cours-python-initiation

September 4, 2022

Les bases de Python Python est un langage de programmation sous licence libre créé en 1991 par Guido van Rossum. Il présente les caractéristiques suivantes :

- interprété : un programme Python ne nécessite pas de phase de compilation explicite pour s'exécuter, son code source est directement interprété.
- multi-paradigme : Python opte pour une approche objet, mais qu'il n'est pas obligatoire d'exploiter pour créer des programmes : on trouve par exemple de nombreux scripts Python écrits dans une approche impérative. Des primitives sont également présentes pour faire de la programmation fonctionnelle avec Python.
- à typage fort et dynamique
- de haut niveau : la manipulation mémoire ou des instructions processeur est laissée à l'interpréteur, pour que l'on n'ait qu'à se concentrer sur le fonctionnement des programmes que l'on écrit.
- multi-plateformes : des interpréteurs Python existent pour la quasi-totalité des systèmes d'exploitation existants (GNU/Linux, UNIX et OSX, MS Windows…), et un même code source Python peut le plus souvent s'exécuter quelle que soit la plateforme sans modification.

Python est utilisé dans les domaines suivants :

- l'automatisation de tâches système : sa portabilité et facilité d'utilisation en fait un excellent langage pour l'automatisation et le scripting dans l'administration système
- le développement de prototypes : permettant d'écrire rapidement des programmes complexes, il permet de valider la réalisabilité technique d'un projet qui sera ensuite réécrit dans un langage plus bas niveau
- le développement web : l'écosystème Python est aujourd'hui largement assez mature pour permettre le développement rapide et efficace de programmes complets et complexes
- le développement d'interfaces graphiques : la plupart des bibliothèques d'interface (comme Qt ou GTK par exemple) disposent de librairies Python pour créer des programmes graphiques
- la recherche et la programmation scientifique : des bibliothèques (souvent écrites en C pour plus de performances) très matures existent pour travailler sur de la manipulation et visualisation de données
- le développement de greffons (*plugins*) pour de nombreux logiciels (QGIS, Blender, GIMP…)

Parmi les points forts de Python, pouvant peser dans le choix de son adoption, on trouve :

- la facilité d'apprentissage et d'écriture : le langage est basé sur une syntaxe particulièrement simple et expressive, motivée par le fait qu'on lit son code beaucoup plus souvent qu'on ne l'écrit
- la maturité de l'écosystème et des processus d'évolution : ces derniers sont bien définis et publics, soumis au vote au sein d'une fondation, dont le seul but est le soutien au langage Python

- la bibliothèque standard (*stdlib*) fournie avec le langage, particulièrement conséquente, qui permet à elle seule de couvrir une grande partie des besoins que l'on retrouve en programmation

0.1 Python2 et Python3

La version 3.0 de Python est sortie en 2008. Elle a apporté des modifications importantes dans la syntaxe Python cassant la rétrocompatibilité : il est possible que du code écrit pour Python2 ne soit pas reconnu par Python3. En raison de cela, et parce que Python3 n'était pas assez mature à ce moment, une grande partie de la communauté a décidé de rester sur la branche 2.x de Python, et a continué à développer l'écosystème en adéquation. Malgré les nouvelles versions de la branche 3.x, et le fait que cette dernière soit aujourd'hui parfaitement mature, et supérieure en de nombreux points à son ainée, la migration a été longue, et s'est faite avec de nombreuses réticences. Python2 a continué d'être développé en parallèle, avec une version 2.7 sortie en 2010, et dont la date de fin de support a été avancée jusqu'en 2020, date à laquelle la fondation Python estime que l'intégralité de la communauté aura finalement adopté Python3.

0.2 Installer Python

Python étant très utilisé pour écrire des outils systèmes, il est déjà présent sur de nombreux systèmes d'exploitation. Néanmoins, il est également possible d'installer une version spécifique :

GNU/Linux ou UNIX (OSX ou *BSD)

Vérifiez d'abord si Python est présent sur votre système, avec la commande suivante :

```
$ python --version
Python 2.7.15
```

Si vous avez un retour similaire, c'est que Python est installé. Sinon, vous obtiendrez un message comme `command not found: python`. Il est probable que la version *par défaut* de Python soit une version 2.7, et que vous souhaitiez travailler sur la dernière version de la branche 3.x. Elle est peut-être tout de même disponible :

```
$ python3 --version
Python 3.6.5
```

Si c'est le cas, vous n'avez rien à installer, et vous pouvez dès à présent utiliser Python. Faites néanmoins attention à bien choisir `python3` pour exécuter vos programmes.

Si Python n'est pas installé, ou pas dans la version que vous souhaitez, vous devriez pouvoir l'installer depuis le gestionnaire de paquets de votre distribution. Par exemple, sous un système Debian (également valable pour Ubuntu et ses dérivés), tapez ceci en tant que `root` :

```
# apt install python3 python3-dev
```

Après confirmation, Python ainsi que ses dépendances s'installent. L'installation supplémentaire de paquets de développement (`python3-dev`) sera nécessaire pour installer certaines bibliothèques.

0.2.1 MS Windows

Rendez-vous sur https://python.org et suivez les instructions pour télécharger la dernière version de Python disponible. Ensuite, exécutez l'installeur téléchargé, suivez les instructions, et n'oubliez

pas de cocher la case « Ajouter Python au PATH ».

Un nouvel élément « Python » s'ajoutera alors dans votre menu démarrer, vous permettant de lancer un invite de commandes Python. Depuis un terminal, vous pourrez également le faire en tapant `python`.

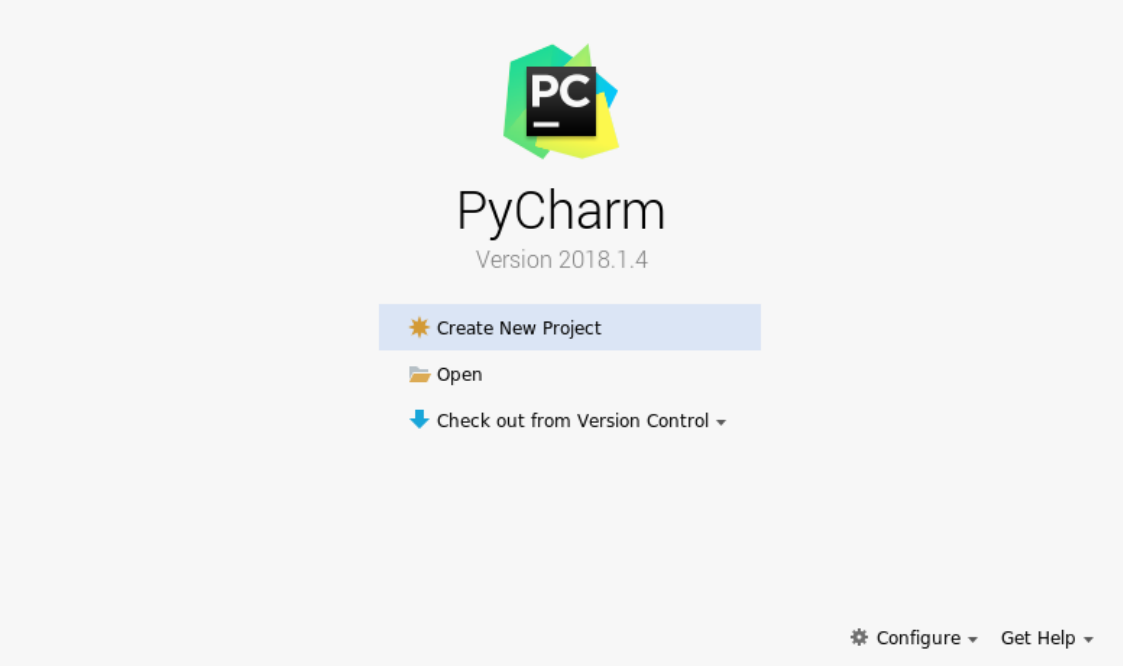
0.3 Installation d'un éditeur de texte

Comme pour tout langage de programmation, vous aurez besoin d'un éditeur de texte (à ne pas confondre avec un *traitement de texte*, comme LibreOffice, dont le but est de mettre en forme un document). Tous conviennent, et vous pouvez choisir celui qui vous correspond le plus. Si vous n'avez pas de préférence, nous conseillons le très bon environnement de développement **PyCharm**, dérivé de JetBeans spécialisé dans le langage Python, et qui dispose d'une version communautaire libre et gratuite, la **PyCharm Community Edition**.

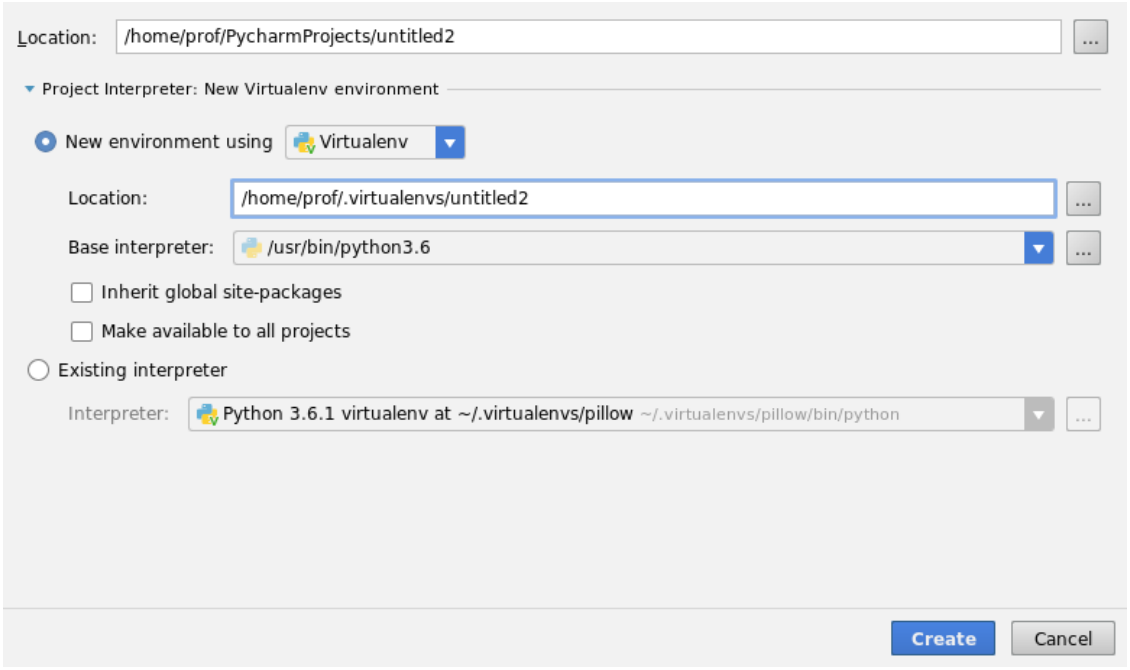
Si votre système d'exploitation dispose d'un gestionnaire de paquets, vous pouvez sans doute installer PyCharm par ce biais. Dans tous les cas, vous trouverez des instructions vous correspondant sur la page https://www.jetbrains.com/pycharm/download

0.4 Mise en pratique

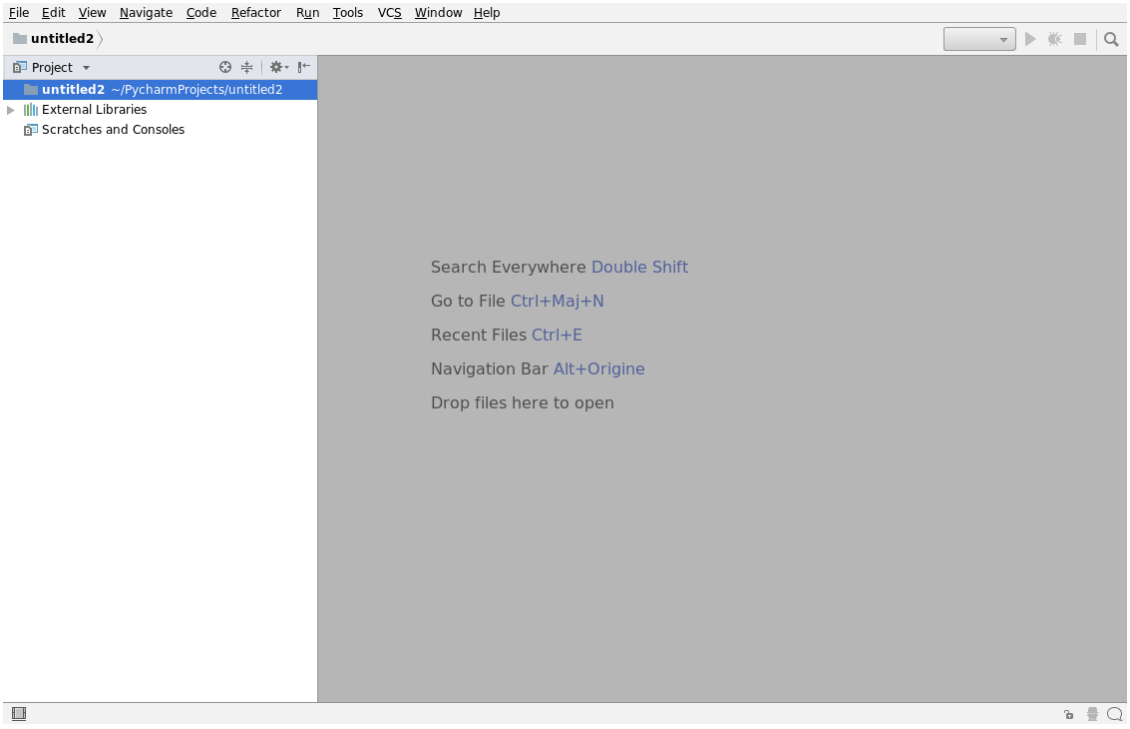
Nous allons utiliser PyCharm pour écrire un premier code Python étape par étape. Commencez par lancer l'éditeur :



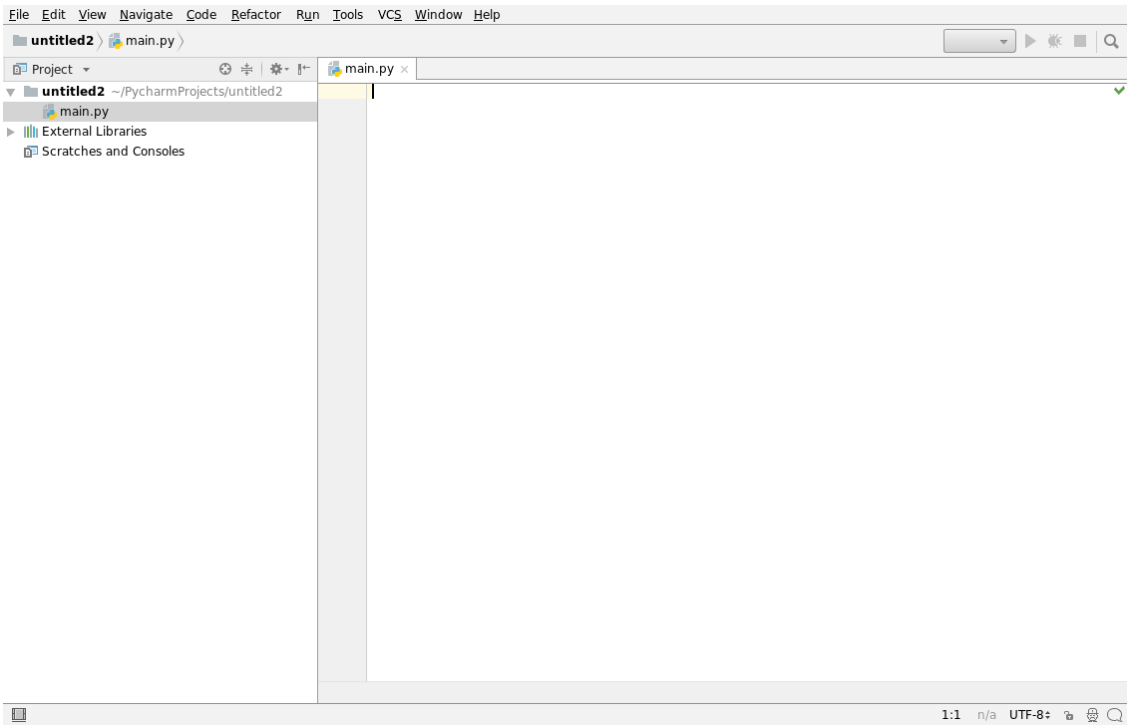
Cliquez sur « Create new project »



En cliquant sur la petite flèche à côté de « Project nterpreter », vous pourrez choisir la version de Python correspondant à ce programme. Assurez-vous d'utiliser Python3. Vous arrivez sur cette fenêtre :



La liste de gauche montre les fichiers de votre projet, sous le dossier principal de votre projet (sur ces captures, `Untitled2`). Faites un clic droit, puis sélectionnez « New > Python file », et entrez `main` comme nom. Le fichier est créé, et s'ouvre immédiatement :

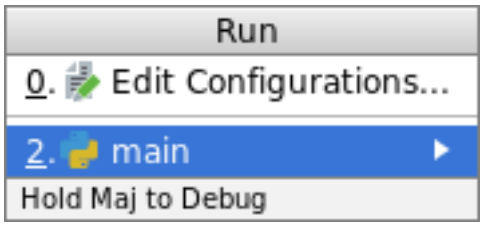


Entrez alors dans le panneau principal le code suivant :

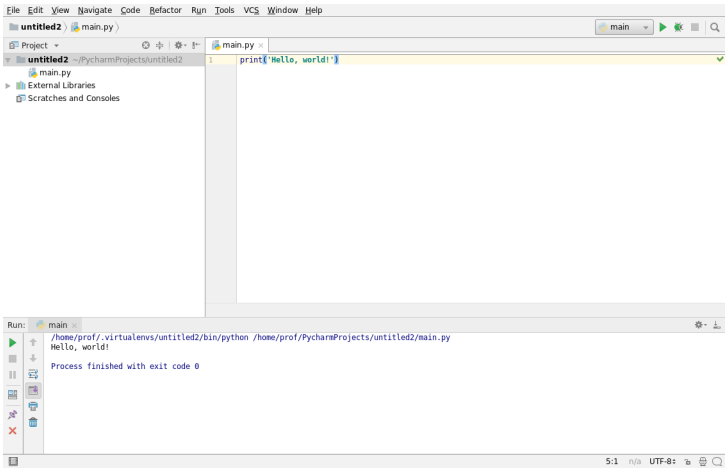
```
[1]: print('Hello, world!')
```

```
Hello, world!
```

Exécutez ensuite la commande « Run > Run » depuis le menu, ou son raccourci (`Alt+Maj+F10`). PyCharm vous demandera alors de sélectionner le *point d'entrée* de votre programme, c'est à dire le fichier à exécuter pour démarrer le projet. Nous n'en avons qu'un, donc choisissez `main`.



Votre programme s'exécute alors, sans erreur. Son retour est affiché dans un nouveau panneau, en bas de la fenêtre.



Les prochaines exécutions de votre programme ne vous demanderont plus le point d'entrée, et lanceront directement `main.py`. Le raccourci sera alors `Maj+F10`.

Maintenant que nous avons tout pour exécuter du Python, lançons-nous directement dans le sujet.

1 Les variables

Un programme se compose d'*instructions*, qui peuvent être vues comme des ordres à donner au processeur : « affiche *ça* », « ouvre *ce fichier* », etc. Ces instructions sont exécutées de façon linéaire, de la première à la dernière. Pour faire le lien entre ces instructions, de sorte à ce qu'elles puissent partager des informations, on utilise des **variables**. Il s'agit d'informations stockées en mémoire vive par le programme, que l'on peut lire, modifier et transmettre à d'autres instructions.

Une variable est, techniquement, une petite zone de mémoire qui, lorsqu'on la crée, est réservée à l'exécution actuelle de notre programme : une variable est limitée au cycle de vie du programme, et est détruite à l'extinction du programme qui l'a créé. Pour faciliter le développement, on donne un nom à cette zone de mémoire, pour pouvoir la référencer plus tard. Voici un premier exemple, permettant d'enregistrer en mémoire la valeur que produit l'expression `1 + 1`

[2]:
```
resultat = 1 + 1
```

Nous avons créé une variable, nommée `resultat`, qui contient, non pas `1 + 1`, mais `2`. En effet, la partie à droite du signe égal est une **expression**. Une expression peut faire appel à des opérateurs, des fonctions, et produit fatalement une **valeur**. C'est cette valeur qui est stockée dans sa variable, et une fois cette ligne exécutée, il n'est pas possible de savoir si le `2` stocké dans `resultat` provient de l'expression `2`, `1 + 1`, `2 / 2 + (0.5 * 2)` ou `int('2')`, par exemple.

Une fois une variable créée, on peut l'utiliser dans n'importe quelle expression :

[3]:
```
print(resultat)  # on affiche directement la valeur
print(resultat + 1)
variable2 = resultat + 1  # on crée une nouvelle variable dont la valeur est␣
↪calculée à partir de resultat
print(variable2)
```

```
2
3
3
```

Nous avons créé une seconde variable, nommée `variable2`. Dès sa création (on parle d'**initialisation**), elle est utilisable par le programme, ce que l'on fait dans la foulée. Bien qu'initialisée à partir de la valeur de `resultat`, les deux variables ne partagent aucun lien, et une modification de `resultat` n'entraîne pas de modification de `variable2`. C'est parce qu'au moment où on a initialisé `variable2`, la valeur de l'expression `resultat + 1` a été calculée (en prenant la valeur de `resultat` à ce moment), et c'est un `3` qui a été stocké dans `variable2`.

[4]:
```
resultat = 5
print(resultat)
print(variable2)
```

```
5
3
```

Nous voyons qu'il est possible de **modifier** la valeur d'une variable (d'où le nom de *variable*, qui peut varier). Il s'agit en fait de stocker une nouvelle valeur, comme on l'avait fait initialement. La valeur `5` a remplacé la valeur `2` en mémoire, et à partir de ce moment, lorsque l'on fera appel à `resultat`, c'est `5` qui ressortira, à moins que l'on modifie encore sa valeur.

1.1 L'initialisation en détail

Nous avons déjà vu la syntaxe de l'initialisation de variable :

[5]:
```
nom_de_la_variable = 'valeur_de_la_variable'
```

Elle se compose **toujours** du nom de la variable à *gauche* du signe égal, puis du signe égal, suivi d'une *expression*, qui peut être, par exemple, une valeur littérale (comme ici), le résultat d'une opération ou d'un appel de fonction…

Un nom de variable est libre, à quelques règles près :

- il ne *peut* contenir *que* des lettres, minuscules ou majuscules, des traits de soulignement (`_`) et des chiffres
- il ne *peut pas* débuter par un chiffre
- il ne *peut pas* être un mot-clé réservé en Python, comme `if`, `class`, `def`, etc. Notez que `print` par exemple n'est pas un mot-clé réservé, mais une simple fonction. Ce n'est pas pour autant que vous *devriez* utiliser `print` comme un nom de variable
- il *devrait* être écrit en minuscules uniquement, avec des traits de soulignement entre chaque mot, si le nom en contient plusieurs (exemple : `nom_de_la_variable`)
- il ne *devrait pas* prendre le nom d'une variable ou fonction existante dans python ou dans votre programme : cela ne serait pas une erreur stricte, mais vous remplaceriez alors la valeur existante ou la fonction par votre nouvelle valeur, et c'est très rarement souhaitable.
- il *devrait* avoir un nom concis et indiquant clairement ce qu'il contient. Par exemple : `jours_par_semaine = 7`

1.2 Les types

Cela n'est pas fondamentalement lié au concept de variables, mais plutôt de valeurs : une valeur possède nécessairement un type. Les types de base sont :

1.2.1 Nombres entiers

Ce type représente un nombre entier (`int`), positif ou négatif, de n'importe quelle ampleur (un entier peut contenir de très grandes valeurs).

1.2.2 Nombers à virgule flottante

Ce type est également appelé nombre flottant ou simplement flottant (`float`). Il est important de noter qu'il ne s'agit pas d'un nombre décimal, même s'il se comporte *presque* de la même façon. Cela se voit souvent quand on effectue des opérations sur des flottants, par exemple :

[6]:
```
print(0.7 + 1.4)
```

```
2.0999999999999996
```

Le résultat n'est pas `2.1` comme il le devrait, mais il s'en approche. C'est dû à la façon dont le processeur stocke les nombres flottants. Prenez ça en compte, et évitez notamment les nomres flottants pour tout calcul sensible, comme des données monétaires. Pour cet exemple, il est notamment souhaitable de manipuler des nombres en centimes.

1.2.3 Chaînes de caractères

Ce type représente un ensemble de caractères (`str`) (lettres, mais également chiffres, espaces, ponctuation, ou tout caractère imprimable ou non) stocké au sein d'une même valeur (nous verrons par la suite les types composites, qui peuvent se comporter de façon similaire). Lorsque vous demandez à l'utilisateur de taper quelque chose, même si c'est un nombre, vous recevrez *toujours* une chaîne de caractères, car Python ne peut pas assumer pour le type que vous attendez. Si vous voulez écrire des chaînes de caractères littérales, vous devez les entourer d'apostrophes (`'chaine'`) ou de guillemets (`"chaine"`).

1.2.4 Valeurs spéciales

Elles sont au nombre de 3 : vrai (`True`), faux (`False`), et nul (`None`). Elles s'écrivent directement dans le code .

1.2.5 Typage fort

Python est un langage dit *fortement typé*. C'est à dire que les types sont conçus pour respecter strictement les valeurs qu'ils représentent, et interdisent notamment certaines opérations entre types différents. Par exemple, il est possible d'additionner deux entiers, ou un entier et un flottant, voire même entre deux chaînes de caractères (`'hello' + 'world'` donne `'helloworld'`). Mais il est impossible d'additionner un nombre (qu'il soit flottant ou entier) et une chaîne de caractères.

[7]:
```
print(1 + 1)
print(0.2 + 0.8)
```

```
print('hello ' + 'world')  # notez que la première chaîne contient une espace : le "+" n'en rajoute pas de lui-même
```

```
2
1.0
hello world
```

[8]:
```
print('hello' + 1)  # ceci ne fonctionne pas
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-b213fc2b97b1> in <module>()
----> 1 print('hello' + 1)  # ceci ne fonctionne pas

TypeError: Can't convert 'int' object to str implicitly
```

Il est néanmoins possible de convertir une valeur d'un type à un autre, en appelant la fonction du type souhaité sur cette valeur. Pour tester les types, on peut utiliser la fonction `type()` :

[9]:
```
print(type('hello'))
print(type(1.1))
```

```
<class 'str'>
<class 'float'>
```

Pour convertir un nombre en chaîne de caractères, on fera donc appel à la fonction `str()`, comme ceci :

[10]:
```
nombre_converti = str(12)
print(type(nombre_converti))
```

```
<class 'str'>
```

Maintenant que l'on a une chaîne, on peut utiliser le symbole `+` pour y coller d'autres chaînes, par exemple (c'est ce qu'on appelle la *concaténation* :

[11]:
```
print(nombre_converti + '3')
```

```
123
```

[12]:
```
print(nombre_converti + 3)  # cela ne fonctionnera pas, on ne peut pas additionner une chaîne et un nombre
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-08700d66595d> in <module>()
----> 1 print(nombre_converti + 3)  # cela ne fonctionnera pas, on ne peut pas additionner une chaîne et un nombre
```

```
TypeError: Can't convert 'int' object to str implicitly
```

1.3 Les types composites

Outre les types vus précédemment, il existe des types composés de plusieurs valeurs : on les appelle les types composites.

1.3.1 Listes

Une liste est un type contenant un nombre variable de valeurs de types mixtes (une liste peut contenir à la fois des entiers, des chaînes et d'autres listes). Elle est ordonnée, et *mutable* (cette notion sera vue par la suite).

[ ]:
```
ma_liste = ['a', 'b', 3, True]
print(ma_liste)
```

La définition littérale d'une liste se fait avec les crochets (`[]`) entourant les valeurs, séparées par des virgules.

Une liste étant ordonnée, ses éléments sont toujours stockés dans le même ordre, et possèdent un *index*, un compteur qui démarre à zéro et qui identifie chaque valeur. La première valeur d'une liste aura donc l'index `0`, la suivante `1`, etc.

Il est possible d'accéder à une valeur d'une liste si on en connaît l'index, en utilisant cette syntaxe :

[ ]:
```
print(ma_liste[2])  # on accède à l'index 2 de la liste (soit la troisième␣
↪valeur)
```

[ ]:
```
print(ma_liste[99])  # tenter d'accéder à un index qui n'existe pas produit une␣
↪valeur
```

On peut connaître la taille d'une liste, comme de tout type composite, par la fonction `len()` :

[ ]:
```
print(len(ma_liste))
```

Une caractéristique très intéressante d'une liste est sa *mutabilité* : c'est à dire qu'il est possible de modifier le contenu d'une liste sans devoir créer une nouvelle valeur : nous pouvons le mettre en évidence avec la fonction `id()` qui donne l'identifiant interne d'une valeur pour Python : deux valeurs possédant le même identifiant sont littéralement stockées au même endroit, et en modifier une modifie l'autre. Ça n'a pas de conséquences sur les types *non mutables*, car ils ne peuvent pas être modifiés, mais pour une liste, ça a de nombreuses conséquences :

[ ]:
```
print(id(ma_liste))
ma_liste2 = ['a', 'b', 3, True]  # ma_liste2 a les mêmes valeurs que ma_liste,␣
↪mais possède un id différent
print(id(ma_liste2))
```

La première conséquence de la mutabilité est qu'il est possible de modifier la valeur d'une liste sans avoir recours à l'opérateur `=` (qui signifie « écrase l'ancienne valeur). Il est possible d'utiliser ce

qu'on appelle des *méthodes* : des fonctions rattachées à une valeur (que celle-ci soit stockée dans une variable ou pas). On accède à une méthode par la syntaxe `<valeur>.<nom_de_la_methode>()`. Les méthodes disponibles dépendent du type de la valeur. Par exemple, la méthode `list.append()` ne se trouve que sur les listes. Elle ajoute la valeur entre parenthèses à la fin de la liste (à gauche du point). Et sans utiliser d'opérateur `=`, on modifie une variable.

[ ]:
```
ma_liste2.append('nouvelle valeur')  # nous insérons une nouvelle ligne à la␣
↪fin de la liste ma_liste2
print(ma_liste2)
print(ma_liste)  # ma_liste n'a pas été modifiée, n'ayant pas de lien avec␣
↪ma_liste2
```

[ ]:
```
print(id(ma_liste2))
```

On constate que malgré un changement de valeur (une *mutation*), `ma_liste2` possède toujours le même identifiant. Ce n'est pas le cas pour les valeurs non mutables, comme un entier ou une chaîne de caractères :

[ ]:
```
ma_chaine = 'Hello world'
print(id(ma_chaine))
ma_chaine = 'Hello Python'  # il y a eu une nouvelle assignation, nous avons␣
↪détruit l'id précédent
print(id(ma_chaine))
```

Voyons ce qui se passe si on assigne une variable contenant une liste dans une autre variable

[1]:
```
liste1 = ['a', 'b', 'c']
liste2 = liste1
print(id(liste1))
print(id(liste2))
```

```
140219749504640
140219749504640
```

Les `id` sont identiques. Ce qui se passe sur `liste1` est répercuté sur liste2. Exemple, en modifiant une valeur existante (que l'on référence par son `index`) :

[2]:
```
liste2[0] = 'A'
print(liste2)
print(liste1)
```

```
['A', 'b', 'c']
['A', 'b', 'c']
```

Bien que nous n'ayions pas modifié explicitement `liste1`, cette variable fait *référence* à la même valeur que `liste2`. Si on voulait éviter ce comportement, nous aurions pu créer une **copie** de `liste1` :

[14]:
```
liste1 = ['a', 'b', 'c']
liste2 = liste1.copy()
```

```
print(id(liste1))
print(id(liste2))
```

```
139881317106760
139881317705224
```

Les `id` sont maintenant différents, et nous pouvons traiter les 2 listes indépendamment, car, si elles ont à ce moment la même *valeur*, elles sont séparées. Nous pouvons le vérifier avec deux opérateurs : `==`, qui vérifie l'égalité de deux valeurs, et `is`, qui vérifie si deux valeurs sont *identiques* (donc ont le même identifiant). Ces deux opérateurs renvoient un *booléen* (`True` ou `False`) :

[15]:
```
print(liste1 == liste2)  # on vérifie l'égalité des valeurs, donc c'est vrai
print(liste1 is liste2)  # les deux listes ont des id différents, donc c'est␣
 ↪faux
```

```
True
False
```

Tuples Un tuple est un type composite semblable à une liste : il possède des valeurs mixtes et ordonnées. Sa différence fondamentale avec le type précédent est qu'il est **non mutable** : de la même manière qu'une chaîne de caractères, il est strictement impossible de modifier un tuple sans faire de réassignation (et donc de remplacer son `id`).

Un tuple se déclare de façon semblable à une liste, si ce n'est que les parenthèses sont utilisées à la place de crochets :

[16]:
```
mon_tuple = ('a', 'b', 'c')
print(mon_tuple)
print(mon_tuple[1])
```

```
('a', 'b', 'c')
b
```

[17]:
```
mon_tuple.append('d')  # ne fonctionne pas, un tuple ne possède pas de méthode␣
 ↪append
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-c7e3d66ad0ec> in <module>()
----> 1 mon_tuple.append('d')  # ne fonctionne pas, un tuple ne possède pas de␣
 ↪méthode append

AttributeError: 'tuple' object has no attribute 'append'
```

[18]:
```
mon_tuple[0] = 'A'  # ne fonctionne pas non plus, on ne peut pas modifier la␣
 ↪valeur d'un tuple
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-1ba37b18c1b1> in <module>()
----> 1 mon_tuple[0] = 'A'  # ne fonctionne pas non plus, on ne peut pas␣
 ↪modifier la valeur d'un tuple

TypeError: 'tuple' object does not support item assignment
```

En réalité, les parenthèses de la déclaration d'un tuple ne sont pas nécessaires, car c'est le symbole virgule qui symbolise le tuple. On met néanmoins souvent des parenthèses pour faciliter la lecture.

[19]:
```
tuple1 = (1)  # ce n'est pas un tuple, il n'y a pas de virgule
print(type(tuple1))
tuple2 = (1, )  # il y a une virgule, donc nous avons bien un tuple d'une valeur
print(type(tuple2))
print(len(tuple2))
tuple3 = 1,  # la déclaration fonctionne sans parenthèses
print(type(tuple3))
```

```
<class 'int'>
<class 'tuple'>
1
<class 'tuple'>
```

Prenons maintenant un instant pour revenir aux chaînes de caractères. Le mot « chaîne » implique une série de caractères, comme un itérable : une liste, ou… un tuple. En effet, une chaîne de caractères n'est **pas** mutable. On peut appliquer des transformations dessus, mais ça produira toujours une nouvelle chaîne. On peut alors considérer que le comportement d'un tuple et celui d'une chaîne de caractères sont assez proches.

1.3.2 Les dictionnaires

Un dictionnaire est un type extrêmement utile, retrouvé dans d'autres langages sous des noms aussi divers que « mappings », « tableaux associatifs », « objets » (bien que le concept d'objet soit applicable en python). Un dictionnaire est une collection de valeurs mixtes identifiées par des valeurs arbitraires, contrairement à la liste, qui identifie ses valeurs par des entiers croissants sur lesquels on n'a pas le contrôle. Le mot « dictionnaire » en lui-même est très parlant : on stocke des *valeurs* (*définitions*) derrière des **clés** (*mots*). Les clés d'un dictionnaire doivent être uniques, mais peuvent être de types variés : entiers, chaînes de caractères… On utilise même parfois des tuples. Un dictionnaire s'écrit avec des accolades entourant des paires clés/valeurs séparées par des virgules. Le caractère deux-points (`:`) sépare la clé de la valeur)

[20]:
```
mon_dict = {'cle1': 'valeur1', 'cle2': 3}
print(mon_dict)
```

```
{'cle2': 3, 'cle1': 'valeur1'}
```

On accède à une valeur d'un dictionnaire comme pour une liste : avec des crochets derrière le

dictionnaire, et la clé recherchée à l'intérieur. Si cette clé n'existe pas, Python renvoie une erreur.

[21]:
```
print(mon_dict['cle1'])
```

```
valeur1
```

Une particularité importante des dictionnaires est qu'ils ne sont **pas ordonnés** : on peut insérer des valeurs dans un dictionnaire, sans avoir la garantie que ces valeurs ressortiront dans le même ordre plus tard (cela ne concerne pas l'assignation clé-valeur, qui elle reste bien garantie). Ce comportement a changé en python3.7, qui rend les dictionnaires ordonnés. Voyons-le par l'exemple, en ajoutant une nouvelle valeur à notre dictionnaire. Ici, pas de `append()`, puisque le concept d'ajouter *à la fin* n'a pas de sens pour un dictionnaire. Pour créer une nouvelle valeur, il suffit d'assigner une nouvelle clé :

[22]:
```
mon_dict['cle3'] = True
mon_dict['cle0'] = True

print(mon_dict)
```

```
{'cle2': 3, 'cle0': True, 'cle3': True, 'cle1': 'valeur1'}
```

On constate bien à l'affichage que l'ordre des clés ne correspond pas du tout à l'ordre d'insertion, ni même à un quelconque tri sur les clés. Cela ne pose pas de problème dans la plupart des cas d'usage d'un dictionnaire, et en cas de besoin, la bibliothèque standard de Python propose un type `OrderedDict` qui comble ce manque des dictionnaires (alternativement, Python3.7 retient l'ordre des éléments en natif).

Lorsque l'on itère sur un dictionnaire (un exemple simple est la conversion d'un dictionnaire en liste), il est important de noter que l'itération se fait **uniquement sur les clés** du dictionnaire, et non pas les paires clés-valeurs. Des méthodes existent pour accéder soit aux valeurs soit aux couples clés-valeurs directement :

[23]:
```
print(list(mon_dict))  # on itère simplement sur le dictionnaire
print(list(mon_dict.values()))  # appeler values() renvoie la liste des valeurs
print(list(mon_dict.items()))  # avec items(), on reçoit des tuples de deux␣
↪valeurs
```

```
['cle2', 'cle0', 'cle3', 'cle1']
[3, True, True, 'valeur1']
[('cle2', 3), ('cle0', True), ('cle3', True), ('cle1', 'valeur1')]
```

La suppression d'une valeur d'un dictionnaire s'opère avec le mot-clé réservé `del`, comme pour une variable simple :

[24]:
```
del(mon_dict['cle0'])
print(mon_dict)
```

```
{'cle2': 3, 'cle3': True, 'cle1': 'valeur1'}
```

1.4 Découper un itérable

La plupart des types itérables proposent une syntaxe permettant d'obtenir une « tranche » d'une valeur. On appelle ça le *slicing*.

Par exemple, à partir d'une liste contenant les noms des jours de la semaine, il est possible d'extraire une sous-liste contenant les 3 premières valeurs, ou les 2 dernières, ou celles au milieu, ou bien créer une nouvelle liste contenant un jour sur deux à partir du 4ème.

[25]:
```
days = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
```

[26]:
```
print(days[1])  # on affiche le second jour. Ce n'est pas une liste, donc il ne␣
↪s'agit pas de slicing
```

```
mardi
```

[27]:
```
print(days[1:4])  # on découpe la chaîne pour obtenir les valeurs entre la␣
↪numéro 1 (incluse) et la 4 (non incluse)
```

```
['mardi', 'mercredi', 'jeudi']
```

[28]:
```
print(days[5:])  # on précise un index de début, mais pas de fin : on prend de␣
↪l'élément 5 au dernier
```

```
['samedi', 'dimanche']
```

[29]:
```
print(days[:3])  # on prend les éléments du début jusqu'au numéro 3 (non inclus)
```

```
['lundi', 'mardi', 'mercredi']
```

[30]:
```
print(days[1:6:2])  # on prend les éléments de 1 (inclus) à 6 (non inclus),␣
↪avec un pas de 2
```

```
['mardi', 'jeudi', 'samedi']
```

[31]:
```
print(days[::3])  # on prend tous les éléments avec un pas de 3
```

```
['lundi', 'jeudi', 'dimanche']
```

Il existe bien d'autres types au sein de Python, mais ceux-ci sont les plus importants et les plus utilisés.

2 Chaînes de caractères en Python

Les chaînes de caractères, au premier abord, se comportent en Python comme dans tout autre langage à typage dynamique.

[32]:
```
my_str = 'Bonjour'
print(my_str)
print(len(my_str))
```

```
my_name = 'le monde'
print(my_str + my_name)  # oups
print(my_str + ' ' + my_name)
```

```
Bonjour
7
Bonjourle monde
Bonjour le monde
```

Une chaîne de caractère est en réalité un **objet**, comme absolument tout en Python d'ailleurs (mais nous y reviendrons). Cela signifie qu'elle possède des *méthodes* capables de s'appliquer à l'objet. Par exemple :

[33]:
```
print(my_str.upper())  # la méthode upper() renvoie la chaîne de caractères␣
↪mise en majuscules
```

```
BONJOUR
```

[34]:
```
print(my_str.replace('jour', 'ne nuit'))  # replace() remplace le premier␣
↪argument par le second au sein de my_str
```

```
Bonne nuit
```

[35]:
```
languages = 'Python, C, Java, Rust'
print(languages.split(', '))  # split() découpe une chaîne selon la chaîne␣
↪donnée, et renvoie une liste
```

```
['Python', 'C', 'Java', 'Rust']
```

[36]:
```
languages = ['Python', 'C', 'Java', 'Rust']
print(', '.join(languages))  # à l'inverse de split(), join() relie les␣
↪éléments d'un itérable avec le caractère donné
```

```
Python, C, Java, Rust
```

Une chaîne de caractère est, comme son nom l'indique, une *chaîne*. Il s'agit donc d'une liste de caractères. Peut-on la parcourir comme une liste ?

[37]:
```
print(my_str[3])  # tentons d'accéder au 4ème caractère
```

```
j
```

[38]:
```
for letter in my_str:
    print(letter)
```

```
B
o
n
j
o
```

```
u
r
```

Mais attention, une chaîne de caractères est **immutable** : il est impossible de la modifier (on peut cependant écraser sa valeur par une autre)

[39]:
```
my_str[0] = 'b'  # erreur !
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-3666ad3d8feb> in <module>()
----> 1 my_str[0] = 'b'  # erreur !

TypeError: 'str' object does not support item assignment
```

[40]:
```
my_str = 'bonjour'  # cela fonctionne, car nous ne modifions pas la valeur␣
↪existante, nous en assignons une nouvelle
```

2.0.1 Caractères accentués

Voyons comment se comporte une chaîne possédant un accent :

[41]:
```
my_str = 'Je suis enchanté'
print(len(my_str))
```

```
16
```

Cela fonctionne comme prévu, car Python3 supporte l'Unicode par défaut, ce qui n'était pas le cas en Python2. Pour être précis, il existe deux types de chaînes de caractères en Python : `str`, qui représente une chaîne unicode, et `bytes`, qui représente une chaîne *ASCII*. Cette dernière a la particularité de représenter un caractère par un octet en mémoire, ce qui facilite le traitement de données binaires.

Une chaîne telle que définie au-dessus est donc une `str`. Pour manipuler explicitement un objet `bytes`, il y a deux façons de procéder : appeler le constructeur de `bytes` sur une chaîne, ou préfixer la notation littérale d'un `b`.

[42]:
```
my_bytes = bytes(my_str, 'utf-8')  # nous devons préciser l'encodage exact de␣
↪la chaîne à convertir
print(my_bytes)
print(len(my_bytes))
```

```
b'Je suis enchant\xc3\xa9'
17
```

[43]:
```
my_bytes = b'Je suis enchant\xc3\xa9'  # il est impossible d'écrire␣
↪littéralement des caractères non-ascii dans un bytes
print(my_bytes)
print(len(my_bytes))
```

```
b'Je suis enchant\xc3\xa9'
17
```

Il est possible de convertir un `bytes` en `str` par l'appel à la méthode `decode()` sur le premier. Pour convertir de `str` en `bytes`, c'est la méthode `encode()`.

[44]:
```
print(my_bytes.decode())
```

```
Je suis enchanté
```

[45]:
```
print(my_str.encode())
```

```
b'Je suis enchant\xc3\xa9'
```

Et Python2 ? En Python2, une chaîne de caractères était par défaut en *ASCII*. Le type `str` était donc le `bytes` d'aujourd'hui. Et le support de l'unicode passait par un type `unicode`, qui permettait d'écrire des chaînes unicode explicitement par le préfixe `u`.

Ce notebook étant en Python3, il ne contient pas d'exemples en Python2. Mais n'hésitez pas à expérimenter le fonctionnement sur vos propres notebooks !

2.1 Formatage de chaînes

Une fonctionnalité particulièrement puissante des chaînes de caractères est la possibilité de les formater. Il s'agit de créer une chaîne contenant des marqueurs, puis de remplacer ces marqueurs par des valeurs.

Il y a deux syntaxes : l'ancienne (toujours présente) basée sur le symbole `%`, et la nouvelle, basée sur la méthode `format()`. Cette dernière étant présente dès Python2.7, elle est plus qu'encouragée (sans compter le fait qu'elle permet de faire plus de choses)

[46]:
```
my_str = 'Bonjour %s'  # ici, nous spécifions un marqueur "str" avec l'ancienne␣
↪syntaxe
print(my_str % 'world')  # pour remplir le marqueur, il suffit d'utiliser␣
↪l'opérateur % sur la chaîne,
                         # suivi de la valeur à insérer
```

```
Bonjour world
```

[47]:
```
my_str = 'Bonjour %s %s'  # il est possible de placer autant de marqueurs qu'on␣
↪veut. En contrepartie, il faudra
                          # naturellement donner autant de valeurs
print(my_str % ('les', 'amis'))  # les différentes valeurs sont fournies sous␣
↪la forme d'un tuple
```

```
Bonjour les amis
```

[48]:
```
my_str = 'Bonjour {}'  # un marqueur avec la syntaxe format()
print(my_str.format('world'))
```

```
Bonjour world
```

[49]:
```
my_str = 'Bonjour {} {}'
print(my_str.format('les', 'amis'))
```

```
Bonjour les amis
```

Il est possible de spécifier quelle valeur on utilise à quel endroit, ce qui est pratique pour localiser des chaînes, dans lesquels l'ordre diffère.

[50]:
```
my_str = 'Nous sommes le {0}/{1}/{2}'
print(my_str.format('27', '09', '2016'))
```

```
Nous sommes le 27/09/2016
```

[51]:
```
my_str = 'We are {1}/{0}/{2}'
print(my_str.format('27', '09', '2016'))
```

```
We are 09/27/2016
```

Il est également possible de nommer les marqueurs. On pourra les nommer lors de l'appel à `format()` par des arguments nommés.

[52]:
```
my_str = 'Nous sommes le {jour}/{mois}/{annee}'
print(my_str.format(jour='27', mois='09', annee='2016'))
```

```
Nous sommes le 27/09/2016
```

Avec l'ancienne syntaxe, cela se faisait en passant un dictionnaire à l'opérateur `%`.

[53]:
```
my_str = 'Nous sommes le %(jour)s/%(mois)s/%(annee)s'
print(my_str % {'jour': '27', 'mois': '09', 'annee': '2016'})
```

```
Nous sommes le 27/09/2016
```

Le formatage de chaînes offre de nombreuses possibilités. Par exemple, il est possible d'afficher la valeur arrondie d'un nombre flottant

[54]:
```
from math import pi
my_str = 'La valeur approximative de  est {pi:.2f} (valeur complète : {pi})'
print(my_str.format(pi=pi))  # avec les marqueurs nommés, on peut utiliser␣
 ↪plusieurs fois le même marqueur dans la chaîne
```

```
La valeur approximative de  est 3.14 (valeur complète : 3.141592653589793)
```

Plus d'options sur le formatage de chaînes sont accessibles sur le site Pyformat

[ ]:
```

```

3 Structures de contrôle

Après la manipulation de variables, la mise en place de structures de contrôles est primordiale pour écrire un programme dynamique. Celles-ci permettent de définir des portions de code qui ne seront exécutées que dans certaines conditions, ou encore de répéter d'autres portions de code.

3.1 Blocs de code

Le point commun de toutes ces structures est le concept de *bloc de code*. Il s'agit d'un ensemble de lignes de code (au moins une) qui est *indenté* par rapport au code précédent. Un bloc de code est introduit par une ligne spéciale (dépendant du type de bloc), qui est au niveau d'indentation précédent. Ces introductions de blocs de code finissent par un caractère deux-points (`:`). Enfin, un bloc de code peut contenir d'autres blocs de code.

Exemple :

```
ceci est du code non indenté.
début du bloc:
    cette ligne fait partie du bloc
    cette ligne aussi
on est revenus au niveau d'indentation précédent, donc on ne fait plus partie du bloc.
début d'un bloc:
    cette ligne fait partie du bloc ouvert en ligne précédente
    début du sous-bloc:
        cette ligne fait partie du bloc « début du sous-bloc », lui-même faisant partie du bloc
ici, on est hors de tout bloc.
```

Techniquement, l'indentation peut être réalisée de plusieurs façons différentes : par le caractère tabulation, par quatre espaces, deux, six… Le choix est large, et Python impose seulement que l'on ne mélange pas plusieurs types d'indentation au sein d'un même fichier.

Les normes de Python veulent que l'indentation soit réalisée par quatre espaces uniquement. La plupart des éditeurs de texte supportant Python sont configurés pour respecter cette convention, même lorsque que l'on utilise la touche « tabulation ».

Voyons plus en détails les structures proposées par Python :

3.2 Structure conditionnelle

La première des structures de contrôle est la condition : elle permet de définir un bloc de code qui sera exécuté **si et seulement si** une condition est vraie. Pour commencer, penchons-nous sur ce qu'est une condition.

3.2.1 Conditions

Une condition n'est rien d'autre qu'une *expression* (opération simple ou composée produisant une valeur), qui devra être équivalente à `True` ou `False` (vrai ou faux). Ainsi, l'expression suivante peut être une condition extrêmement simple :

[55]:
```
print(True)
```

```
True
```

Cette condition est vraie. D'autres exemples de conditions peuvent être :

[56]:
```
print(True or False)

ma_variable = 'test'
print(len(ma_variable) == 4)  # notez l'utilisation de l'opérateur ==

print(ma_variable != 'test')  # cet opérateur est l'opposé de ==
```

```
True
True
False
```

Cet exemple introduit trois nouveaux opérateurs, extrêmement utiles pour les conditions :

`or`, et son confrère `and`. Il s'agit d'opérateurs booléens, renvoyant `True` ou `False` selon la valeur booléenne des opérandes. La *valeur booléenne* est la résultante de la conversion d'une valeur en `bool` :

[57]:
```
ma_variable = 1
print(type(ma_variable))
ma_variable_bool = bool(ma_variable)
print(ma_variable_bool)
print(type(ma_variable_bool))
```

```
<class 'int'>
True
<class 'bool'>
```

Cette notion de conversion implicite des valeurs dans les conditions est très commune. En réalité, la valeur `1` dans une condition équivaut à `True`, car elle sera implicitement convertie. Il est alors utile de savoir quelle est la conversion booléenne des valeurs que l'on manipule. C'est assez simple, car en dehors de quelques valeurs spécifiques, **toutes** les valeurs sont équivalentes à `True`. Les exceptions, qui valent alors `False`, sont :

- `False`
- `None`
- `0`, vaut aussi pour `0.0` ou toute valeur numérique valant zéro (attention, la chaîne de caractères `'0'` n'entre pas dans cette catégorie)
- un itérable de longueur 0 (rappel : les chaînes de caractères sont des itérables)

Une des applications de cette règle est la possibilité de mentionner une variable contenant une liste, par exemple, en guise de condition. Si cette liste est vide, la condition sera fausse. Ainsi, les deux conditions suivantes sont équivalentes :

[58]:
```
ma_liste = []
print(bool(ma_liste))  # le bool() ici est utile seulement pour démontrer la␣
↪valeur booléenne
print(len(ma_liste) != 0)
```

```
False
False
```

Mais revenons aux opérateurs introduits dans la cellule précédente. Outre `or` et `and`, nous avons l'opérateur d'égalité : `==`. Celui-ci ne doit pas être confondu avec l'opérateur d'*assignation*, qui permet de créer une variable. Comme pour les opérateurs suivants, le `==` renvoie un booléen si la valeur de gauche équivaut à la valeur de droite. Deux listes possédant les mêmes valeurs seront identiques du point de vue de cet opérateur (même si elles ont des `id` différents). L'opérateur `!=` est le « non égal », et renvoie simplement `False` dans les cas où `==` renvoie `True`.

Maintenant que nous avons démystifié ce qu'est une condition, attardons-nous sur la première structure qui l'exploite :

3.2.2 Syntaxe

Une structure conditionnelle définit un bloc de code, qui sera exécuté si une condition est vraie. Le mot-clé de cette structure est `if` (« *si* »). Il se compose de la sorte :

```
if <condition>:
    code à exécuter si la condition est vraie
```

[59]:
```
ma_variable = 'test'
if ma_variable == 'test':
    print('ma_variable vaut bien "test"')
```

```
ma_variable vaut bien "test"
```

[60]:
```
if ma_liste:
    print('ma_liste contient quelque chose')
```

[61]:
```
if not ma_liste:  # le mot-clé not inverse la condition
    print('ma_liste est vide')
```

```
ma_liste est vide
```

Si la condition d'un `if` est fausse, il est possible d'exécuter un autre bloc de code (qui ne sera exécuté que dans ce cas). Il faut alors faire suivre le `if` d'un `else` (qui ne prend pas de condition, lui), puis d'un bloc :

[62]:
```
if ma_liste:
    print('ma_liste contient quelque chose')
else:
    print('ma_liste est vide')
```

```
ma_liste est vide
```

Il est parfois nécessaire de vérifier plusieurs conditions : par exemple, si une variable a telle valeur, on exécute ça, si elle en a telle autre, on exécute cet autre bloc, etc. Le mot-clé `elif` existe, contraction de `else if`. Il fonctionne exactement comme un `if`, mais doit se mettre à la suite d'un `if`, et ne s'exécute que si la première condition est fausse. On peut mettre autant de `elif` que l'on veut sous un `if`, et la première à être vraie sera exécutée, ignorant les autres.

[63]:
```
action = 'courir'

if action == 'arreter':
    print('On arrête')
elif action == 'marcher':
    print('On marche')
elif action == 'courir':
    print('On court')
elif action == 'voler':
    print('On vole')
else:  # si aucune des conditions précédentes n'est vraie, on exécute ce bloc
    print('On ne sait pas quoi faire')
```

```
On court
```

Certains langages disposent d'une structure `switch […] case`, qui n'est pas présente en Python. Mais elle est remplacée par cette forme de `if […] elif`.

3.3 Boucles

Les deux structures suivantes permettent de répéter un bloc de code. En général, avec une modification à chaque passage, sous la forme d'une valeur différente pour une variable.

3.3.1 Boucle itérative

La boucle `for` permet de parcourir un itérable, de n'importe quel type. Pour (« *for* ») chaque valeur de cet itérable, on exécutera la boucle en stockant la valeur en question dans ce qu'on appelle une **variable d'itération**, qui sera assignée par la structure `for`. Elle s'écrit de la sorte :

```
for <variable> in <iterable>:
    faire quelque chose avec <variable>
```

Dans cet exemple, on crée une variable `<variable>` au niveau du `for`, qui vaudra coup sur coup toutes les valeurs présentes dans `<iterable>`.

[64]:
```
ma_liste = ['Freddie Mercury', 'Brian May', 'Roger Taylor', 'John Deacon']

for nom in ma_liste:
    print('nom vaut : ' + nom)
```

```
nom vaut : Freddie Mercury
nom vaut : Brian May
nom vaut : Roger Taylor
nom vaut : John Deacon
```

Notons que les différentes valeurs mises dans la variable d'itération sont bien les valeurs extraites de l'itérable, et non leurs clés ou indexes, comme dans d'autres langages. La force de la boucle `for` réside dans sa simplicité, car si on veut un comportement particulier, il faudra modifier l'itérale fourni, et non la structure `for` en elle-même. Par exemple, pour itérer simplement un nombre de fois défini, la fonction `range()` nous produit un itérable d'entiers, de 0 à `n` par défaut :

[65]:
```
for i in range(10):  # range(10) nous produit une pseudo-liste des entiers de 0␣
↪à 9
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[66]:
```
for i in range(5, 8):  # range() peut prendre deux arguments, qui seront alors␣
↪la valeur initiale et la borne haute
    print(i)
```

```
5
6
7
```

[67]:
```
for i in range(3, 100, 5):  # range() peut même prendre un troisième argument,␣
↪qui sera alors le pas
    print(i)
```

```
3
8
13
18
23
28
33
38
43
48
53
58
63
68
73
78
83
88
93
98
```

Il peut arriver que l'on souhaite itérer sur quelque chose, par exemple une liste, mais en sachant quel est l'index de l'élément parcouru. Avec un itérable ordonné, on pourrait créer une variable incrémentée à chaque passage, mais Python nous propose une solution plus… pythonesque : on itère non pas sur une liste de valeurs, mais sur une liste de paires `(index, valeur)`. Ainsi, comme vu plus haut, c'est bien l'itérable fourni à `for` qui est manipulé. On peut faire ça via la fonction `enumerate()` :

[68]:
```
for i in enumerate(ma_liste):
    print(i)
```

```
(0, 'Freddie Mercury')
(1, 'Brian May')
(2, 'Roger Taylor')
(3, 'John Deacon')
```

Nous récupérons bien les deux informations qui nous intéressent. Mais il faudra utiliser `i[0]` pour référencer l'index, et `i[1]` pour la valeur, ce qui n'est pas pratique, et même contraire à la philosophie de lisibilité de Python. Nous pouvons alors exploiter l'*unpacking* pour assigner deux variables à la fois avec les deux valeurs d'un tuple :

[69]:
```
for i, nom in enumerate(ma_liste):
    print('[{}] {}'.format(i, nom))
```

```
[0] Freddie Mercury
[1] Brian May
[2] Roger Taylor
[3] John Deacon
```

Revenons une seconde sur les dictionnaires. Nous l'avons vu, ils stockent des paires clés-valeurs, mais lorsque que l'on itère dessus, on récupère seulement les clés :

[70]:
```
membres = {'chant': 'Freddie Mercury', 'guitare': 'Brian May', 'batterie':␣
 ↪'Roger Taylor', 'basse': 'John Deacon'}

for i in membres:
    print(i)
```

```
guitare
basse
batterie
chant
```

De la même façon que le `enumerate()` qui renvoie une liste de tuples, la méthode `items()` sur un dictionnaire renvoie une valeur similaire, permettant d'itérer à la fois sur les clés et les valeurs

[71]:
```
for instrument, nom in membres.items():
    print('{}: {}'.format(nom, instrument))
```

```
Brian May: guitare
John Deacon: basse
```

```
Roger Taylor: batterie
Freddie Mercury: chant
```

3.3.2 Boucle conditionnelle

La boucle `for` est pertinente pour itérer sur quelque chose de connu, donc on « sait » potentiellement le nombre d'itérations qui seront effectuées.

Il est courant d'être dans une situation où le nombre de passages dans la boucle ne peut pas être prédit. Par exemple, lorsque l'on demande une information à l'utilisateur, et que l'on veut vérifier que la valeur entrée correspond à ce qui est attendu. Il n'est pas possible de boucler un nombre de fois défini en espérant que l'utilisateur parvienne à entrer une valeur acceptable dans ce délai, et il faut s'attendre à ce que celui-ci soit toujours plus imprévisible qu'un programme qui exécute par exemple 10 fois le même test avant de jeter l'éponge.

Pour ces besoins, la structure `while` exécute un bloc de code **tant qu'une condition est vraie**. On retrouve donc les conditions telles que définies précédemment. Si la condition est fausse d'emblée, la boucle `while` n'est jamais exécutée. Pour que cette boucle fonctionne correctement, il faut que la condition puisse varier, et devenir fausse à un moment, sinon on est en présence d'une boucle infinie (ce qui est parfois souhaitable, mais on repère généralement ce cas particulier car il utilise `True` comme condition). Il convient alors d'utiliser au moins une variable dans cette condition, et de la faire varier au sein de la boucle.

Syntaxiquement, une boucle `while` s'écrit comme ceci :

```
while <condition>:
    actions
```

[1]:
```
nom = ''  # on initialise la variable, sinon la condition testera une variable␣
 ↪inexistante la première fois

while not nom:  # bool(nom) renvoie True si nom n'est pas vide, le not devant␣
 ↪inverse cela.
                # on exécute donc la boucle si nom est vide
    nom = input('Quel est votre nom ? ')

print('Bonjour, {}'.format(nom))
```

```
Quel est votre nom ?
Quel est votre nom ?
Quel est votre nom ?
Quel est votre nom ?
Quel est votre nom ?
Quel est votre nom ?
Quel est votre nom ?  Gaëlle
```

```
Bonjour, Gaëlle
```

Voyons un autre exemple : nous avons une liste de noms, que nous voulons stocker, en générant un identifiant aléatoire pour chacun (dans un dictionnaire, par exemple, mais ça pourrait être une base de données). Par nature, l'aléatoire peut produire une valeur déjà sortie, il faut donc vérifier que

ce n'est pas le cas. Et comme la même valeur peut ressortir plusieurs fois d'affilée, le seul moyen d'être sûrs que notre valeur n'est pas déjà sortie est d'intégrer ce test dans une boucle `while` :

[73]:
```
import random  # ne nous occupons pas de ça pour le moment

noms = ['Floor Jansen', 'Damian Wilson', 'Hansi Kürsch', 'Tommy Karevik',␣
 ↪'Anneke van Giersbergen',
       'Marco Hietala', 'Jonas Renkse', 'Mike Mills', 'Marcela Bovio', 'Irene␣
 ↪Jansen', 'Robert Soeterboek',
       'John Jaycee Cuijpers', 'Edward Reekers', 'Jay van Feggelen', 'Maggy␣
 ↪Luyten', 'Lisette van den Berg']

identifications = {}  # on initialise un dictionnaire vide, qui sera rempli par␣
 ↪les noms

for nom in noms:  # notez que nom et noms sont deux variables bien différentes
    identifiant = None
    while identifiant is None or identifiant in identifications:  # "i in dict"␣
 ↪vérifie si i est une clé de dict
        identifiant = random.randint(1, 20)  # on produit un entier aléatoire␣
 ↪entre 1 et 20 compris
        print('Essayons {}…'.format(identifiant))
    # ici, on est sortis du while, donc on a l'assurance que la condition est␣
 ↪fausse : identifiant ne vaut pas None
    # et il n'est pas déjà présent dans la liste
    print('[{}] {}'.format(identifiant, nom))
    # enfin, on sauvegarde dans le dectionnaire
    identifications[identifiant] = nom
```

```
Essayons 16…
[16] Floor Jansen
Essayons 12…
[12] Damian Wilson
Essayons 9…
[9] Hansi Kürsch
Essayons 12…
Essayons 10…
[10] Tommy Karevik
Essayons 4…
[4] Anneke van Giersbergen
Essayons 3…
[3] Marco Hietala
Essayons 20…
[20] Jonas Renkse
Essayons 9…
Essayons 1…
[1] Mike Mills
```

```
Essayons 10…
Essayons 9…
Essayons 2…
[2] Marcela Bovio
Essayons 9…
Essayons 1…
Essayons 2…
Essayons 10…
Essayons 11…
[11] Irene Jansen
Essayons 20…
Essayons 17…
[17] Robert Soeterboek
Essayons 14…
[14] John Jaycee Cuijpers
Essayons 18…
[18] Edward Reekers
Essayons 7…
[7] Jay van Feggelen
Essayons 8…
[8] Maggy Luyten
Essayons 14…
Essayons 14…
Essayons 14…
Essayons 2…
Essayons 20…
Essayons 13…
[13] Lisette van den Berg
```

[74]:
```
print(identifications)
```

```
{1: 'Mike Mills', 2: 'Marcela Bovio', 3: 'Marco Hietala', 4: 'Anneke van
Giersbergen', 7: 'Jay van Feggelen', 8: 'Maggy Luyten', 9: 'Hansi Kürsch', 10:
'Tommy Karevik', 11: 'Irene Jansen', 12: 'Damian Wilson', 13: 'Lisette van den
Berg', 14: 'John Jaycee Cuijpers', 16: 'Floor Jansen', 17: 'Robert Soeterboek',
18: 'Edward Reekers', 20: 'Jonas Renkse'}
```

3.4 Voir plus loin

Il existe d'autres structures interagissant avec les boucles : fonctions, classes, gestionnaires de contexte… que nous verrons par la suite.

4 Fonctions

Nous avons déjà utilisé des fonctions, à commencer par `print()`. Il s'agit de blocs de code enregistrés sous un nom (le nom `print` dans cet exemple), et exécutés à volonté dans notre code. Une fonction reçoit des informations (**arguments**) nécessaires à son fonctionnement. Le but d'une fonction est d'exploiter ces arguments pour produire une valeur, que l'on appelle **valeur de retour**.

Lors de l'exécution du code, un appel à une fonction produit cette valeur, qui peut être utilisée dans une expression.

Dans certains rares cas, une fonction fait *autre chose* que produire une valeur. C'est le cas de `print()` par exemple, qui affiche des choses dans la sortie standard du programme. C'est ce qu'on appelle un **effet de bord**, et la fonction dépend alors du contexte dans lequel elle est exécutée. Les effets de bord peuvent êtres un appel réseau, la manipulation d'un fichier, etc. Utiliser des fonctions à effet de bord, ou en créer soi-même, n'est pas une erreur en soi, mais elles sont à utiliser judicieusement et avec parcimonie, parce qu'elles compliquent les tests et l'assurance qualité du programme.

4.1 Appel d'une fonction

De nombreuses fonctions sont présentes dans Python, que ce soit en *built-ins* (les fonctions présentes et directement utilisables lorsque l'on exécute Python), comme `print()` ou `type()`, ou dans les innombrables modules de la bibliothèque standard.

L'appel (on parle également d'exécution) d'une fonction consiste à nommer cette fonction, suivie de parenthèses contenant les arguments attendus. Lors de sa déclaration, une fonction indique les arguments nécessaires. Certains peuvent être optionnels, donc omis au moment de l'appel, et certaines fonctions peuvent accepter un nombre arbitraire d'arguments. Au moment de l'exécution, l'appel de la fonction est remplacé par sa valeur.

[75]:
```
valeur_retour = range(10)
```

Ici, `range(10)` est l'appel de la fonction `range` avec `10` comme unique argument. L'expression `range(10)` produit une valeur de retour (un objet contenant un intervalle de valeurs), que l'on peut utiliser comme n'importe quelle valeur. Ici, on le stocke dans une variable. On peut également utiliser un appel de fonction comme argument d'une autre fonction, étant donné qu'il se comporte comme une valeur.

[76]:
```
print(valeur_retour)
```

```
range(0, 10)
```

`range()` est un exemple de fonction acceptant un seul ou plusieurs arguments : elle fonctionne différemment selon que l'on lui donne un, deux ou trois arguments (mais n'en accepte pas plus). Tout cela est spécifié dans la définition de la fonction (et donc sa documentation)

4.1.1 Arguments positionnés, arguments nommés

Pour le moment, tous les appels à des fonctions que nous avons effectués prenaient des arguments dont la position déterminait le rôle (par exemple, `range(1, 10)` et `range(10, 1)` sont très différents, car on ne met pas les valeurs dans les mêmes arguments). Il est également possible de fournir des arguments à une fonction en les nommant explicitement, pour une plus grande lisibilité :

[77]:
```
print('Hello, world', end='!')  # nous fournissons l'argument "end"␣
↪explicitement
print('Seconde chaîne, sur la même ligne')
```

```
Hello, world!Seconde chaîne, sur la même ligne
```

Nous avons fourni un second paramètre à `print()`, nommé `end`. Si nous l'avions transmis sans préciser son nom, il ne se serait pas placé dans l'argument `end`, et le comportement aurait été autre :

[78]:
```
print('Hello, world', '!')
print('Seconde chaîne, sur une ligne différente')
```

```
Hello, world !
Seconde chaîne, sur une ligne différente
```

La raison en est que `print()` accepte un nombre indéfini d'arguments *positionnés*, c'est à dire non nommés, et affiche tous ces arguments séparés par une espace. L'argument optionnel `end`, lui, définit la chaîne de caractères à utiliser pour finir une ligne. Par défaut, elle vaut le caractère de saut de ligne `'\n'`. En spécifiant un autre `end`, `print()` n'insère plus automatiquement de retour à la ligne à la fin de son exécution, mais la chaîne que l'on lui a fourni.

Toutes les fonctions n'acceptent pas forcément que l'on les appelle avec des arguments nommés à la place des positionnés, et inversement. Référez-vous à la documentation des fonctions que vous souhaitez utiliser.

4.2 Déclaration de fonctions

Naturellement, vous voudrez rapidement créer vos propres fonctions, et ce pour plusieurs raisons :

- segmenter votre code en briques réutilisables. Si vous écrivez une fonction effectuant un calcul particulier, vous voudhez sans doute la réutiliser à plusieurs moments
- éclater la complexité globale de votre code : isolez les comportements spécifiques dans des fonctions, ainsi une fois créées, leur comportement interne les regardera, et vous pourrez les utiliser plus facilement, sans vous rappeler exactement les opérations effectuées à l'intérieur. Cette propriété est appelée **l'encapsulation**. Par exemple, vous demandez-vous constamment comment fonctionne `print()` en interne lorsque vous avez besoin de l'utiliser ?
- améliorer la structure du projet, sa compréhension et sa testabilité. Plus la logique est isolée en une myriade de fonctions simples, plus il sera facile d'assurer la qualité du produit.

La déclaration d'une fonction doit impérativement s'effectuer avant l'utilisation de celle-ci. Avant de déclarer une fonction, vous devez être en mesure de documenter trois choses la concernant :

- à quoi sert cette fonction ? Pourquoi avez-vous besoin de la créer, à quel besoin répond-elle ?
- quels sont les arguments de cette fonction, s'il y en a ? Votre fonction ne devrait pas pouvoir accéder à d'autres informations que celles que vous lui transmettez, alors ne négligez pas cette partie. Précisez tous les arguments attendus, leur nom, leur but, et leur(s) type(s)
- quelle est la valeur de retour de la fonction ? Quel est son type ? Et, quand nous aurons couvert ce sujet, quelles exceptions sont susceptibles d'être lancées par votre fonction, directement ou pas ?

Une fois que vous avez ces trois points en tête, ou posés par écrit, vous devriez pouvoir écrire la fonction en elle-même. La syntaxe est la suivante :

```
def <nom de la fonction>(<argument1>, <argument2>):
    corps de la fonction
    return <valeur de retour>
```

Le mot-clé `def` (pour *define*) indique le début d'une définition de fonction. Il est suivi par le nom de la fonction (qui suit exactement les mêmes règles de nommage que les variables, c'est à dire : des lettres, chiffres et des traits de soulignement, on ne commence par par un chiffre, et les mots-clés réservés comme `def` sont interdits), puis de parenthèses contenant les noms des arguments. Si votre fonction ne prend aucun argument, il faudra mettre des parenthèses vides.

Après cette ligne vient le corps de la fonction : c'est un bloc de code comme les autres, mais qui se termine automatiquement dès qu'il atteint une instruction `return`. C'est la valeur à droite du `return` qui sera la valeur de retour, et étant donné que c'est la finalité de la fonction, toute ligne se trouvant après le `return` ne sera jamais exécutée.

Au moment de la déclaration de votre fonction, celle-ci **n'est pas exécutée**. Elle est simplement mémorisée par Python, en attendant que l'on s'en serve.

[79]:
```
def addition(n1, n2):
    """
    Cette fonction très simple additionne deux nombres passés en arguments, et␣
↪renvoie leur résultat.
    Elle fait également un appel à print() pour des besoins de démonstration,␣
↪mais c'est à éviter sur
    une vraie fonction.

    Par ailleurs, cette chaîne de caractères multi-lignes écrite en début de␣
↪fonction est ce qu'on appelle une
    Docstring. Elle est enregistrée par Python au sein de la fonction en tant␣
↪que documentation : il est donc
    recommandé de toujours documenter ses fonctions par ce biais
    """
    print('n1 vaut {}, n2 vaut {}'.format(n1, n2))
    return n1 + n2
    print('Nous sommes après le return, cette ligne ne sera jamais exécutée')
```

Bien que la cellule précédente ait été exécutée, rien n'est affiché : la fonction est créée, mais reste inerte tant que l'on ne l'exécute pas.

[80]:
```
resultat = addition(2, 5)
```

```
n1 vaut 2, n2 vaut 5
```

Lors de l'exécution, Python interprète les lignes de la fonction, jusqu'à pouvoir produire une valeur de retour. Si aucun `return` n'est présent, à la fin du bloc de la fonction, la valeur `None` est retournée.

[81]:
```
print(resultat)
```

```
7
```

Les fonctions que l'on définit peuvent par défaut accepter les arguments nommés comme positionnés. Voyons ça avec un autre exemple :

[82]:
```
def division(quotient, diviseur):
    """
    Cette fonction effectue une division euclidienne de l'argument quotient par␣
 ↪l'argument diviseur,
    et renvoie le résultat.
    """
    return quotient / diviseur
```

[83]:
```
print(division(10, 2))
```

```
5.0
```

[84]:
```
print(division(2, 10))
```

```
0.2
```

[85]:
```
print(division(diviseur=2, quotient=10))
```

```
5.0
```

Le dernier exemple met en évidence l'utilisation de paramètres nommés. Si on en utilise, leur ordre importe peu, comme pour un dictionnaire. En conséquence, si l'on veut utiliser à la fois des arguments nommés *et* positionnés dans un même appel, il est nécessaire de mettre les positionnés d'abord.

4.2.1 Arguments optionnels

On peut définir des arguments comme optionnels lors de la déclaration d'une fonction : il s'agit alors de leur fournir une valeur par défaut, ce qui permettra d'appeler cette fonction en omettant cet argument. Il prendra alors la valeur par défaut. Pour la même raison que précédemment, lors de la déclaration, les arguments optionnels doivent être déclarés après les obligatoires.

[86]:
```
def division(quotient, diviseur, entier=False):  # l'argument entier est␣
 ↪optionnel
    """
    Cette fonction effectue une division euclidienne de l'argument quotient par␣
 ↪l'argument diviseur.
    Si l'argument optionnel entier est passé à True, elle renverra la partie␣
 ↪entière du résultat.
    Sinon, elle renvoie le résultat complet sous forme flottante.
    """
    print('entier vaut {}'.format(entier))
    resultat = quotient / diviseur
    if entier:
        return int(resultat)

    # pas besoin de else, car si la condition précédente est vraie, on est␣
 ↪sorti de la fonction
```

```
    return resultat
```

[87]:
```
print(division(5, 2))
```

```
entier vaut False
2.5
```

[88]:
```
print(division(5, 2, True))
```

```
entier vaut True
2
```

[89]:
```
print(division(5, 2, entier=False))  # on passe l'argument entier par son nom,␣
↪pour plus de lisibilité
```

```
entier vaut False
2.5
```

5 Gestion des fichiers

Au-delà des éléments de syntaxe du langage et des concepts sémantiques (comme l'approche objet), les interactions se sont limitées à écrire du texte sur la sortie standard, et attendre de l'utilisateur qu'il entre une chaîne de caractères. L'interaction, autrement dit la manipulation des entrées et sorties, est indispensable à tout programme pour concrétiser son exécution.

Parmi les entrées et sorties fréquemment manipulées, la gestion de fichiers est probablement ce qui vient à l'esprit en premier. Il s'agit d'être capables de lire un fichier présent sur le système d'exploitation, ou bien de pouvoir écrire dans un fichier, existant ou pas.

Commençons immédiatement, avec la fonction essentielle pour la manipulation de fichiers : `open`, qui est un *built-in*, au même titre que `print` (on n'a pas besoin de l'importer) :

[90]:
```
my_file = open('files/my_file')
print(my_file)
```

```
<_io.TextIOWrapper name='files/my_file' mode='r' encoding='UTF-8'>
```

La fonction `open` prend en paramètre un nom de fichier, et l'ouvre **en lecture**. Il renvoie ensuite un objet **représentant** ce fichier (de la classe `TextIOWrapper`). Naturellement, si le fichier n'existe pas, nous obtenons une erreur.

Nous avons ouvert le fichier en lecture, donc il convient d'accéder à son contenu. Nous disposons de plusieurs méthodes :

- `my_file.read()` lit l'intégralité du fichier, et le renvoie sous forme de chaîne de caractères
- `my_file.readline()` lit la prochaine ligne du fichier, et la renvoie sous forme de chaîne de caractères
- `my_file.readlines()` lit toutes les lignes du fichier, et les renvoie sous la forme d'un itérable

Notez que `read` et `readlines` parcourront effectivement l'intégralité du fichier, qui sera alors transféré en mémoire, tandis que `readline` ne stocke en mémoire qu'une ligne à la fois. Si vous prévoyez

de manipuler de gros fichiers, ou que vous n'avez pas connaissance de leur taille à l'avance, il est plus sûr d'utiliser `readline`.

Pour le moment, nous connaissons le contenu de `my_file`, donc allons vers la simplicité :

[91]:
```
content = my_file.read()
print(content)
```

```
Ce fichier
contient
du texte
```

Pour le moment, `open` a ouvert le fichier. Du côté du système d'exploitation, notre programme Python a posé un verrou sur ce fichier, pour signaler son intention de le lire. Il est crucial de bien libérer ces ressources une fois que l'on a fini notre opération.

[92]:
```
my_file.close()
print(my_file.closed)
```

```
True
```

On ne peut plus lire un fichier fermé (comme on ne peut pas appeler deux fois de suite `read` sur un fichier ouvert : une fois parcouru, il est parcouru)

[93]:
```
my_file.read()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-93-2b1e0f30a114> in <module>()
----> 1 my_file.read()

ValueError: I/O operation on closed file.
```

5.1 Context Managers

Un problème se pose alors : pour une multitude de raisons, notre code peut ne jamais atteindre l'appel à la méthode `close` : une exception peut se produire entre l'ouverture et la fermeture, et remontera la pile d'exécution sans jamais fermer la ressource. Il existe plusieurs moyens de résoudre efficacement ce problème. L'un d'entre eux consiste à utiliser la clause `finally` d'une gestion d'exception. L'autre est l'utilisation d'un *context manager*, une structure Python (intégrée dans le langage dans sa version 2.5) permettant d'ouvrir une ressource dans un bloc de code défini, et de garantir sa fermeture lorsque l'on sort du bloc, d'une manière ou d'une autre.

Les *context managers* sont compatibles avec les objets implémentant les méthodes magiques `__enter__` et `__exit__` ou décorées par `@contextmanager`, comme c'est le cas pour la fonction `open`. Leur utilisation passe par le mot-clé `with`.

[94]:
```
with open('files/my_file') as fd:
    print('Le fichier est-il fermé ? {}'.format(fd.closed))
print('Une fois sorti du with, le fichier est-il fermé ? {}'.format(fd.closed))
```

```
Le fichier est-il fermé ? False
Une fois sorti du with, le fichier est-il fermé ? True
```

On a maintenant la garantie que notre fichier (appelé ici *fd*, pour *file descriptor*) sera bien fermé même si une exception non traitée se produit dans le bloc `with`. Un appel à `close` n'est donc plus nécessaire, car il est implicite dans l'utilisation du *context manager*.

Revenons à notre objet fichier. En dehors des cas où on contrôle le fichier à lire, et qu'on choisit de le lire intégralement par facilité, il est préférable de le parcourir comme un générateur. De cette façon, l'empreinte mémoire du fichier est bien plus sous contrôle. Nous avons évoqué la méthode `readline`, mais la classe `TextIOWrapper` (comme toute classe dite de *file-like object*) possède des méthodes magiques permettant l'itération directe :

[95]:
```
with open('files/my_file') as fd:
    for line in fd:
        print('Nouvelle ligne : « {} »'.format(line))
```

```
Nouvelle ligne : « Ce fichier
 »
Nouvelle ligne : « contient
 »
Nouvelle ligne : « du texte
 »
```

Notez que les guillemets fermants se retrouvent à la ligne. C'est parce que `readline` (appelé implicitement par la structure `for`) renvoie les lignes entières, caractère de saut de ligne compris. Il ne faut pas l'oublier en manipulant un fichier.

5.2 Manipulation de fichiers binaires

Nous avons vu que Python possède deux types représentant des chaînes de caractères : `str` et `bytes`, ce dernier découpant octet par octet une chaîne, naturellement plus adapté pour traiter les chaînes qui ne représentent pas du texte, comme ça peut être le cas pour le contenu d'un fichier image. Si on veut manipuler un tel fichier, il sera préférable d'avoir accès à son contenu sous forme d'objet `bytes`. Cela se fait directement sur l'appel à `open`, qui prend un second paramètre, optionnel, indiquant le type d'ouverture que l'on souhaite : en lecture ou en écriture ? En binaire ou en texte ?

Cette valeur se transmet sous la forme d'une chaîne de caractère comprenant le ou les *drapeaux* à activer. Par défaut, elle vaut `r`, pour « read » (ce qui permet de lire un fichier). Si on veut écrire dans le fichier, il faut le passer à `w` (« write »). Mais indépendamment de cela, dans les deux cas, on peut souhaiter ouvrir le fichier en mode binaire. Il faut alors ajouter `b` à cette chaîne. Par exemple, une ouverture en lecture binaire s'écrira `rb`.

Contrairement à l'ouverture en mode texte, où on peut légitimement s'attendre à trouver de nombreux caractères de saut de ligne, un fichier binaire ne peut pas être découpé efficacement ligne

par ligne. La méthode d'itération recommandée est donc de parcourir chaque octet jusqu'à ne plus rien obtenir (l'objet `byte` sera vide) :

[96]:
```
with open('files/chat.jpg', 'rb') as fd:
    count = 0
    while True:
        b = fd.read(1)
        if b != '':
            count += 1
            if count >= 10:
                break
            print(b)
```

```
b'\xff'
b'\xd8'
b'\xff'
b'\xe0'
b'\x00'
b'\x10'
b'J'
b'F'
b'I'
```

Notez que nous obtenons à chaque fois des objets `bytes`, en opposition aux `str` plus haut.

5.3 Écriture de fichier

Pour ouvrir un fichier en écriture, il faut passer `w` en mode à la fonction `open`. Un fichier ouvert en écriture ne peut pas être lu, tout comme il est impossible d'écrire dans un fichier ouvert en lecture.

[97]:
```
with open('files/my_file_2', 'w') as fd:
    fd.write('contenu')
```

Vérifions maintenant le contenu du fichier :

[98]:
```
with open('files/my_file_2') as fd:
    print(fd.read())
```

```
contenu
```

Attention : le mode d'ouverture `w` écrase le fichier, s'il existe déjà. Il est parfois souhaitable d'écrire dans un fichier, mais à la fin du contenu existant. Dans ce cas, le mode d'ouverture `a` (« append ») est nécessaire.

[99]:
```
with open('files/my_file_2', 'a') as fd:
    fd.write('\nnouveau contenu')
```

[100]:
```
with open('files/my_file_2') as fd:
    print(fd.read())
```

```
contenu
nouveau contenu
```

Notez le rajout du caractère `\n`, pour provoquer un saut de ligne. Aucun caractère n'est rajouté automatiquement par Python.

6 Les imports et modules en Python

Organiser l'intégralité d'un projet au sein d'un même fichier python est très certainement une erreur. L'ensemble des classes, fonctions et actions, selon leur finalité, doit être organisé de façon logique et de manière à favoriser la maintenance du projet. Les imports nous permettent alors de manipuler dans du code python des structures définies ailleurs.

6.1 Vocabulaire

Un fichier Python (c'est à dire contenant du code python et portant généralement le suffixe `.py`) est appelé un **module**.

Un dossier contenant des **modules** Python importables est appelé un **package**. Pour que les modules contenus dans un package soient importables, ce dernier doit contenir un fichier nommé `__init__.py`, même s'il est vide.

Un **module** Python est manipulable dans le code sous la forme d'un objet. Cet objet possède, comme attributs, les **noms** définis dans ce module.

[101]:
```python
# dans mymodule.py


def my_function():
    pass


class MyClass:
    pass
```

Ici, le module `mymodule.py` possèdera `my_function` et `MyClass` comme **noms**. Ceux-ci font logiquement référence aux objets définis (fonction et classe). Le fait que ces noms fassent partie d'un module est un **espace de noms** (ou **namespace**).

Un interpréteur Python possède d'emblée plusieurs noms actifs (comme la fonction `print`). Il s'agit de **built-ins**, soit des noms automatiquement intégrés à l'**espace de noms** courant. Ils sont visibles dans l'objet `__builtins__` (qui est lui-même un **built-in**)

[102]:
```python
print('\n'.join(__builtins__.__dict__.keys()))
```

```
__debug__
__doc__
TypeError
IsADirectoryError
sum
```

```
abs
getattr
NotImplementedError
BufferError
UnboundLocalError
chr
KeyboardInterrupt
ConnectionError
ConnectionRefusedError
UnicodeWarning
ReferenceError
divmod
isinstance
__import__
TabError
Ellipsis
ConnectionAbortedError
ArithmeticError
OSError
StopIteration
ChildProcessError
ZeroDivisionError
FileExistsError
bytearray
memoryview
ConnectionResetError
UnicodeError
UserWarning
None
BlockingIOError
globals
DeprecationWarning
map
set
reversed
BrokenPipeError
property
zip
delattr
UnicodeEncodeError
ImportWarning
bytes
BytesWarning
FileNotFoundError
AttributeError
str
SystemExit
True
```

```
NotImplemented
ord
bin
UnicodeTranslateError
classmethod
FutureWarning
__spec__
__build_class__
TimeoutError
IOError
object
enumerate
repr
OverflowError
filter
open
float
callable
help
__IPYTHON__
len
input
round
exec
oct
ProcessLookupError
SystemError
display
setattr
copyright
__name__
complex
PermissionError
AssertionError
hex
EOFError
UnicodeDecodeError
any
id
issubclass
__loader__
get_ipython
InterruptedError
NotADirectoryError
format
super
SyntaxWarning
BaseException
```

```
bool
locals
FloatingPointError
Warning
print
hash
frozenset
dict
IndentationError
vars
RuntimeError
slice
min
staticmethod
EnvironmentError
NameError
license
pow
eval
IndexError
all
hasattr
credits
ValueError
KeyError
SyntaxError
iter
ImportError
sorted
dir
LookupError
GeneratorExit
__package__
max
RuntimeWarning
ascii
False
compile
range
MemoryError
type
int
list
tuple
next
PendingDeprecationWarning
Exception
ResourceWarning
```

Au-delà des `built-ins`, on peut constater deux types de modules importables, même s'ils ne possèdent structurellement aucune différence : les modules de la **bibliothèque standard** (couramment qualifiée de **stdlib**) et les **modules tiers**. Les premiers sont présents dans l'installation de Python, et n'attendent que d'être importés. Les seconds doivent être installés par l'utilisateur.

6.2 Import de module

L'import de module consiste à chercher un module présent sur le système, et l'importer dans notre **namespace**, intégralement (sous sa forme de module) ou partiellement, en ciblant un ou plusieurs noms dans ce module. Voyons voir par l'exemple avec un module de la bibliothèque standard.

[103]:
```
import datetime

print(type(datetime))
print(type(datetime.date))  # datetime.date est le type directement utilisable
print(type(datetime.datetime))  # datetime est un module, datetime.datetime est␣
↪une classe (un type)
```

```
<class 'module'>
<class 'type'>
<class 'type'>
```

L'exemple précédent aurait pu s'écrire en important directement le type (donc la classe) `date`, ce qui aurait eu pour effet de ne pas importer le module (et donc ses noms) mais directement `date`

[104]:
```
from datetime import date

print(date.today())
```

```
2018-06-30
```

Il est possible d'importer plusieurs noms directement.

[105]:
```
from datetime import date, datetime

print(type(datetime))  # ici, datetime est bien le nom contenu dans le module␣
↪nommé de la même façon
```

```
<class 'type'>
```

7 Programmation objet

7.1 Qu'est-ce qu'un objet ?

Un objet est un… truc. Un machin. Un bidule.

Ça peut paraître une définition floue, mais c'est parce que c'est exactement ce que peut être un objet: n'importe quoi que vous décidiez de coder. L'objet est un moyen de dire à la machine : « ce < entrez_ici_un_nom_de_truc > possède telle donnée, et fait telle chose avec ».

En Python, absolument tout est un objet : une chaîne, un entier, un dictionnaire, une liste, une fonction… Vous avez donc manipulé des objets sans le savoir. Maintenant vous allez créer les vôtres.

Créer des objets se fait en deux étapes : décrire à quoi ressemble votre objet, et demander à l'ordinateur d'utiliser cette description pour le fabriquer.

[106]:
```
class MyClass():
    pass
new_object = MyClass()
print(new_object)
```

```
<__main__.MyClass object at 0x7f38a839d128>
```

La description de l'objet (le plan de construction) est le bloc défini par `class`.

[107]:
```
class MyClass():
    pass
```

C'est ce qu'on appelle une classe. La classe est le moyen, en Python, de décrire à quoi va ressembler un objet.

On dit ici : « Cet objet s'appelle MyClass. »

C'est tout.

Ensuite, on crée l'objet :

[108]:
```
new_object = MyClass()
```

`MyClass()` (notez les parenthèses), est la syntaxe Python pour dire « fabrique un objet à partir de ce plan ». Le nouvel objet va être retourné, et mis dans la variable `new_object`.

La variable `new_object` contient un objet issu de la classe `MyClass`, on dit qu'il contient une *instance* de `MyClass`. C'est pour cela que l'action de créer un objet à partir d'une classe est appelée l'`instanciation`.

Python ne sait pas grand chose sur cet objet si ce n'est son nom et son adresse en mémoire. Si on fait `print` dessus, c'est donc ce qu'il vous donne :

[109]:
```
print(new_object)
```

```
<__main__.MyClass object at 0x7f38a839dd68>
```

On peut créer autant d'objets que l'on veut à partir d'une classe :

[110]:
```
MyClass()
```

[110]:
```
<__main__.MyClass at 0x7f38a839dcc0>
```

[111]:
```
MyClass()
```

[111]:
```
<__main__.MyClass at 0x7f38a839d240>
```

[112]:
```
MyClass()
```

[112]:
```
<__main__.MyClass at 0x7f38a839d320>
```

7.2 Méthodes

Les méthodes sont des fonctions déclarées à l'intérieur de la classe. *Méthode* est juste un nom pour dire « cette fonction est dans une classe ».

[113]:
```
class MyClass():

    def the_method(this_object):
        return 'Edgar Morin'
```

Nous avons créé une « fonction » nommée `the_method` dans le bloc de la classe. Puisqu'elle est dans ce bloc, cette fonction est ce qu'on appelle une méthode.

Elle n'existe pas en dehors de la classe.

[114]:
```
the_method()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-114-9ebb5f1a9f40> in <module>()
----> 1 the_method()

NameError: name 'the_method' is not defined
```

Mais, elle est attachée à chaque objet. On peut l'utiliser en faisant `<variable_contenant_objet>.<nom_de_méthode>`. Ainsi, pour appeler une méthode, il nous faut forcément une instance :

[115]:
```
new_object = MyClass()
new_object.the_method()
```

[115]:
```
'Edgar Morin'
```

Arrivé à ce stade, vous devez vous demander « quel est ce `this_object` » qui est défini comme paramètre de la méthode ?

C'est une spécificité de Python : quand vous appelez une méthode depuis un objet, l'objet est automatiquement passé en premier paramètre par Python. C'est automatique, et invisible.

C'est très facile à comprendre en faisant une méthode qui retourne `this_object` pour voir ce qu'il y a dedans :

[116]:
```
class MyClass():

    def the_method(this_object):
```

```
        return this_object

the_object = MyClass()
print(the_object)
```

```
<__main__.MyClass object at 0x7f38a8359438>
```

[117]:
```
print(the_object.the_method())
```

```
<__main__.MyClass object at 0x7f38a8359438>
```

Python fait comme si on avait fait :

[118]:
```
MyClass.the_method(the_object)
```

[118]:
```
<__main__.MyClass at 0x7f38a8359438>
```

`this_object` et `the_object` contiennent la même chose : l'objet en cours. `this_object` est juste le nom que nous avons donné à notre premier argument, mais Python passe toujours automatiquement en premier argument l'objet en cours afin que vous puissiez modifier cet objet à l'intérieur d'une méthode.

On dit donc que `this_object` est l'instance courante.

Une méthode doit donc toujours déclarer au moins un paramètre pour accueillir la référence à l'objet en cours, sinon ça plante. Voyez plutôt :

[119]:
```
class MyClass():

    def method_without_this_object():
        return 'test'

    def method_with_this_object(this_object):
        return 'test'

the_object = MyClass()
the_object.method_with_this_object()
```

[119]:
```
'test'
```

[120]:
```
the_object.method_without_this_object()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-120-7a4d0b8752e9> in <module>()
----> 1 the_object.method_without_this_object()
```

```
TypeError: method_without_this_object() takes 0 positional arguments but 1 was␣
↪given
```

Python dit que `method_without_this_object` est déclarée sans argument mais on lui en a passé un tout de même. Et en effet, Python lui a passé de manière invisible l'objet en cours en tant que premier argument.

En résumé, on déclare une classe avec une méthode ainsi :

[121]:
```
class MyClass():

    def name_of_method(this_object):
        # faire quelque chose ici
        pass
```

Et on l'utilise ainsi :

[122]:
```
variable = MyClass()
variable.name_of_method()
```

7.3 Conventions

En Python, peu de choses sont forcées. La plupart des choses sont des conventions. Mais ce sont des conventions fortes, les gens y tiennent.

Parmi ces conventions, il y a les conventions de nommage, à savoir :

- On nomme les classes sans espace, avec des majuscules : `NameOfTheClass` ;
- On nomme le reste en minuscule avec des underscores : `name_of_the_method`, `name_of_the_attribute`, etc.

La convention la plus surprenante est celle du premier paramètre des méthodes qui contient l'objet en cours. Son nom n'est pas forcé, contrairement aux autres langages (comme `this` en Java par exemple), en fait c'est un paramètre tout à faire ordinaire. Néanmoins, la (très très forte) convention est de l'appeler `self` :

[123]:
```
class MyClass():

    def name_of_method(self):
        # faire quelque chose ici
        pass
```

Appelez-donc TOUJOURS le premier paramètre de chaque méthode `self`, et rappelez-vous qu'il contiendra toujours l'instance courante.

Enfin, les concepteurs de Python ont ajouté une convention supplémentaire : les méthodes appelées automatiquement sont appelées `__method_name__` (avec deux underscores de chaque côté).

En effet, il existe un certains nombre de méthodes que vous pouvez écrire, et si vous les nommez d'une certaine façon, elle seront appelées automatiquement quand une condition spéciale (liée au nom), se présente.

Nous verrons cela avec la méthode `__init__`.

Attributs

Vous avez vu que les méthodes étaient juste des fonctions attachées à un objet. Hé bien les attributs sont juste des variables attachées à un objet.

En fait, techniquement, les méthodes sont aussi des attributs, mais faisons comme si ce n'était pas le cas et séparons :

- méthodes : fonctions pour objets
- attributs : variables pour objets

Comme Python est un langage dynamique, on peut ajouter n'importe quel attribut à n'importe quel objet en utilisant la syntaxe `<objet>.<nom_attribut>` :

[124]:
```
print(the_object.attribute)  # l'attribut n'existe pas
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-124-1c2a8eb8673d> in <module>()
----> 1 print(the_object.attribute)  # l'attribut n'existe pas

AttributeError: 'MyClass' object has no attribute 'attribute'
```

[125]:
```
the_object.attribute = 'valeur'
print(the_object.attribute)
```

```
valeur
```

Et c'est là que vous allez voir l'interêt de `self` :

[126]:
```
class MyClass():

    def show(self):
        # self est l'objet en cours, donc on à accès à ses attributs !
        return self.attribute

    def modify(self):
        # on peut modifier les attributs depuis l'intérieur
        self.attribute = 'autre valeur'
```

On peut accéder aux attributs depuis l'intérieur et l'extérieur d'un objet :

[127]:
```
the_object = MyClass()
the_object.show()  # l'attribut n'existe pas encore
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-127-ea0319fd7cee> in <module>()
```

```
      1 the_object = MyClass()
----> 2 the_object.show()  # l'attribut n'existe pas encore

<ipython-input-126-04984948e6f8> in show(self)
      3     def show(self):
      4         # self est l'objet en cours, donc on à accès à ses attributs !
----> 5         return self.attribute
      6
      7     def modify(self):

AttributeError: 'MyClass' object has no attribute 'attribute'
```

[128]:
```
the_object.attribute = "maintenant il existe !"
the_object.show()
```

[128]:
```
'maintenant il existe !'
```

[129]:
```
the_object.modify()  # l'attribut est modifié par la méthode
the_object.attribute
```

[129]:
```
'autre valeur'
```

Comme une méthode est juste une fonction attachée à un objet, on peut lui passer des paramètres. Et une méthode a accès à l'objet en cours, elle peut modifier l'objet en cours.

Donc, une méthode peut modifier l'objet en cours en fonction des paramètres passés :

[130]:
```
class MyClass():

    def show(self):
        return 'La valeur actuelle est {}'.format(self.attribute)

    def modify(self, value):
        self.attribute = value * 2

the_object = MyClass()
the_object.attribute = 1  # on met un entier en valeur cette fois
the_object.show()
```

[130]:
```
'La valeur actuelle est 1'
```

[131]:
```
the_object.modify(2)  # on passe un paramètre
the_object.show()
```

[131]:
```
'La valeur actuelle est 4'
```

Les méthodes servent exactement à cela : retourner les valeurs à l'intérieur de l'objet, ou à les modifier.

7.4 Initialisation d'un objet

On souhaite généralement donner un état de départ à tout nouvel objet créé. Par exemple, si vous travaillez sur un jeu vidéo de course de voitures, vous voudrez peut-être créer un objet voiture avec du carburant et une couleur de peinture.

Il serait contre productif de devoir les spécifier à chaque fois. Pour automatiser le travail, Python met à disposition des méthodes appelées automatiquement quand une condition est remplie. Ce sont les méthodes nommées `__method__`.

Dans notre cas, on veut que notre objet ait un état de départ, donc on va utiliser la méthode qui est appelée automatiquement après la création de l'objet. C'est la méthode `__init__`.

Si vous définissez une méthode avec le nom `__init__`, Python va automatiquement, et de manière invisible, appeler la méthode après avoir créé l'objet :

[132]:
```
class MyClass():

    def __init__(self):
        print("L'objet a été créé !")
        print('La méthode est appelée automatiquement')

the_object = MyClass()
```

```
L'objet a été créé !
La méthode est appelée automatiquement
```

`__init__` est une méthode tout à fait ordinaire. Seul le nom est important: comme elle est appelée ainsi, Python la détecte automatiquement et l'appelle au bon moment.

C'est très utile pour fournir un état de départ :

[133]:
```
class Vehicle:

    def __init__(self):
        self.fuel = 100
        self.color = 'noir'

v = Vehicle()
v.color
```

[133]:
```
'noir'
```

[134]:
```
v.fuel
```

[134]:
```
100
```

Bien entendu, on veut généralement pouvoir changer l'état de départ. Ici encore, Python nous aide : **si vous passez des paramètres à l'instanciation, ils sont passés automatiquement à `__init__`**.

Comme `__init__` est une méthode ordinaire (et donc une fonction), on peut lui donner des paramètres avec des valeurs par défaut mais accepter une nouvelle valeur si elle est passée :

[135]:
```
class Vehicle:

    def __init__(self, how_much_fuel=100, what_color='noir'):
        self.fuel = how_much_fuel
        self.color = what_color

v = Vehicle()  # sans paramètres, les valeurs par défaut sont utilisées
v.color
```

[135]:
```
'noir'
```

[136]:
```
v.fuel
```

[136]:
```
100
```

[137]:
```
v2 = Vehicle(50, 'rose')  # on change les valeurs
v2.color
```

[137]:
```
'rose'
```

[138]:
```
v2.fuel
```

[138]:
```
50
```

[139]:
```
v3 = Vehicle(what_color="fushia virant sur l'aigue marine")
v3.color
```

[139]:
```
"fushia virant sur l'aigue marine"
```

[140]:
```
v3.fuel
```

[140]:
```
100
```

Les valeurs passées dans `Vehicle(given, args)`, sont automatiquement repassées à `__init__(self, given, args)`.

En résumé : `__init__` est appelée automatiquement après que l'objet soit créé, et on lui passe les paramètres passés à l'instanciation. Comme `__init__` est une méthode et a accès à l'instance courante, elle peut ajouter des attributs à l'objet en cours en fonction de ces paramètres. On utilise ça pour créer des objets avec un état de départ par défaut, et permettre une personnalisation de cet état de départ.

Nous avons mis des noms différents pour les paramètres d'`__init__` et les attributs de l'objet, mais en réalité on leur met souvent le même nom par convention. Donc l'exemple ci-dessus s'écrirait plutôt :

[141]:
```python
class Vehicle():

    def __init__(self, fuel=100, color="noire"):
        self.fuel = fuel
        self.color = color
```

`fuel` et `color` ne sont pas la même chose que `self.fuel` et `self.color` : les premiers n'existent que dans `__init__`. Une fois qu'`__init__` a fini de s'exécuter, ils disparaissent. Les seconds sont des attributs de l'objet en cours, et sont donc toujours disponibles tant que l'objet existe. Le fait qu'ils portent le même nom ne pose aucun problème car la syntaxe `self.` les différencie.

7.5 Résumé

La classe, c'est un plan.

L'objet, c'est ce qu'on crée avec le plan.

Une méthode, c'est une fonction déclarée dans une classe (qui est attachée à chaque objet produit, et on lui passe en premier paramètre l'objet en cours).

Un attribut, c'est une variable attachée à un objet.

Une instance d'une classe, c'est l'objet issu d'une classe.

8 Découverte de la bibliothèque standard Python

Lorqu'on installe Python, on a accès à l'interpréteur Python lui-même, qui exécute du code Python selon la syntaxe définie dans sa spécification, mais également un ensemble de modules, chacun spécialisé dans un traitement particulier. L'ensemble de ces modules s'appelle la **bibliothèque standard** (qu'on appelle souvent *stdlib*, pour « · standard library · »).

Cette bibliothèque standard a la juste réputation d'être très complète : elle vous permettra de manipuler énormément de sujets, gérer un grand nombre de formats et protocoles, sans avoir à installer quoi que ce soit d'autre que Python (ce qui facilite la portabilité des programmes et leur installation). Par ailleurs, les modules de la bibliothèque standard Python sont également couverts par les exigences qualité du projet Python global.

Grâce à son impressionnante bibliothèque standard, on dit de Python qu'il est livré « avec les piles ». Nous allons voir ce que cela signifie, en découvrant les bibliothèques les plus indispensables.

8.1 Manipulation des dates et temps

Les calculs temporels sont une chose incroyablement compliqué, d'un point de vue mathématique : ils ne répondent pas à des règles universelles, et sont remplis de cas exceptionnels (est-il possible de dire exactement le nombre de jours que compte un mois ? Et une heure ?). Par conséquent, l'utilisation d'une bibliothèque robuste connaissant toutes les règles est indispensable pour pouvoir manipuler sereinement des valeurs temporelles.

Le package `datetime` contient plusieurs modules qui nous seront utiles :

- `date`, qui permet de manipuler des **jours**
- `time`, qui permet de manipuler des **heures**, sans les rattacher à un jour précis

- `datetime`, qui permet de manipuler des valeurs précises dans le temps · : des objets **date et heure**
- `timedelta`, qui permet de manipuler des **intervalles de temps** (qui, par nature, sont variables en valeur réelle)

Commençons avec le module `date`. Après l'avoir importé, vous pouvez instancier un objet à une date précise de cette façon :

[142]:
```
from datetime import date

holy_grail_release_date = date(day=9, month=4, year=1975)
print(holy_grail_release_date)
```

```
1975-04-09
```

Attention à ne pas, par réflexe, inscrire les valeurs de jour ou de mois sur deux chiffres : ils seraient alors probablement reconnus par Python comme une notation numérique en octal, et n'auraient pas la même valeur que celle que vous croyez avoir entrée.

Maintenant que l'on a un objet `date`, il est possible d'accéder à plusieurs de ses attributs :

[143]:
```
print(holy_grail_release_date.month)
print(holy_grail_release_date.day)
print(holy_grail_release_date.weekday())
```

```
4
9
2
```

Il est possible d'instancier un objet `date` à la date actuelle via `date.today()` :

[144]:
```
today_date = date.today()
print(today_date.year)
```

```
2018
```

Les objets `time` se manipulent de façon similaire :

[145]:
```
from datetime import time

workday_start = time(hour=9)
print(workday_start.minute)
```

```
0
```

Contrairement à `date`, il n'est pas possible d'instancier un objet représentant l'heure courante seule. Pour cela, il vaut mieux passer par un objet `datetime` : celui-ci cumule les deux objets précédemment vus, et permet le cibler un instant précis à la microseconde près.

[146]:
```
from datetime import datetime

now = datetime.now()
```

```
print(now.second)
print(now.microsecond)
```

```
20
271508
```

Il est possible d'accéder à la composante `date` ou `time` d'un objet `datetime` :

[147]:
```
print(now.time())
print(now.date())
```

```
12:40:20.271508
2018-06-30
```

Une des actions les plus puissantes de la bibliothèque `datetime` est le formatage de dates : il permet de créer une représentation textuelle d'une date et/ou heure grâce à un **patron de formatage** : une chaîne de caractère comprenant les marqueurs spécifiques qui seront remplacés par les composantes concernées de l'objet.

[148]:
```
print(now.strftime('%d/%m/%Y'))
```

```
30/06/2018
```

Ici, trois marqueurs sont utilisés :

- `%d` représente le jour du mois, sur deux chiffres
- `%m` représente le mois, sur deux chiffres
- `%Y` représente l'année, sur quatre chiffres

La documentation officielle de ce module recence tous les marqueurs existants. Il faut savoir que ces marqueurs supportent la *localisation*, ce qui signifie qu'ils tiendront compte de la locale de date active pour formater l'objet.

[149]:
```
print(now.strftime('%A, %B %d'))
```

```
Saturday, June 30
```

Il est également possible de **lire** une date représentée dans un format défini, à partir d'une chaîne de caractères. Contrairement au formatage, cela ne fonctionne que sur un objet `datetime`.

[150]:
```
date_repr = '01/02/17'
this_date = datetime.strptime(date_repr, '%d/%m/%y')
print(this_date)
```

```
2017-02-01 00:00:00
```

Travailler avec une date, c'est bien, mais avec deux, c'est mieux ! Surtout quand il s'agit de calculer la différence entre ces dates. On travaille alors avec un objet `timedelta`. Les opérations standard sur les objets de date renvoient d'elles-mêmes des `timedelta`s

[151]:
```
delta = now - this_date
print(delta)
```

```
print(delta.days)
```

```
514 days, 12:40:20.271508
514
```

[152]:
```
new_random_date = holy_grail_release_date + delta
print(new_random_date)
```

```
1976-09-04
```

8.2 Fonctions mathématiques

Le module `math` réserve peu de surprises. C'est lui qui contient toutes les fonctions mathématiques moins courantes que les opérateurs accessibles en *built-ins* :

[153]:
```
import math

print(math.sin(math.pi / 2))
print(math.cos(math.radians(45)))
print(math.sqrt(81))
```

```
1.0
0.7071067811865476
9.0
```

8.3 Calculs décimaux

En informatique, les calculs décimaux sont un problème, car les nombres flottants, que l'on utilise au quotidien, souffrent d'imprécision :

[154]:
```
print(0.1 + 0.1 + 0.1)
```

```
0.30000000000000004
```

Pour les calculs nécessitant de la fiabilité (par exemple, pour tout ce qui touchera à des prix), cette limite n'est pas acceptable. On peut alors utiliser le module `decimal`, fournissant un type `Decimal` conçu pour être rigoureusement exact.

[155]:
```
from decimal import Decimal

print(Decimal('0.1') + Decimal('0.1') + Decimal('0.1'))
```

```
0.3
```

[156]:
```
print(Decimal(10) / Decimal(3))
print(10 / 3)
```

```
3.3333333333333333333333333333
3.3333333333333335
```

8.4 Pseudo-aléatoire

Un processeur binaire est fondamentalement incapable de générer de l'aléatoire : il exécute de façon prédictible les instructions qui lui sont fournies. On peut néanmoins, en collectant des valeurs difficilement prédictibles, comme les identifiants des processus actuellement exécutés sur le système, la date et l'heure, et d'autres variables, générer des nombres **pseudo-aléatoires** : ils auront l'air aléatoires, mais ne doivent pas être utilisés à des fins cryptographiques, là où la robustesse de l'aléa est une des clés de la sécurité.

Python fournit un module pratique pour traiter ce pseudo-aléatoire : `random`.

[157]:
```
import random

for i in range(10):
    print('{}: {}'.format(i, random.randint(0, 9)))
```

```
0: 3
1: 0
2: 6
3: 9
4: 2
5: 4
6: 3
7: 7
8: 0
9: 6
```

`randint` génère un nombre entier situé entre la valeur minimale et la maximale, incluses.

`choice` permet de piocher une valeur aléatoire dans un itérable.

[158]:
```
letters = 'abcdefghijklmnopqrstuvwxyz'
print(random.choice(letters))
```

```
u
```

`shuffle` permet de mélanger aléatoirement les valeurs d'un itérable. Celui-ci doit être mutable, car il est directement modifié.

[159]:
```
letters_list = list(letters)
random.shuffle(letters_list)
print(letters_list)
```

```
['p', 'd', 'f', 'j', 'c', 'y', 'k', 'g', 'h', 'n', 'w', 'l', 'o', 't', 'b', 'v',
'q', 'm', 's', 'i', 'z', 'r', 'u', 'e', 'x', 'a']
```

8.5 Outils d'itération

L'itération est le cœur de Python. Outre les possibilités disponibles en *built-in*, le module `itertools` fournit de nombreux moyens de traiter un itérable, ou encore des fonctions permettant de créer des générateurs prédéfinis.

[160]:
```
import itertools

counter = itertools.count()

print(next(counter))
print(next(counter))
print(next(counter))
```

```
0
1
2
```

[161]:
```
cycler = itertools.cycle(['first value', 'second value', 'third value'])

for _ in range(10):
    print(next(cycler))
```

```
first value
second value
third value
first value
second value
third value
first value
second value
third value
first value
```

[162]:
```
repeater = itertools.repeat('We are the champions!', 3)

for value in repeater:
    print(value)

print('Of the world!')
```

```
We are the champions!
We are the champions!
We are the champions!
Of the world!
```

8.6 Collections

Le module `collections` fournit plusieurs types d'itérables venant compléter ceux disponibles en *built-ins*.

Par exemple, le `namedtuple` permet de créer un type sur mesure cumulant cumulant la non-mutabilité d'un tuple avec l'expressivité d'un dictionnaire.

[163]:
```
import collections

Color = collections.namedtuple('Color', ('red', 'green', 'blue'))
print(Color)
```

```
<class '__main__.Color'>
```

[164]:
```
pink = Color(red=241, green=93, blue=215)
print(pink)
print(pink.blue)
```

```
Color(red=241, green=93, blue=215)
215
```

L'une des plus grosses limites du type `dict` est le fait qu'il n'est pas ordonné. Il existe de nombreux cas dans lesquels il peut être important de savoir dans quel ordre sont enregistrées les valeurs d'un dictionnaire. On peut utiliser la classe `OrderedDict` pour cela. Elle se comporte exactement comme un dictionnaire, mais garantit de fournir les valeurs dans l'ordre dans lesquelles il les a reçues.

[165]:
```
od = collections.OrderedDict()
od['key1'] = 'value1'
od['key2'] = 'value2'

for key, value in od.items():
    print('{}: {}'.format(key, value))
```

```
key1: value1
key2: value2
```

La classe `DefaultDict` fonctionne comme un dictionnaire, mais possède une valeur par défaut, qui est retournée si on accède à une clé inexistante de l'itérable. La valeur par défaut est un *callable* pour pouvoir personnaliser plus en profondeur son comportement.

[166]:
```
dd = collections.defaultdict(lambda: 'default value')
dd['a'] = 'a value'
print(dd['a'])
print(dd['b'])
print(dd['c'])
```

```
a value
default value
default value
```

8.7 Manipulation de fichiers

Outre la fonction *built-in* `open`, le module `os` permet de manipuler des fichiers, et des systèmes de fichiers : il est capable de lister des dossiers, supprimer des fichiers, vérifier la présence de fichiers, ou encore de manipuler les chemins de fichiers en abstraction avec le système d'exploitation (car la façon de représenter un chemin peut varier selon le système d'exploitation : sur un système GNU/Linux ou UNIX, on peut avoir `/path/to/file`, mais sur Windows `C:\path\to\file`)

[167]:
```
import os

print(os.listdir('files'))
```

```
['chat.jpg', 'my_file', 'my_file_2', 'note.xml']
```

[168]:
```
os.mkdir('files/test')
print(os.path.isfile('files/test'))
print(os.path.isdir('files/test'))
os.rmdir('files/test')
```

```
False
True
```

[169]:
```
print(os.stat('stdlib.ipynb'))
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-169-fe123cd958d4> in <module>()
----> 1 print(os.stat('stdlib.ipynb'))

FileNotFoundError: [Errno 2] No such file or directory: 'stdlib.ipynb'
```

[170]:
```
file_path = os.path.join('files', 'test')
print(file_path)
```

```
files/test
```