[2, 3, 4],
16        [5, 6, 7]])
17 >>> array * 3
18 array([[ 3,  6,  9],
19        [12, 15, 18]])
20 >>> array + np.arange(10, 16).reshape(2, 3)
21 array([[11, 13, 15],
22        [17, 19, 21]])

```

Pour aller plus loin, rendez-vous [sur la documentation de la bibliothèque Numpy](#) [↗](#).

Intéressez-vous aussi aux bibliothèques [SciPy](#) [↗](#) (calculs numériques, calculs formels) et [pandas](#) [↗](#) (analyse de données) construites autour de *Numpy* qui s'utilisent conjointement avec elle.

La [matplotlib](#) [↗](#), une bibliothèque de visualisation et de dessin de graphiques en Python est aussi couramment utilisée avec *Numpy*, [un cours](#) [↗](#) est même disponible sur Zeste de Savoir à son sujet.

IX.3.3. Django

Django est la référence en tant que *framework* pour le web, c'est une bibliothèque très complète qui permet de mettre en place en site Internet en Python. *Django* embarque par exemple de quoi abstraire la base de données en modèles (types Python) et générer des pages à partir de

gabarits (*templates*).

Zeste de Savoir par exemple est codé avec *Django* : <https://github.com/zestedesavoir/zds-site> ↗

Django s'installe à l'aide de la commande `pip install Django`.

Un [tutoriel](#) ↗ sur le site permet d'apprendre à utiliser *Django*, mais il n'est plus maintenu à jour, vous pouvez alors vous orienter [vers le tutoriel officiel et la documentation traduits en français](#) ↗.

Sur le site, vous trouverez un autre tuto pour [apprendre à déployer une application Django](#) ↗

Django n'est pas la seule bibliothèque pour créer des sites web. *Flask* ↗ est notamment connu pour être un *framework* plus léger (mais moins complet).

IX.3.4. Pillow

Pillow est un paquet Python dédié à l'imagerie, c'est une bibliothèque qui permet de lire et créer des images.

Le paquet *Pip* s'appelle *Pillow* (`pip install Pillow`), mais c'est ensuite un module `PIL` qu'il faut importer depuis le code. *PIL* était l'ancien nom du paquet (en Python 2), *Python Image Library*.

```
1 from PIL import Image, ImageDraw
2
3 img = Image.new('RGB', (400, 300), color='red') # crée une
    nouvelle image
4 draw = ImageDraw.Draw(img)
5 draw.rectangle(((100, 100), (300, 200)), fill='blue')
6 img.show() # affiche l'image dans une fenêtre
7 img.save('output.png') # enregistre l'image
```

Des informations complémentaires sur *Pillow* peuvent être trouvée dans [la documentation de la bibliothèque](#) ↗.

IX.3.5. PyGObject (PyGTK)

PyGObject (anciennement *PyGTK*) est une bibliothèque qui permet de créer des programmes fenêtrés, des interfaces graphiques riches (*GUI*) formées de zones de texte, de menus ou encore de boutons.

Elle repose sur *Gtk*, un module logiciel écrit en C pour réaliser de telles interfaces, notamment utilisé par Gimp, Firefox ou encore Gnome.

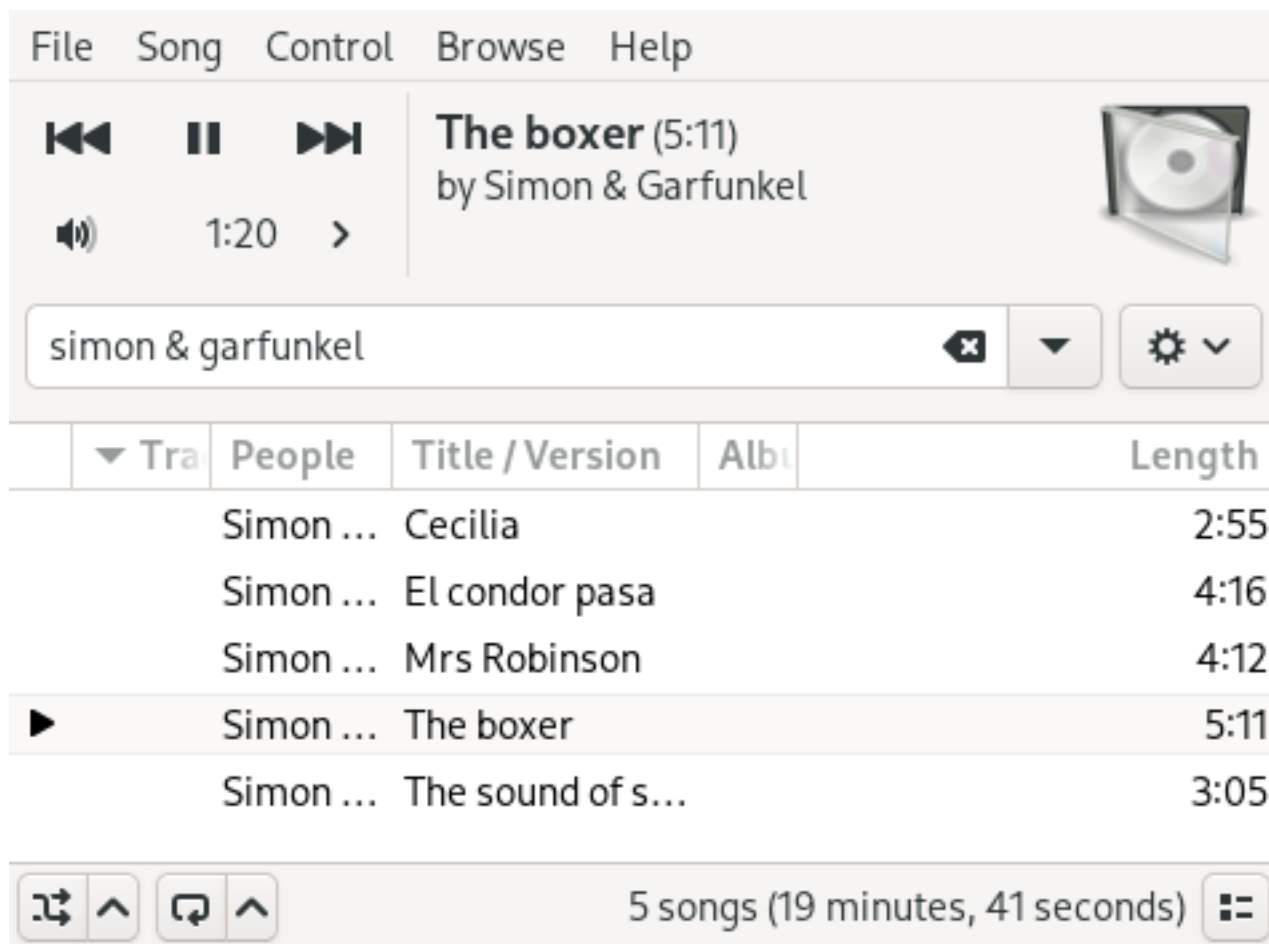


FIGURE IX.3.1. – Capture d'écran de *Quod Libet*, un lecteur de musique écrit en Python utilisant *PyGtk*.

La procédure d'installation de *PyGObject* varie selon les systèmes, je vous laisse donc consulter [cette page](#) pour trouver celle qui vous convient.

Ensuite je vous invite à lire [ce tutoriel](#) de @Wizix sur Zeste de Savoir pour apprendre comment prendre en main cette bibliothèque. Vous pouvez aussi vous reporter au [tutoriel officiel](#) (en anglais).

Voici néanmoins un code permettant de réaliser une petite interface.

```

1  import gi
2  # Vérification de la version de Gtk
3  gi.require_version("Gtk", "3.0")
4
5  from gi.repository import Gtk
6
7  # Création d'une fenêtre
8  window = Gtk.Window(title='Hello World')
9  box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=6)
10 window.add(box)
11
12 # Ajout d'éléments graphiques
13

```

```

14 label = Gtk.Label(label='Salut')
15 box.pack_start(label, True, True, 0)
16
17 button = Gtk.Button(label='Clique !')
18 box.pack_start(button, True, True, 0)
19
20 # Connexion des événements et affichage
21
22 window.show_all()
23 window.connect('destroy', Gtk.main_quit)
24 button.connect('clicked', Gtk.main_quit)
25
26 Gtk.main()

```

Consultez aussi [le site officiel](#) pour aller plus loin avec *PyGtk*.

i

Bon à savoir : la bibliothèque standard de Python embarque le module [tkinter](#) pour créer des programmes fenêtrés, mais qui n'est pas forcément le module le plus abordable pour cela.

Sur certains systèmes (Debian/Ubuntu par exemple) il n'est pas présent par défaut et il faut installer le paquet *APT* `python3-tk`.

Voir [Programmation avec tkinter](#) de @Dan737 ou ce [tutoriel](#) de @pascal.ortiz pour en apprendre plus sur *tkinter*.

PyGtk et *tkinter* ne sont pas les seules bibliothèques pour écrire des programmes *GUI*, on trouve par exemple [PyQt](#) ou [wxPython](#).

IX.3.6. Pygame

Pygame est aussi une bibliothèque dédiée à écrire des interfaces graphiques, mais pas des programmes fenêtrés. Elle est plutôt dédiée aux interfaces «imagées», comme des jeux vidéo.

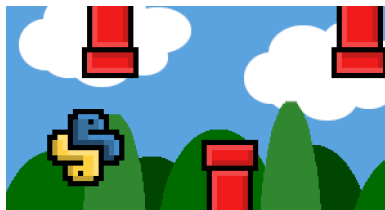


FIGURE IX.3.2. – *Flappy Python*, un jeu-vidéo fictif que l'on pourrait réaliser avec *Pygame*.

Elle s'installe par la commande `pip install pygame` dans l'environnement courant.

Elle met ensuite à disposition différents éléments graphiques pour créer des fenêtres et dessiner à l'écran.

```

1 import pygame
2
3 pygame.init()
4

```

```
5 screen = pygame.display.set_mode((800, 600))
6 running = True
7
8 screen.fill((255, 255, 255))
9 pygame.draw.rect(screen, (0, 0, 255), (200, 200, 400, 200))
10
11 while running:
12     pygame.display.update()
13
14     for event in pygame.event.get():
15         if event.type == pygame.QUIT:
16             running = False
17
18 pygame.quit()
```

Pour apprendre à utiliser *Pygame*, je vous renvoie à [ce tutoriel](#) de @SuperFola sur Zeste de Savoir et [à la documentation](#) de la bibliothèque.

Une fois encore, *Pygame* n'est pas seule dans son domaine, vous trouverez ainsi [pyglet](#) (abstraction autour d'*OpenGL*, plus bas-niveau), [Arcade](#) (construite autour de *pyglet*) ou [PySFML](#) (qui ne semble plus maintenue).

IX.4. Tests

Introduction

Je vous ai plusieurs fois parlé des tests au long de ce cours, mais nous n'avons pas vraiment de bonne manière de les lancer. Voyons alors ce que propose Python pour ça.

IX.4.1. Pytest

Pytest est une bibliothèque tierce fréquemment utilisée pour l'écriture de tests en Python, par la simplicité avec laquelle elle permet de décrire les cas de tests.

Premièrement vous pouvez installer *Pytest* avec la commande `pip install pytest`.

Celle-ci installe l'utilitaire `pytest` dans l'environnement courant.

Ensuite, il suffit d'utiliser la commande `pytest`, seule ou accompagnée de fichiers ou répertoires en arguments (par défaut il explorera le répertoire courant). *Pytest* se charge d'identifier les fichiers de tests, qui sont les fichiers Python préfixés de `test_`.

À l'intérieur de ces fichiers, les fonctions avec ce même préfixe sont identifiées comme des fonctions de tests.

Ainsi, les modules de tests que vous nous avons écrits précédemment, contenant des fonctions de tests formées d'assertions, sont déjà compatibles avec *Pytest*.

```
1 from operations import addition, soustraction
2
3
4 def test_addition():
5     assert addition(3, 5) == 8
6     assert addition(1, 0) == 1
7     assert addition(5, -8) == -3
8
9
10 def test_soustraction():
11     assert soustraction(8, 5) == 3
12     assert soustraction(5, 8) == -3
13     assert soustraction(1, 0) == 1
14     assert soustraction(3, -5) == 8
```

Listing 86 – `test_operations.py`

```
1 % pytest test_operations.py
2 ===== test session starts
   =====
```

```

3 platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0,
  pluggy-1.0.0
4 rootdir: /home/antoine
5 collected 2 items
6
7
8 test_operations.py ..
  [100%]
9
10 ===== 2 passed in 0.01s
  =====

```

Tout se passe bien, nos fonctions valident les tests ! En cas d'erreur, *Pytest* s'arrête à la première assertion fautive de la fonction et affiche un rapport explicite du problème.

```

1 ===== test session starts
  =====
2 platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0,
  pluggy-1.0.0
3 rootdir: /home/antoine
4 collected 2 items
5
6
7 test_operations.py F.
  [100%]
8
9 ===== FAILURES
  =====
10 ----- test_addition
  -----
11
12 def test_addition():
13     assert addition(3, 5) == 8
14     assert addition(1, 0) == 1
15 >    assert addition(5, -8) == 3
16 E    assert -3 == 3
17 E        + where -3 = addition(5, -8)
18
19 test_operations.py:7: AssertionError
20 ===== short test summary info
  =====
21 FAILED test_operations.py::test_addition - assert -3 == 3
22 ===== 1 failed, 1 passed in 0.02s
  =====

```

Pytest permet d'aller plus loin que ça, et fournit des outils pour paramétrer facilement nos tests (générer différentes valeurs en entrée), abstraire les entrées et sorties standards (pour tester des fonctions qui utiliseraient `print` ou `input`) et bien d'autres encore que vous découvrirez [dans sa documentation](#) ↗.

IX.4.2. Unittest

Unittest est le module de la bibliothèque standard dédié à l'écriture de tests. Je ne vous en ai pas parlé jusqu'ici parce que celui-ci nécessite l'écriture de classes, qui ne sont abordées que dans le cours sur [la programmation orientée objet en Python](#) ¹.

On peut en apprendre plus [sur la page de documentation du module](#) ² et on découvre notamment quelle structure respecter pour écrire une suite de tests.

```

1 import unittest
2
3 from operations import addition, soustraction
4
5
6 class TestOperations(unittest.TestCase):
7     def test_addition(self):
8         self.assertEqual(addition(3, 5), 8)
9         self.assertEqual(addition(1, 0), 1)
10        self.assertEqual(addition(5, -8), -3)
11
12    def test_soustraction(self):
13        self.assertEqual(soustraction(8, 5), 3)
14        self.assertEqual(soustraction(5, 8), -3)
15        self.assertEqual(soustraction(1, 0), 1)
16        self.assertEqual(soustraction(3, -5), 8)

```

Listing 87 – `test_operations.py`

Il faut ainsi écrire une classe `TestFooBar`¹ que l'on indique comme étant un cas de test (via `unittest.TestCase` entre parenthèses, qui signifie que notre classe dérive de `TestCase`) à l'intérieur de laquelle on place nos fonctions de tests.

Ces fonctions possèdent un paramètre spécial `self` qui sera fourni automatiquement. Cet objet `self` possède différentes méthodes, notamment `assertEqual` pour vérifier que les deux arguments sont égaux, `assertTrue` qui revient à faire une assertion et `assertFalse` pour l'inverse (vérifier qu'une expression est fausse).

On peut exécuter un fichier de tests à l'aide de la commande `python -m unittest`.

```

1 % python -m unittest test_operations.py
2 ..
3 -----
4
5     Ran 2 tests in 0.000s
6
7     OK

```

En cas d'erreur(s), celles-ci sont aussi signalées par le programme.

1. Il est coutume d'utiliser un style *CamelCase*, où les différents mots qui forment le nom sont écrits avec une majuscule et ne sont pas séparés d'*underscores*.

```
1 % python -m unittest test_operations.py
2 F.
3 =====
4 FAIL: test_addition (test_operations.TestOperations)
5 -----
6 Traceback (most recent call last):
7   File "/home/antoine/test_operations.py", line 10, in
8     test_addition
9     self.assertEqual(addition(5, -8), 3)
10  AssertionError: -3 != 3
11 -----
12 Ran 2 tests in 0.000s
13
14 FAILED (failures=1)
```

IX.5. Outils

Introduction

L'écosystème Python est peuplé de nombreux outils pour vous aider à écrire et maintenir votre code.

IX.5.1. Linters

Un *linter* est un programme qui permet de vérifier le style des fichiers de code, et notamment qu'ils respectent [les règles énoncées par la PEP8](#) .

La PEP8 est à l'origine une description du style que doivent adopter les développements au sein de Python lui-même, qui s'est popularisée et est maintenant considérée comme un standard pour tous les projets Python.

Il ne faut cependant pas la voir comme un énoncé de règles strictes, comme dirait un célèbre pirate «c'est plus un guide qu'un véritable règlement».

Pour en savoir plus sur les règles de style d'un code Python en général, je vous invite à consulter mon article dédié aux [secrets d'un code pythonique](#) .

IX.5.1.1. Flake8

Flake8 est donc un outil en ligne de commande permettant de vérifier la conformité avec la PEP8, relevant toutes les *infractions* trouvées dans les fichiers de code (mauvais espacements, lignes trop longues, etc.).

On installe l'outil par la commande `pip install flake8`, puis il s'utilise via `flake8`, optionnellement suivi de répertoires ou de fichiers à explorer (par défaut il explorera tout le répertoire courant).

Flake8 est configurable, et permet d'activer ou désactiver certaines règles de style, je vous invite pour cela à consulter [sa page de documentation](#) .

IX.5.1.2. Pylint

Pylint est un outil qui va plus loin que *Flake8*. Il ne se contente pas de relever les fautes de style, mais cherche aussi à identifier de potentiels erreurs et problèmes de conception.

L'installation `pip install pylint` fournit un utilitaire `pylint`, que l'on appelle en lui donnant les fichiers à vérifier en arguments.

Il est lui aussi hautement configurable, et je vous renvoie pour cela [à sa documentation](#) .

IX.5.1.3. Black

Black est un outil relativement récent dont l'objectif est d'unifier les règles de style Python et éviter les querelles de chapelles : il est prévu pour ne pas être configurable et donc appliquer le même style partout.

Après une installation via `pip install black`, on utilise la commande `black` en lui fournissant des fichiers ou répertoires à explorer, que *Black* se chargera de réécrire selon son style.

[La page de documentation](#) du projet vous renseignera davantage sur ses fonctionnalités.

IX.5.1.4. isort

isort est un outil d'un autre genre, qui s'occupe de l'ordonnancement des lignes d'import. On peut lui spécifier une configuration et il se chargera de réordonner de façon logique les imports : d'abord la bibliothèque standard, puis les modules tiers, puis le paquet courant, etc.

Il s'installe en tapant `pip install isort` puis est disponible par la commande `isort` qui prend optionnellement des fichiers ou répertoires en arguments (s'applique à tout le répertoire par défaut).

Pour plus d'informations, rendez-vous [sur la documentation d'*isort*](#).

IX.5.2. mypy

mypy est un outil d'analyse statique, qui permet de s'assurer du bon comportement d'un programme.

Sans exécuter le code, *mypy* va *simplement* l'analyser pour regarder si les opérations qui sont faites sur les données sont cohérentes (via les annotations de types) et ainsi éviter un certain nombre de bugs (par exemple des `ValueError` à l'exécution car un cas aurait été oublié).

Les annotations de types permettent de restreindre l'ensemble de définition des fonctions (via les types de valeurs autorisées) et ainsi mettre en évidence les cas qui ne s'y conforment pas : par exemple si une fonction attend un argument `int` et qu'on l'appelle avec une valeur issue d'une autre fonction qui peut renvoyer un `int` ou `None`, cela soulève un problème car le cas de `None` n'est pas correctement géré.

On installe *mypy* avec la commande `pip install mypy`, puis on l'exécute en lui fournissant les fichiers à tester : `mypy fichier.py`. Il s'occupera alors d'analyser le code et de reporter les erreurs qu'il y trouvera.

La correction des erreurs renvoyées est à votre discrétion, vous pouvez choisir de les ignorer si vous savez qu'elles n'ont pas de chance de produire de bug à l'exécution, mais il est alors préférable d'annoter le programme en conséquence pour éviter qu'elles soient relevées.

IX.6. Ressources

Introduction

Avant de terminer, j'aimerais vous donner quelques ressources pour continuer à vous documenter et apprendre le Python.

IX.6.1. Liens utiles

Pour commencer, quelques liens à garder en marque-pages concernant Python :

- [Documentation de Python](#) (traduite en français en grande partie)
- [PyPI](#), index des paquets Python
- [Site de l'AFPy](#), association francophone Python (événements, offres d'emploi)

IX.6.2. Cours

Ensuite j'aimerais vous donner les liens de différents cours (majoritairement francophones) pour compléter votre apprentissage du Python.

IX.6.2.1. Sur Zeste de Savoir



- [Les slices en Python](#) de @pascal.ortiz
- [Les secrets d'un code pythonique](#)
- [La programmation orientée objet en Python](#)
- [Pygame pour les zesteurs](#) de @SuperFola
- [À la découverte de turtle](#) de @Smokiev
- [Programmation avec tkinter](#) de @Dan737
- [Des interfaces graphiques en Python et GTK](#) de @Wizix
- [Variables, scopes et closures en Python](#)
- [La puissance cachée des coroutines](#) de @nohar
- [Découvrons la programmation asynchrone en Python](#) de @nohar
- [Introduction aux graphiques en Python avec matplotlib.pyplot](#) de @Karnaj
- [Des bases de données en Python avec sqlite3](#) de @Smokiev
- [MicroPython: Python pour les microcontrôleurs](#) de @Aabu

IX.6.2.2. Ailleurs sur le web

- [Apprendre à programmer avec Python 3](#) de Gérard Swinnen, livre francophone de référence pour l'apprentissage du Python
- [Realpython](#), une collection de tutoriels (en anglais) sur des sujets variés de Python
- [Archives des articles de sametmax](#), un ancien blog aux contenus parfois osés, mais avec des articles de qualité sur Python

IX.6.3. Exercices




Je voudrais ajouter quelques sites d'exercices pour vous entraîner avec Python :

- [Hackinscience](#) , une plateforme d'exercices dédiés au Python
- [Exercism](#) , une plateforme plus générale






IX.6.4. Discussions

Pour trouver de l'aide en cas de problème(s).

IX.6.4.1. Forums








- Forum programmation sur Zeste de Savoir 
- Forum de l'AFPy 
- Forum Python sur OpenClassrooms 

IX.6.4.2. Salons de discussions

- Le salon de discussion [#python](#)  (anglophone) sur le réseau IRC Liberachat ([irc.libera.chat](#))
 - et [#python-fr](#)  son équivalent francophone (<https://web.libera.chat/#python-fr> ) , pour toutes questions et discussions relatives au langage
- [Serveur Discord non-officiel de Zeste de Savoir](#) , très général mais avec avec un salon dédié à l'informatique
- [Serveur Discord de l'AFPy](#) 

IX.6.5. Conférences

Parce qu'on peut découvrir tout un tas de choses lors de conférences et qu'on trouve généralement les vidéos en ligne ensuite.

- La [PyConFr](#) , conférence annuelle francophone et gratuite sur Python, dont les vidéos sont mises en ligne sur [pyvideo.org](#) 
- L'[EuroPython](#) , conférence annuelle à l'échelle européenne dont on retrouve aussi les vidéos sur [pyvideo.org](#) 
- Plus généralement toutes les [PyCon](#)  qui peuvent avoir lieu tout autour du monde
- La conférence [PyParis](#)  qui a lieu à Paris (en anglais)
- L'événement [dotPy](#)  (idem)

Conclusion

Conclusion

Après la lecture de ce cours, ta quête du Python est loin d'être terminée. Tu connais maintenant les fondamentaux du langage, mais il y a encore tant à voir !

Je t'invite à continuer ton apprentissage en te dirigeant vers [la programmation orientée objet en Python](#) , pour découvrir comment mettre en place tes propres types de données.

Je te conseille aussi de jeter un œil aux [secrets d'un code pythonique](#) afin de bien t'imprégner de la philosophie Python.

De manière plus générale, regarde du côté des [ressources](#) données en annexe pour voir quels sujets pourraient t'intéresser, il y est question d'interfaces graphique, de programmation asynchrone ou encore de bases de données.

Un zeste de Python... Python in Zest'

C'est un travail de [plus de 4 ans](#) de réflexion et d'écriture qui s'achève avec ce tutoriel.

Je tiens à remercier toutes les personnes qui m'ont aidé et soutenu pendant cette période, notamment les relecteurs et relectrices de la bêta ainsi que l'équipe de validation (un grand merci à @artrags pour le temps qu'il y a consacré).

Je rappelle enfin que ce cours est diffusé sous licence *Creative Commons Attribution-ShareAlike 4.0* et que toute contribution est bienvenue. Les sources sont disponibles à l'adresse suivante: https://github.com/entwanne/cours_python_debutant .

Liste des abréviations

AFPy Association Francophone Python. 24

BDFL Benevolent Dictator for Life. 24

PyCon Python Conference. 24