Python, programmation objet

« It was nice to learn Python: a nice afternoon »

- D. Knuth, Trento, 2012

Darko Stankovski

Avertissements

Ce document est un support de formation Python. Il est destiné à être utilisé lors de la formation et n'est pas une référence du langage Python à lui tout seul.

version 1709

Présentation

Historique

Python est un langage créé en 1989 par Guido Van Rossum alors au Centrum voor Wiskunde en Informatica (Amsterdam, Pays-Bas). Fan des Monthy Pythons, il baptise ce projet Python. La première version publique est publiée sur le forum Usenet all.sources en 1991 en tant que version 0.9.0.

La fondation Python (Python Software Foundation), association sans but lucratif, est fondée en 2001.

La version 3 du langage est publiée en 2008. Celle-ci casse la compatibilité ascendante mais cette mise à jour était nécessaire pour réparer certains défauts du langage accumulés avec le temps.

En 2005, Guido Van Rossum rejoint Google où il travaille sur la plate-forme AppEngine. Il quitte Google pour Dropbox en 2012.

Guido Van Rossum est *Benevolent Dictator for Life* du langage Python.

Qui utilise Python

Une liste non exhaustive est disponible sur le wiki Python :

https://wiki.python.org/moin/OrganizationsUsingPython

On peut citer :

- Google (Google spider, search engine, Google app engine), YouTube (pipeline et front)
- Red Hat (Anaconda)
- Walt Disney Feature Animation et ILM
- La NASA

D'autres acteurs peuvent être cités, principalement des start-up qui ont adopté Python pour la productivité que permet ce langage. Parmi ceux-ci :

- Instagram
  Voir : https://www.quora.com/What-programming-languages-are-used-at-Instagram
- Pinterest
  Voir : https://www.quora.com/What-is-the-technology-stack-behind-Pinterest-1
- EVE Online
  Basé sur Stackless Python, voir : https://fr.wikipedia.org/wiki/Stackless_Python

Caractéristiques du langage

Python est un **langage interprété**. Il n'y a donc pas de phase de compilation, un code écrit peut être exécuté dans la foulée.

Pour ce faire, il nécessite un interpréteur. L'interpréteur rend ce langage multiplate-forme, que ce soit des systèmes d'exploitation (Linux, Windows ou Os X) ou des environnements logicielles (Java via Jython ou .Net via IronPython). Il existe des interpréter spécialisés comme PyPy qui est orienté performance au prix d'une rupture de la compatibilité.

Python est un langage de **haut niveau** (le développeur n'a pas à gérer la mémoire) **multi-paradigmes**. Il est **non déclaratif** mais **fortement typé**.

Boite à outils

Ce chapitre présente les outils nécessaires pour un développement Python, la manière de se les procurer et de les configurer.

L'interpréteur

L'interpréteur est le seul outil indispensable. Lorsqu'il est proposé par défaut sur un système, c'est malheureusement la version obsolète 2.7. Il faudra donc l'installer.

- Pour Windows et Mac OS, télécharger la dernière version de Python via l'URL https://www.python.org
- Pour Linux, une version à installer est disponible sur python.org. Préférez cependant la gestion par les paquets.

Gestionnaire de packages : Pip

Pip est le gestionnaire de packages Python recommandé par PyPA[1]. Il permet d'installer les packages avec leurs dépendances, mettre à jour, désinstaller et même recréer un environnement de développement.

Pour toute information : https://pypi.python.org/pypi/pip

L'installation de pip dépend du système.

- Sous **Windows** ou **Os X**, pip peut être installé lors de l'installation de Python. Vérifiez que la case a été cochée…
- Sous **Linux**, préférez le gestionnaire de packages

L'usage de pip dans le contexte de la formation est détaillé dans la section suivante.

iPython, l'interpréteur interactif évolué

iPython est un interpréteur interactif *évolué*. Il permet l'introspection (et donc la complétion), l'accès au shell système et propose ses propres *commandes magiques*.

Pour l'installation : `pip install ipython`

Plus d'informations : http://ipython.org

Documentation : http://ipython.readthedocs.io/en/stable/

Virutalenv

Virtualenv est la réponse au besoin de gérer plusieurs environnements sur un même système. Virtualenv permet de compartimenter et donc d'isoler les environnements de travail les un des autres.

Virtualenv s'installe via pip

Environnement de développement

L'usage d'un environnement de développement n'est pas nécessaire pour Python. Il est possible d'utiliser un simple éditeur de texte en s'assurant qu'il est configuré en UTF-8. Un IDE (Environnement de Développement Intégré) apportera tout de même certaines facilitées.

Il existe un certain nombre d'environnements intégrés plus ou moins complets. La liste suivante n'est pas exhaustive.

[1] Python Packaging Authority : https://www.pypa.io/

IDLE

IDLE est un environnement développement inclus dans le paquet Python. Il est donc toujours disponible. IDLE est écrit en Python à l'aide de Tkinter.

Eclipse + PyDev

PyDev est le plug-in pour Eclipse qui permet de développement en Python. Il est assez complet.

PyCharm

PyCharm est un IDE proposé par la société JetBrains à qui on doit IntelliJ Idea. PyCharm est un logiciel qui existe sous deux licences, une commerciale et une *community*. Cette seconde est gratuite et largement suffisante.

https://www.jetbrains.com/pycharm/download/

Autres outils

Anaconda

Destiné à l'écosystème scientifique, Anaconda propose un environnement complet comprenant l'interpréteur Python et des bibliothèques pré-compilées pour les environnements Windows et Mac Os. Il facilite ainsi la gestion de certaines bibliothèques comme MathPlotLib ou l'environnement Spyder.

https://www.continuum.io/

Jupyter

Le projet Jupyter notebook est un projet open-source d'application web qui permet de partager des pages contenant du code exécutable. Ce projet sera utilisé dans le cadre de la formation.

http://jupyter.org/

Python pour Windows ou Mac Os

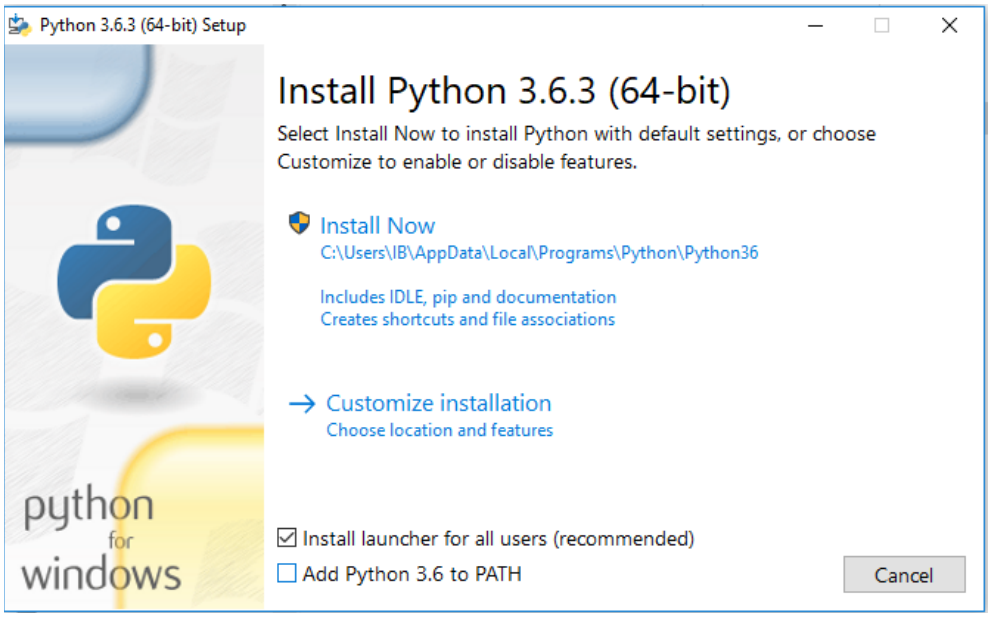
Sur Windows, Python n'est pas installé par défaut. Sur Mac Os, Python est installé en version 2.7. il est donc nécessaire dans les deux cas d'installer Python dans sa version 3.x

Récupérez la dernière version de Python sur https://www.python.org/

La manière la plus sûr d'installer Python est de passer par l'installation avancée.

- Lancez l'installation et choisissez Installation avancée
- Assurez-vous que l'installation de pip soit activée
- Assurez-vous que l'installation crée les paths ou variables d'environnement.
- Lancez l'installation

Autrement, le plus important sous Windows est cocher la case de l'ajout au PATH qui est décochée par défaut.



Fenêtre d'installation, sous Windows, cochez la case Add Python to PATH.

Python pour Linux (CentOs)

En fonction des distributions, Python peut être installé en version 3. Dans le cas où seul la version 2 est présente, suivez les instructions suivantes.

Installez Python à partir du centre des paquets. Les instructions suivantes sont destinées à un système Red Hat ou dérivé adaptés à la plupart des Linux de formation qui sont des CentOS :

```
su -
yum install python34
yum install python34-pip
yum install python34-tkinter
python3 -m pip install --user ipykernel
python3 -m ipykernel install --user
```

La première instruction vous donne les privilèges super-utilisateur et nécessite le mot de passe root. Cette instruction est nécessaire pour installer les paquets. Ces instructions installent les outils Python dans la version 3.4 qui est celle disponible sous CentOs à l'écriture de ce document.

CentOs étant une distribution destinée aux serveurs, les interfaces graphiques ne sont pas incluses dans les paquets. C'est pour cela qu'il est nécessaire d'installer tkinter séparément sachant que tkinter est autrement disponible avec l'installation standard de Python.

Vérifiez l'installation de Python

Pour vérifier que l'installation s'est bien passé, nous allons utiliser un terminal (ou cmd pour Windows). Dans le terminal, si il n'y avait aucune version se Python précédemment installée, exécutez les instructions suivantes :

```
python --version
pip --version
```

Si une version précédente était installée, exécutez

```
python3 --version
pip3 --version
```

Notez que pip est écrit en Python. L'exécutable est donc un *raccourci* pour l'une ou l'autre des instructions suivantes :

```
python -m pip --version
python3 -m pip --version
```

Installation de PyCharm

Les illustrations de cette formation reposeront sur l'IDE PyCharm. PyCharm propose une forte intégration de nombreux outils. Récupérez la version de PyCharm Community correspondant à votre système sur https://www.jetbrains.com/pycharm/download/ .

Lancez l'installation, vous pouvez utiliser les sélections par défaut.

Récupérez les sources du projet.

Cette formation est accompagnée de sources disponibles sur le service d'hébergement GitHub. Ces sources sont disponibles à l'adresse https://github.com/darko-itpro/training-python. Pour récupérer une version sur votre poste, choisissez le *bouton* Clone or download.

Vous avez deux options pour récupérer ces sources : télécharger une archive ou cloner le référentiel. Pour télécharger une archive, choisissez Download as ZIP.

Cloner le référentiel nécessite Git sur votre poste. Vous pouvez tester la présence de git via l'instruction suivante saisie dans un terminal :

```
git --version
```

Si git est absent, vous pouvez le récupérer et l'installer à partir de https://git-scm.com/ .

L'avantage de cloner le référentiel est que si durant la formation, des évolutions sont apportées aux sources, vous pourrez les mettre à jour. Vous pourrez aussi créer une branche et gérer les évolutions de votre code durant la formation.

Cette formation n'est pas une formation à git, aussi, le choix vous est laissé. Si vous souhaitez utiliser Git sans connaitre les instructions, l'usage le plus simple est de passer par son intégration dans PyCharm.

À partir de la fenêtre de démarrage de PyCharm, choisissez l'option . Dans le formulaire suivant

Installation les dépendances

Le projet de référence utilise quelques dépendances, c'est à dire des bibliothèques qui ne sont pas proposées dans l'installation standard. Nous allons évidemment utiliser pip pour les récupérer. L'utilitaire pip est capable s'installer toutes les dépendances d'un projet grâce à un fichier, le pip requirement file. Ce type de fichier s'appelle par convention requirements.txt et est situé à la racine du projet. Vous pouvez consulter ce fichier qui est un fichier texte.

Pour installer les dépendances du projet à partir de ce fichier, dans un terminal (ou cmd), déplacez-vous jusqu'au répertoire contenant le fichier requirements. Exécutez ensuite l'une des instructions suivante :

`pip3 install -r requirements.txt` si une précédente version était installée.

`pip install -r requirements.txt` si c'est la seule version installée

`sudo pip3 install -r requirements.txt` si vous êtes sur un système Linux et que vous n'êtes plus root et que une version de pip était installée pour Python 2.

L'utilitaire pip va télécharger toutes les dépendances listées dans ce fichier ainsi que leurs dépendances.

Introduction : manipulation et visualisation de données

La programmation est destinée à manipuler des données. Ces données peuvent être des informations fournies au programme et en fonction du résultat de cette manipulation, le programme peut nous retourner une information.

Python nous permet d'exécuter des instructions à l'aide de l'interpréteur intéractif. Vous pouvez faire l'essai et voir le résultat

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 3 + 5
8
```

L'interpréteur interactif est disponible avec toute installation de Python, cependant, il est assez limité dans son usage. Les développeurs Python lui préfèrent iPython qui en soi est une *surcouche* de l'interpréteur standard.

```
Python 3.6.0 (v3.6.0:41df79263a11, Dec 22 2016, 17:23:13)
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hello World")
Hello World

In [2]: 3 + 5
Out[2]: 8
```

Dans le cadre de la formation, nous utiliserons également les Jupyter Notebooks qui permettent de conserver le code qui nous servira d'illustration.

Traitement et retours des instructions

Nous avons vu une instruction, `print()`, qui est une fonction. Cette instruction affiche la donnée fournie entre parenthèses à la ligne suivante.

```
In [1]: print("Hello world")
Hello world
```

Python permet de réaliser des opérations mathématiques. L'opérateur + permet d'additionner deux valeurs. Lorsque vous exécutez une instruction d'addition dans le shell interactif ou iPython, le résultat s'affiche à la ligne suivante.

```
In [2]: 3 + 5
Out[2]: 8
```

Si vous exécutez ces instructions dans un notebook Jupyter, chaque instruction dans sa cellule, vous aurez le même résultat. Mais si vous exécutez dans une même cellule l'addition suivie de l'affichage, le résultat de l'addition n'est plus affiché.

```
In [2]: 3 + 5
        print("Hello world")

        Hello world
```

Et c'est tout à fait normal : l'instruction d'addition n'est pas une instruction d'affichage dans un terminal.

L'addition additionne les deux valeurs et produit une nouvelle donnée, la somme de ces deux valeurs. L'instruction retourne alors cette valeur à l'interpréteur pour qui elle devient une valeur pour l'opération suivante. Vous connaissez déjà ce principe avec la décomposition des opérations mathématiques :

```
2 * ( 3 + 5)
2 * 8
16
```

Dans cette opération à deux opérateurs où on impose la priorité, l'opération d'addition est exécutée et son résultat devient un argument pour l'opération suivante.

Lorsque la dernière opération est exécutée, la valeur produite est simplement *perdue*.

Il y a cependant un comportement particulier dans l'interpréteur interactif (et donc iPython et les notebooks Jupiter). Cette dernière valeur produite est *retournée* par l'interpréteur au terminal. Cela ressemble à un affichage, mais avec iPython, nous pouvons voir une différence entre les deux (voir les instructions 1 et 2 ci-dessus) : lorsqu'iPython affiche le retour de la dernière instruction, cette donnée est précédée d'un `Out [ligne] :`.

Le comportement est en fait similaire avec les notebooks, ceux-ci reposant sur iPython, à la différence que seul le retour de la dernière ligne est affiché.

Ne confondez donc pas avec les interpréteur interactifs les affichages et les retours des fonctions. L'addition est une instruction qui retourne une donnée, la fonction `print()` est une fonction qui affiche une donnée mais ne retourne rien (pas de ligne commençant par `Out`).

Conserver une donnée : les variables

Afin d'utiliser plus tard une donnée retournée par une instruction, il faudra conserver une *référence* sur celle-ci (attention, il n'est pas question de références et pointeurs ici). L'usage le plus classique implique les *variables*.

Une variable est un nom auquel nous allons associer une donnée. Cette association est réalisée par l'opérateur d'affectation `=`. Nous pouvons donc aussi bien affecter une donnée à une variable qu'une donnée résultant d'une opération.

```
In [1]: answer = 42

In [2]: result = 3 + 5
```

Vous remarquez que l'affectation est une instruction qui n'a pas de retour. Pour l'instruction 2, le résultat retourné par l'addition est une donnée pour l'opérateur d'affectation.

Pour l'interpréteur, une variable retourne son contenu (lignes 3 et 4). Puisqu'une variable retourne la donnée, elle peut être utilisée là où une instruction attends une donnée (lignes 5 et 6).

```
In [3]: answer
Out[3]: 42

In [4]: result
Out[4]: 8

In [5]: print(result)
8

In [6]: 2 * result
```

Les variables `result` et `answer`[2] contiennent donc une donnée suite à l'affectation d'une valeur ou d'un retour d'une instruction. Nous pouvons essayer d'affecter à une variable le retour d'une instruction de type `print()`.

```
In [7]: print_result = print("hello world")
hello world

In [8]: print_result

In [9]: print(print_result)
None
```

[2] voir : Le Guide du voyageur galactique, Douglas Adams (1978), ISBN:978-2070437436

Après l'instruction 7, nous avons l'affichage attendu de la fonction `print()`. La ligne 8 montre que, contrairement aux variables `answer` et `result` aux lignes 3 et 4, `print_result` n'a aucun retour. La ligne 9 affiche le contenu de la variable `print_result` qui est donc un type `None`.

Les bases

Dans cette partie, nous allons voir comment manipuler des données.

Données et variables

En Python, les variables **référencent** des données. Une variables ne contient pas la valeur mais une référence vers une valeur. En Python, nous ne déclarons pas le type des données référencées par les variables. **Le type est déterminé dynamiquement** par l'interpréteur.

Python est un langage de haut niveau qui va gérer la mémoire par un **mécanisme de comptage de références**. Le développeur n'a pas à se soucier de la gestion de la mémoire. Les données seront détruites lorsqu'elles ne seront plus accessibles, c'est à dire lorsqu'elles ne seront plus référencées.

```
In [10]: # Affectation d'une donnée à une variable
In [11]: ma_variable = 10
In [12]: print(type(ma_variable))
<class 'int'>
In [13]: del ma_variable
In [14]: # la variable ma_variable n'existe plus et la donnée a été
    ...: supprimée
```

Règles de nommage

Les variables en Python peuvent contenir un caractère alphanumérique sans diacritique[3] (tel que les accents, cédille…) ou un caractère souligné _. Le premier caractère ne peut pas être un caractère numérique (de 0 à 9). Python est sensible à la casse (capitale ou minuscule).

L'expression rationnelle représentant un nom de variable Python peut s'écrire **[a-zA-Z_][a-zA-Z0-9_]***

En plus de la règle de nommage, vous ne pouvez pas utiliser les mots réservés dont voici la liste.

```
False       class       finally     is          return
None        continue    for         lambda      try
True        def         from        nonlocal    while
and         del         global      not         with
as          elif        if          or          yield
assert      else        import      pass
break       except      in          raise
```

[3] voir : https://fr.wikipedia.org/wiki/Diacritique

Conventions de codage : la PEP 8

Tout développement nécessite de s'accorder sur des *conventions de codage* afin de rendre l'ensemble du code lisible et maintenance. Dans l'univers Python, ces règles et conventions sont généralisés dans la **PEP 8**[4]. Une **PEP**, pour **Python Enhancement Proposal**, est un document destiné à proposer une évolution du langage puis, si accepté, à documenter cette évolution. L'explication plus détaillée est publiée dans la PEP 1[5] et la liste de toutes les PEPs dans la PEP 0[6].

La PEP 8 est le document formalisant les règles de codage adoptées par la communauté. La PEP 8 propose des règles sur le choix des noms à adopter ainsi que leur typographie. Par exemple :

- Évitez les caractères l, O ou I comme nom de variable à une lettre (en général des indices) car en fonction des polices, ils ne sont pas lisibles.
- Lettres minuscules pour les noms de modules, variables et fonctions. Pour représenter plusieurs mots, les séparer par le caractère souligné (exemple : `ma_variable`).
- Lettres capitales et souligné pour les constantes (exemple : `MA_CONSTANTE`)
- Camelcase (ou CapWords) pour les noms de classes (exemple: `MaClasse`)

[4] voir : https://www.python.org/dev/peps/pep-0008/

[5] voir : https://www.python.org/dev/peps/pep-0001/

[6] voir : https://www.python.org/dev/peps/

Affecter des données à des variables

L'opérateur égal = permet d'affecter une donnée à une variable. Il y a plusieurs manières d'utiliser cet opérateur comme illustré dans le code ci-après :

- l'affectation simple (ligne 1)
- l'affectation multiple (la même donnée est référencée par plusieurs variables (ligne 2)
- l'affectation en parallèle (ligne 3 et 4)
- l'assignation de valeurs consécutives (ligne 5)

```
In [1]: var = 42
In [2]: var1 = var2 = 42
In [3]: question, answer = 42, "Meaning of life"
In [4]: question, answer = answer, question
In [5]: var1, var2, var3 = ["meaning", 42, None]
```

Pour une question de lisibilité et de maintenance, **privilégiez l'affectation simple**. Utilisez l'affectation en parallèle pour inverser le contenu de variables comme l'illustre la ligne 4.

L'utilité de l'assignation par valeurs consécutives prend tout son sens avec le retour de valeurs multiples d'une fonction.

Manipuler et comprendre des données

Python propose plusieurs outils permettant de comprendre les variables et données manipulés. Soit une variable `ma_variable` :

- `print(ma_variable)` affiche la valeur référencée par la variable
- `print(type(ma_variable))` affiche le type de la valeur référencée par la variable
- `dir(ma_variable)` retourne la liste des attributs d'une variable, et plus généralement, d'un type
- `help(ma_variable)` affiche *l'aide* sur une variable ou plutôt du type de la variable.

Les types de base (built-in)

Les types numériques

En Python, nous avons 3 types numériques : `int` qui représente les entiers, `float` pour les réels et `complex` pour les nombres complexe.

Il existe également le type booléens (`bool`) qui est un un sous-type des entiers.

Python est un langage non déclaratif, c'est à dire que vous ne déclarez pas le type de la donnée. Cependant, si vous devez manipuler un donnée d'un autre type que celle déduite par l'interpréteur, il faut la convertir grâce aux fonctions correspondantes : `int(var)`, `float(var)` ou `complex(var1, var2)`.

Ci-dessous, vous avez un exemple de déclaration de types numérique

```
In [10]: var_int = int(42.2)
In [11]: var_int = 42

In [12]: var_float = float(3)
In [13]: var_float = 3.
In [14]: var_float = 3.14

In [15]: var_cpx = complex(42, 2)
In [16]: var_cpx = 42 + 2j
In [17]: var_cpx.real
Out[17]: 42.0
In [18]: var_cpx.imag
Out[18]: 2.0
```

En Python, il n'y a pas de taille maximum des entiers. Si sa valeur est comprise entre **-2^(n-1)** et **2^(n-1)-1**, il est géré en registre, sinon en mémoire.

Opérations sur les types numériques

Python propose les opérateurs mathématique classiques.

x + y	Addition
x - y	Soustraction
x * y	Multiplication
x / y	Division
x // y	Division entière
x % y	Reste
-x	Opposé
+x	
x ** y	Puissance

La priorité des opérateurs est la priorité mathématique. À priorité égale, les opérations sont résolues de gauche à droite. Les parenthèses ont la plus grande priorité et permettent de grouper les opérations.

Les opérateurs binaires

Python permet des opérations binaires.

x | y	Ou binaire
x ^ y	Ou exclusif
x & y	Et binaire
x << y	Décalage à gauche
x >> y	Décalage à droite
~ x	Inversion

Exemple :

```
In [1]: x = 5

In [2]: y = 6

In [3]: res = x | y

In [4]: print(res)
7
```

5 en binaire est `0b101`

6 en binaire est `0b110`

Donc `0b101` OU BINAIRE `0b110` donne `0b111` soit 7.

Les séquences

La notion de *séquences* regroupe les types qui représentent des *collections*. Python propose plusieurs types de base : les chaines de caractères (`string`), les listes, les tuples, les ensembles (`set`) et les dictionnaires.

String, list et tuples fonctionnent de manière très similaire. Le tableau suivant regroupe certains opérateurs et les deux méthodes qui sont communs ces trois séquences. Dans ce tableau, s est une séquence, x un élément de la séquence, n et i des entiers.

x in s	True si s contient x, sinon False
x not in s	False si s contient x, sinon True
s1 + s2	Concaténation
s * n	Répétition
s[i]	Élément à l'indice ou clef i
len(s)	Taille de la chaine
min(s)	Plus petit élément de la séquence
max(s)	Plus grand élément de la séquence
s.index(x)	Indice de la première occurence de x
s.count(x)	Nombre total d'occurrences de x

Les chaines de caractères

Les chaines de caractères sont délimitées par des simples, doubles ou triple-double-quotes. Ces derniers sont des string-literais dont le contenu est échappé. Les chaines de caractères sont **immuables** et sont encodées en **Unicode**.

Dans l'exemple suivant, les lignes 11 et 12 sont équivalentes. La ligne 13 vous montre comment définir un menu d'une interface en ligne de commande.

```
In [11]: question = 'Meaning of life'
In [12]: question = "Meaning of life"
In [13]: choice = """Choose:
    ...:  1) first choice
    ...:  2) second choice"""
    ...:

In [14]: print(choice)
Choose:
 1) first choice
 2) second choice
```

Les listes

Les listes sont des **séquences ordonnées d'objets**. Les listes **peuvent être modifiées**. Elles sont représentées entre crochets.

```
In [1]: knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galaad"]

In [2]: h2g2 = ["Meaning of life", 42]
```

Les tuples

Les tuples sont des **séquences ordonnées d'objets**. Les tuples **ne peuvent pas être modifiées**. Ils sont représentées entre parenthèses.

Si ils sont en général présentés entre parenthèses, c'est surtout la présence de la virgule qui fait le tuple. Ceci est d'autant plus important pour le tuple *singleton* pour lequel il ne faut pas oublier une virgule.

```
In [1]: knights = ("Arthur", "Lancelot", "Robin", "Bedevere", "Galaad")

In [2]: h2g2 = "Meaning of life", 42
In [3]: type(h2g2)
Out[3]: tuple

In [5]: tuple_singleton = 42,
In [6]: type(tuple_singleton)
Out[6]: tuple
```

Les types `str`, `list` et `tuple` possèdent un ensemble de méthodes qui permet de les manipuler. Vous pouvez consulter l'ensemble des méthodes par l'instruction `help(str)`, `help(list)` et `help(tuple)`. Vous avez ci-dessous un exemple d'utilisation des méthodes du type `str`.

```
In [10]: question = 'Meaning of life'

In [11]: print(question.upper())
MEANING OF LIFE

In [12]: print(question.split())
['Meaning', 'of', 'life']

In [13]: ".".join(question)
Out[13]: 'M.e.a.n.i.n.g. .o.f. .l.i.f.e'

In [23]: question.isalnum()
Out[23]: False
```

Ainsi que des méthodes pour le type `list`.

```
In [21]: knights = ["Arthur", "Lancelot"]

In [22]: knights.append('Robin')

In [23]: knights.extend(['Bedevere', 'Galaad'])

In [24]: knights.pop()
Out[24]: 'Galaad'

In [25]: knights.insert(2, 'Galaad')
In [26]: print(knights)
['Arthur', 'Lancelot', 'Galaad', 'Robin', 'Bedevere']

In [27]: knights.remove('Robin')
In [28]: print(knights)
['Arthur', 'Lancelot', 'Galaad', 'Bedevere']

In [29]: knights.reverse()

In [30]: knights.sort()
In [31]: print(knights)
['Arthur', 'Bedevere', 'Galaad', 'Lancelot']
```

Accès aux éléments

L'accès aux éléments d'une chaine, liste ou tuple se fait par l'indice de cet élément entre crochet. Python permet l'usage d'indices négatifs afin d'accéder aux éléments à partir de la fin.

```
In [30]: knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galaad"]
In [31]: antagonist = 'Killer Rabbit of Caerbannog'

In [32]: print(knights[2])
Robin

In [33]: print(antagonist[2])
l

In [34]: print(knights[-1])
Galaad

In [35]: print(antagonist[-5])
a
```

Affectation d'élément

Un élément peut être remplacé dans une liste en attribuant une valeur à son indice. La valeur à cet indice est alors remplacée par la nouvelle. Les chaines de caractères et les tuples sont immuables, un de leur élément ne peut pas être remplacé.

```
In [50]: knights = ["Arthur", "Robin", "Bedevere", "Galaad"]

In [51]: knights[1] = 'Lancelot'
In [52]: print(knights)
['Arthur', 'Lancelot', 'Bedevere', 'Galaad']

In [53]: antagonist = 'Killer Rabbit of Caerbannog'

In [54]: antagonist[0] = 'M'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-677a915eb39c> in <module>()
----> 1 antagonist[0] = 'M'

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

Slicing

Le slicing permet d'extraire des sous-séquences. L'opérateur de slicing est le suivant (`seq` étant une séquence de type `str`, `list`, ou `tuple`) :

- `seq[x:y]` : extrait une séquence de l'indice x à l'indice y exclu
- `seq[:y]` : extrait une séquence de l'indice 0 à l'indice y exclu
- `seq[x:]` : extrait une séquence de l'indice x à la fin de la séquence seq
- `seq[x:y:z]` : extrait une séquence de l'indice x à l'indice y exclu avec un pas de z. x et y sont donc optionnels.

Si les bornes sont en dehors des limites de la séquence, le slicing retournera une séquence vide.

```
In [50]: knights = ["Arthur", "Lancelot", "Robin", "Bedevere", "Galaad"]

In [51]: knights[2:4]
Out[51]: ['Robin', 'Bedevere']

In [52]: knights[2:]
Out[52]: ['Robin', 'Bedevere', 'Galaad']

In [53]: knights[:2]
Out[53]: ['Arthur', 'Lancelot']

In [54]: antagonist = 'Killer Rabbit of Caerbannog'

In [55]: antagonist[::2]
Out[55]: 'Kle abto arang'
```

Les ensembles (set)

Le type set permet de gérer des ensembles. Il s'agit d'une collection non ordonnée d'éléments non redondants.

La fonction set prend en paramètre une séquence de laquelle elle supprime les redondances potentielles.

Les sets sont destinés aux opérations ensemblistes.

```
In [51]: camelot = ['Arthur', 'Merlin', 'Lancelot', 'Robin']

In [52]: wizards = ['Merlin', 'Morgan']

In [53]: camelot_set = set(camelot)

In [54]: camelot_set.union(wizards)
Out[54]: {'Arthur', 'Lancelot', 'Merlin', 'Morgan', 'Robin'}

In [55]: camelot_set.intersection(wizards)
Out[55]: {'Merlin'}

In [56]: set(wizards).issubset(camelot_set)
Out[56]: False
```

Les dictionnaires

Les dictionnaires sont des séquences non-ordonnées de couples clef/valeur. Il n'y a donc pas de notion de position. Dans un dictionnaire, chaque clef est unique. La valeur est obtenue à partir de la clef. Clef et valeur sont des objets.

Un dictionnaire peut-être créé vide (lignes 10 et 11) ou avec des valeurs. La création de dictionnaire avec des valeur la plus courante est celle de la ligne 14.

```
In [10]: mydict = {}
In [11]: mydict = dict()

In [12]: mydict = dict(question='Meaning of life', answer=42)
In [13]: mydict = dict([['question', 'Meaning of life'], ['answer', 42]])

In [14]: mydict = {'question': 'Meaning of life', 'answer': 42}
```

Pour ajouter un couple clef/valeur ou modifier la valeur d'une clef, on utilise l'opérateur crochet en spécifiant la clef entre crochets et en affectant une valeur. Le même opérateur est utilisé pour accéder à une valeur par la clef. Si la clef n'existe pas, une erreur de type KeyError est levée

```
In [71]: mydict = {'question': 'Meaning of life', 'answer': 42}

In [72]: mydict['reference'] = 'Hitchhicker guide'

In [73]: mydict['reference']
Out[73]: 'Hitchhicker guide'

In [74]: mydict['reference'] = 'H2G2'

In [75]: mydict['reference']
Out[75]: 'H2G2'

In [76]: mydict['nothing']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call
last)
<ipython-input-76-584c9c422e37> in <module>()
----> 1 mydict['nothing']

KeyError: 'nothing'

In [77]: mydict['reference'] = None

In [78]: print(mydict['reference'])
None
```

Il est possible de tester la présence d'une clef dans le dictionnaire avec l'opérateur `in`. Lorsqu'une valeur est testée via cet opérateur `in`, cette valeur est recherchée parmi les clefs du dictionnaire.

```
In [79]: 'answer' in mydict
Out[79]: True

In [80]: 'nothing' in mydict
Out[80]: False
```

Le type `dict` propose quelques méthodes utiles. On citera :

- `dict.get(key[, default])` : retourne la valeur de key ou défault. Si default n'est pas spécifié, retourne `None`.
- `dict.items()` : retourne un objet similaire à un set contenant la liste des couples clef/valeur
- `dict.values()` : retourne un objet la liste des valeurs
- `dict.clear()` : vide le dictionnaire
- `dict.copy()` : retourne une copie du dictionnaire
- `dict.pop(key)` : supprime le couple clef/valeur correspondant à la clef key et retourne la valeur.

Interaction avec l'utilisateur

Maintenant que nous savons manipuler les chaines de caractères, nous pouvons ajouter un peu d'interaction dans nos programmes. Il s'agira de poser des questions dans le terminal et de mettre en forme des résultats.

Saisie de l'utilisateur

Demander une saisie à l'utilisateur se fait à l'aide ce la fonction `input(prompt=None)`. Cette fonction lit une chaine de caractères dans l'entrée standard et la retourne. Le paramètre `prompt` est une chaine de caractères optionnelle qui, si elle est présente, sera affichée.

L'usage consiste à récupérer le retour de cette fonction au sein d'une variable. Attention, le retour est toujours une chaine de caractères, si un autre type est attendu, il faudra transtyper ce retour.

```
In [20]: input()
42
Out[20]: '42'

In [21]: answer = input("Meaning of life ")
Meaning of life 42
In [22]: print(answer)
42
In [23]: type(answer)
Out[23]: str

In [24]: answer = int(input("Meaning of life "))
Meaning of life 42
In [25]: type(answer)
Out[25]: int
```

Affichage et mise en forme

L'affichage dans le terminal est réalisé par la fonction `print(…)`. Vous pouvez passer plusieurs paramètres à la fonction print, ils seront affichés séparés par un espace.

Afin de mettre en forme un résultat sous forme d'une chaine de caractères de manière lisible et maintenable, Python propose 3 outils :

- L'opérateur `%`, aussi appelé le *printf-style*, aujourd'hui déconseillé
- la méthode `format(…)` des chaines de caractères
- les *string literals* depuis Python 3.6

Ces trois outils produisent une chaîne de caractères. Utilisés en paramètre de la fonction print(…), c'est donc la chaîne de caractères résultantes qui est le paramètre de la fonction. Ils faut également comprendre qu'ils ne sont pas spécifiques à l'affichage dans un terminal.

Printf-style

Cet usage est documenté à l'adresse :

https://docs.python.org/3/library/stdtypes.html#old-string-formatting

La manière historique de formater une chaine de caractère est héritée de la fonction `sprintf` du C. Elle est aujourd'hui connue sous le nom de *old-style string formatting* ou *interpolation*. Elle met en œuvre l'opérateur modulo `%`. Son usage général est :

template % values

template est une chaine de caractères contenant des spécifications de conversion avec le symbol `%`. *values* est soit une valeur soit un tuple contenant plusieurs valeurs. *values* doit contenir autant de valeurs que *template* contient de spécifications de conversion. Les exemples ci-dessous illustrent l'usage.

```
In [10]: 'The answer is %d' % answer
Out[10]: 'The answer is 42'

In [11]: 'The answer to "%s" is %d' % ('Meaning of life', answer)
Out[11]: 'The answer to "Meaning of life" is 42'

In [12]: 'numero abonné : %06d' % 244
Out[12]: 'numero abonné : 000244'

In [13]: import math

In [14]: print(math.pi)
3.141592653589793

In [15]: 'π est arrondi à %.2f' % math.pi
Out[15]: 'π est arrondi à 3.14'
```

La fonction format

La chaine à formater par la fonction format contient les champs à remplacer par des accolades `{}`. Ces champs seront replacés par les paramètres de la fonction `format()`. Ces accolades permettent de spécifier la mise en forme des valeur par un microlangage. La documentation de ce microlangage est disponible :

https://docs.python.org/3/library/string.html#formatstrings

Voici quelques exemples

```
In [20]: 'The answer is {}'.format(answer)
Out[20]: 'The answer is 42'

In [21]: print('Le prix est de {} {}'.format(42, '€'))
Le prix est de 42 €

In [22]: print('The price is {1} {0}'.format(42, '$'))
The price is $ 42

In [23]: print('Le prix est de {:.2f} {}'.format(42, '€'))
Le prix est de 42.00 €

In [24]: for price in [42.5766, 12, 1250]:
    ...:     print('Le prix est de {:8.2f} {}'.format(price, '€'))
    ...:
Le prix est de    42.58 €
Le prix est de    12.00 €
Le prix est de  1250.00 €
```

Les String Literals

Python 3.6 a ajouté la notion de **String Literals** ou **f-string** : vous pouvez déclarer dans la chaine, préfixée par un `f` ou `F`, un nom de variable qui sera remplacé par la valeur. Une documentation est disponible à l'adresse suivante :

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

```
In [1]: f"The answer is {answer}"
Out[1]: 'The answer is 42'

In [2]: import math
In [3]: f"π: {math.pi:{5}.{3}}"
Out[3]: 'π:  3.14'
```

Les structures de contrôle

Les *structures de contrôle* servent à **structurer** l'exécution du code. Elles permettent de définir des **blocs** qui vont être exécutés de manière **conditionnelle** ou **en boucle**.

Le bloc est un ensemble d'instructions qui est précédé d'une **déclaration**. Une déclaration est toujours terminée par deux points (`:`). Il n'y a pas de délimiteur comme les accolades en Python. Ce qui définira un bloc, c'est que les instructions seront **indentées** par rapport à la déclaration.

Les blocs

Les blocs sont donc un ensemble d'instructions exécutées en fonction de la déclaration qui le précède. Ces instructions sont indentées par rapport à la déclaration, c'est à dire qu'ils sont en retrait. Ce retrait est défini par la pep 8 comme étant de 4 espaces.

Veillez donc à vous assurer que votre éditeur de code convertisse les tabulations en 4 espaces.

L'instruction `pass`

L'instruction `pass` est une instruction que ne fait… rien. Après une déclaration de bloc (que ce soit une structure de contrôle, une déclaration de fonction ou de classe), l'interpréteur attends une ligne indentée. Si vous n'avez aucune instruction à déclarer dans le bloc (voir les cas lors de la formation), l'instruction `pass` permet de ne pas avoir d'erreur d'interprétation.

```
In [10]: if True:
    ...:     pass
    ...:
```

Exécution conditionnelle avec `if`

L'instruction `if` permet d'exécuter un bloc d'instructions de manière conditionnelle.

```
In [1]: if knight == "Arthur":
   ...:     print("Is the King")
   ...:
```

Une ou plusieurs conditions alternatives peuvent être déclarées avec l'instruction `elif`. L'instruction `else` permet de gérer tous les autres cas.

```
In [2]: if knight == "Arthur":
   ...:     print("Is the King")
   ...: elif knight == "Robin":
   ...:     print("He's not that brave")
   ...: else:
   ...:     print("Who's that lad ?")
   ...:
```

Les différentes conditions sont exclusives. À la première condition évaluée à vrai (`True`), l'interprèteur exécuté le bloc puis sort de la structure conditionnelle.

La condition, aussi appelée structure de contrôle, doit retourner une expression booléenne. Cette expression peut aussi bien être le retour d'une fonction que le résultat d'un opérateur de comparaison ou d'un opérateur logique.

Les opérateurs de comparaison

Les opérateurs de comparaisons entre deux déclarations sont :

<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Les opérateurs logiques

Les opérateurs logiques, nom utilisé pour les valeurs de vérité de l'algèbre de Bool, sont la conjonction (et), la disjonction (ou) et la négation (non). En Python, les mots-clef respectifs sont `and`, `or` et `not`.

Soient X et Y deux expressions qui retournent une valeur booléenne.

- Conjonction : X ET Y est vrai si et seulement si X est vrai et Y est vrai
  En python, elle s'écrit `X and Y`
- Disjonction : X OU Y est vrai si et seulement si X est vrai ou Y est vrai
  En python, elle s'écrit `X or Y`
- Négation : la négation de X est vrai si et seulement si X est faux
  En python, elle s'écrit `not X`

En python, les fonctions de conjonction et de disjonction sont des opérations *paresseuses*. Si l'évaluation du premier terme ne laisse aucun doute sur le résultat de l'opération (si X est faux pour une conjonction ou X est vrai pour une disjonction), le second terne n'est pas évalué. Gardez ceci à l'esprit si les expressions sont des appels de fonctions.

Les valeurs booléennes

Les types booléens ne sont pas les seuls types à avoir une valeur booléenne en Python. Le tableau suivant liste les valeurs de différents types qui ont également la valeur booléenne faux :

Type	FAUX
Bool	False
int	0
float	0.0
string	""
tuple	()
list	[]
dict	{}
None Type	None

Ne surchargez donc pas les tests en Python. Au lieu de

```
In [3]: if mon_tableau != None and len(mon_tableau) > 0:
   ...:     pass
   ...:
```

Préférez

```
In [4]: if mon_tableau:
   ...:     pass
   ...:
```

L'expression ternaire

L'expression ternaire sert à obtenir un résultat en fonction d'une condition. La structure de l'expression ternaire est la suivante :

si_vrai `if` condition `else` si_faux

En fonction de la condition, l'interpréter évaluera l'expression à gauche ou à droite et retournera le résultat de cette évaluation. L'expression ternaire est généralement utilisée pour simplifier l'écriture de l'affectation à une variable.

```
In [1]: status = "Retard" if delay < 10 else "annulé"
```

Un cas pratique est la configurations de variables en fonction de préférences utilisateur. Dans l'instruction suivante, si un paramètre utilisateur est indéfini, il est égal à None :

```
In [2]: pref = user_pref if user_pref else default_value
```

Boucles avec `for` : les itérations sur des séquences

En Python, l'instruction `for` sert à parcourir des séquences. Le bloc est donc un traitement appliqué à chaque élément de la séquence. L'instruction `for` est comparable à l'instruction *foreach* dans certains autres langages.

```
In [11]: for element in sequence:
    ...:     print(element)
    ...:
```

Pour itérer sur des indices, Python propose la fonction `range`. La fonction `range` est un générateur d'entiers. La fonction `range` peut accepter jusqu'à 3 paramètres qui seront la borne inférieur, la borne supérieur et le pas. Le code suivant illustre les 3 déclarations possibles de la fonction range et un usage pratique avec l'instruction `for`.

```
In [12]: range(10)
Out[12]: range(0, 10)
In [13]: list(range(10))
Out[13]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [14]: list(range(5, 20))
Out[14]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
In [15]: list(range(5, 20, 2))
Out[15]: [5, 7, 9, 11, 13, 15, 17, 19]

In [16]: for count in range(10):
    ...:     print(count, end='-')
    ...:
0-1-2-3-4-5-6-7-8-9-
```

Pour itérer sur une liste tout en ayant connaissance de l'indice de l'élément, utilisez la fonction `enumerate` qui retourne un tuple composé de l'élément suivant de la liste et son indice.

```
In [17]: for indice, element in enumerate(['Arthur', 'Lancelot',
    ...:                                   'Galahad']):
    ...:     print(indice, element)
    ...:
0 Arthure
1 Lancelot
2 Galahad
```

Boucles avec `while` : itération en fonction d'un état

L'instruction `while` permet de répéter le bloc tant que la condition est vrai. Il s'agit donc d'une itération en fonction d'un état.

Un exemple est l'affichage d'une suite jusqu'à une certaine valeur, ici une suite de Fibonacci[7] :

```
In [4]: a, b = 0, 1

In [5]: while b < 10:
   ...:     print(b)
   ...:     a, b = b, a + b
   ...:
1
1
2
3
5
8
```

Contrôle des boucles, `break`, `continue` et `else`

Python propose 3 instructions de contrôle des boucles :

- `break` interrompt la boucle et en sort
- `continue` interrompt la boucle et passe à l'itération suivante
- `else` permet de définir un bloc d'instructions qui ne sera exécuté après la boucle uniquement si il n'y a pas eu se sortie par un break.

Break et continue sont destinés à être déclarés dans les blocs à la suite à une condition.

```
In [9]: for knight in knights:
   ...:     if have_fount_the_Grail(knight):
   ...:         print("Job done")
   ...:         break
   ...:     if knight == "Robin":
   ...:         print("Such a coward…")
   ...:         continue
   ...:     assign_another_quest(knight)
   ...: else:
   ...:     print("We are gone for another round")
   ...:
```

[7] Voir : https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

Comprehension lists, les listes en intension

En logique, l'intension d'un concept[8] est sa définition.

L'objectif des listes en intension, inspiré de Haskell, est de faciliter le filtre d'une liste et/ou la transformation de ses éléments. De manière classique, la transformation des éléments se ferait de la manière suivante.

```
In [41]: knights = ['arthur', 'lancelot', 'galahad']

In [42]: cap_knights = []
In [43]: for knight in knights:
    ...:     cap_knights.append(knight.capitalize())
    ...:
```

Avec les listes en intension, elle sera la suivante.

```
In [51]: knights = ['arthur', 'lancelot', 'galahad']

In [52]: cap_knights = [knight.capitalize() for knight in knights]
```

La structure générale d'une déclaration d'une liste en intension est la suivante :

```
[transformation for element in sequence if condition]
```

Le mécanisme des listes en intension permet donc de créer une nouvelle liste en filtrant les éléments de la liste et/ou en les transformant.

```
In [53]: on_quest = [knight for knight in knights if is_on_quest(knight)]

In [54]: on_quest = [knight.capitalize()
                     for knight in knights
                     if is_on_quest(knight)]
```

[8] Voir : https://fr.wikipedia.org/wiki/Intension_et_extension

Les fonctions

Les fonctions permettent de grouper des instructions afin de les exécuter par un nom logique. Elles peuvent recevoir des paramètres et peuvent produit une valeur de retour.

Une fonction est déclarée par le mot clef def suivi du nom de la fonction et de parenthèses. Le corps de la fonction est un ensemble d'instructions indentées par rapport à cette déclaration. Pour appeler cette fonction, il suffit de déclarer son nom suivi de parenthèses.

```
In [1]: def count_to_three():
   ...:     print('one')
   ...:     print('two')
   ...:     print('three')
   ...:

In [2]: count_to_three()
one
two
three
```

Si la fonction attend des paramètres, ceux-ci doivent être déclarés dans les parenthèses. Tout comme les variables classiques, aucun type n'est déclaré pour ces paramètres. Le nom des paramètres définissent des variables qui devront être utilisées dans la fonction. Lors de l'appel de la fonction, il devra y avoir autant de valeurs dans les parenthèses que de paramètres attendus.

```
In [3]: def your_favorite_language(answer):
   ...:     if answer != "Python":
   ...:         print("Oh, cool, but try Python")
   ...:     else:
   ...:         print("Good Choice !")
   ...:

In [4]: your_favorite_language("Python")
Good Choice !
```

Une fonction peut retourner une donnée. L'instruction pour retourner une donnée est return suivi de la donnée. Lorsque l'interpréteur rencontre l'instruction return, il sort de la fonction en retournant la donnée. Les instructions suivantes ne seront donc pas exécutées. L'instruction return est optionnelle. Si elle est omise, la fonction retournera une donnée par défaut : None.

```
In [5]: def double(x):
   ...:     if x:
   ...:         return x * 2
   ...:     print('Other than None')
   ...:

In [6]: double(None)
Other than None
```

En Python, on ne déclare pas le retour (une fonction a toujours un retour). Il est possible de retourner plusieurs valeurs en les déclarant à la suite après l'instruction return. La fonction retournera alors un tuple contenant tous ces éléments.

```
In [7]: def multi_result():
   ...:     return 'Question', 42
   ...:

In [8]: res = multi_result()

In [9]: print(res[0], res[1])
Question 42

In [10]: res1, res2 = multi_result()

In [11]: print(res1, res2)
Question 42
```

Des arguments d'une fonction peuvent être optionnels. Pour qu'un argument soit optionnel, il faut lui attribuer une valeur par défaut dans la déclaration de la fonction. Si un argument est déclaré optionnel, tous les arguments suivants de la déclaration de la fonction doivent également être optionnels.

Lors de l'appel à une fonction, une valeur peut être attribuée à un paramètre de manière positionnelle ou par affectation spécifique au paramètre.

```
In [35]: def create_account(nid, value='100', allowed=1000):
    ...:     print(nid, value, allowed)
    ...:

In [36]: create_account('THX1138')
THX1138 100 1000

In [37]: create_account('THX1138', 500)
THX1138 500 1000

In [38]: create_account('THX1138', allowed=500)
THX1138 100 500
```

Fonctions variadics

Une fonction peut accepter un nombre non-prédéfini de paramètres. Afin d'offrir cette possibilité, il faut y dédier une variable caractérisée par l'opérateur $*$ communément appelé **splat**. L'opérateur splat peut être utilisé conjointement à d'autres paramètres si il est placé en dernier paramètre positionnel. Toutes les valeurs qui n'ont pas été affectés à un paramètre seront passées à celui-ci qui contiendra un tuple.

Il est possible de passer des variable sous la forme clef/valeur de manière similaire à l'affectation de valeur a un paramètre. Dans ce cas, il faut définir un autre paramètre conteneur préfixé par un double-splat $**$.

Communément, on utilise les noms `*args` et `**kargs` ou `**kwargs` (pour KeyWord). Ce nom n'est pas imposé mais suffisamment communs pour être identifiés de suite.

```
In [39]: def multi_args(*args, **kargs):
    ...:     for n in args:
    ...:         print(n)
    ...:     for k in kargs.keys():
    ...:             print(k, ":",  kargs[k])
    ...:

In [40]: multi_args('toto', 'titi', 42, answer='42')
toto
titi
42
answer : 42
```

Portée des variables

Pour comprendre la notion de portée des variable, il faut savoir que Python définit des espaces de nommage en fonction du contexte. On va appeler contexte global le niveau d'un module et contexte local celui des fonctions. Les instructions déclarées dans les fonctions s'exécutent donc dans le contexte de la fonction.

Une variable déclarée dans une fonction est donc déclarée dans l'espace local de la fonction et n'est visible que dans la fonction. Lorsque l'exécution de la fonction est terminée, cette variable est détruite.

Une variable déclarée en dehors de la fonction est visible dans la fonction à la condition qu'elle ai été déclarée avant. Cette variable ne sera visible au sein de la fonction qu'en lecture seule. Ainsi, si la fonction déclare une variable du même nom, cette variable sera une nouvelle variable déclarée dans l'espace local qui masquera celle du bloc supérieur.

Cette restriction de visibilité existe pour éviter une altération silencieuse des données du bloc appelant. La bonne pratique pour modifier des données suite à un appel de fonction est de récupérer les données par retour de fonction et les affecter explicitement.

Python offre la possibilité d'accéder en écriture a une variable globale lorsque cela est indispensable. Pour cela, il faut déclarée dans la fonction la variable par la directive global.

```
In [1]: answer = 42

In [2]: def func():
   ...:     global answer
   ...:     answer == 1
   ...:

In [3]: func()

In [4]: print(answer)
42
```

La directive global réfère au scope du module. Il existe une autre directive, nonlocal, qui réfère un scope supérieur à l'exclusion du global.

Les fonctions sont des objets

Une fonction est définie comme étant un callable. Lorsque vous utilisez une fonction, vous y faites appel par son nom suivi de parenthèses. La présence de ces parenthèses indiqué à l'interprèteur qu'il doit exécuter la fonction et retourner le résultat.

Si vous déclarez le nom d'une fonction sans le faire suivre des parenthèses, alors vous manipulée une référence vers la fonction. Pour rappel, les variables contiennent des références. Une référence vers une fonction peut donc être attribuée à une variable. La variable devient alors elle même *callable*. C'est la base de la programmation fonctionnelle.

```
In [1]: def apply_base_tva(value):
   ...:     return value * 1.2
   ...:

In [2]: apply_base_tva
Out[2]: <function __main__.apply_base_tva>
```

Une référence à une fonction est une donnée qui peut donc être utilisée comme n'importe quelle donnée. Elle peut être affectée à une variable.

```
In [3]: my_tva = apply_base_tva

In [4]: my_tva
Out[4]: <function __main__.apply_base_tva>

In [5]: my_tva(42)
Out[5]: 50.4
```

Elle peut donc être affectée à une collection.

```
In [6]: tva_dict = {}

In [6]: tva_dict['full'] = apply_base_tva

In [7]: def apply_reduced_tva(value):
   ...:     return value * 1.055
   ...:

In [8]: tva_dict['reduced'] = apply_reduced_tva

In [9]: tva_dict['reduced'](42)
Out[9]: 44.309999999999995
```

Une référence à une fonction peut également être un paramètre ou le retour d'une fonction. Une fonction peut donc attendre don algorithme par un paramètre de type fonction.

```
In [10]: def display_with_tva(value, tva_func):
    ...:     print("You bill is", tva_func(value))
    ...:

In [11]: display_with_tva(24, apply_reduced_tva)
You bill is 25.32
```

Fonctions anonymes (lambda)

Les lambda sont des fonctions particulières pour lesquelles l'interpreteur ne conserve pas de référence. Elles sont destinées à être utilisées à la déclaration.

Une lambda est déclarée par l'instruction `lambda` suivie de la liste des paramètres, deux points et l'instruction.

Les lambda ont plusieurs contraintes : elles ne peuvent être écrites que sur une ligne et ne peuvent contenir qu'une seule instruction.

L'usage le plus commun des lambda est de les déclarer à la volée là où une fonction est attendue comme pour notre fonction précédente.

```
In [12]: display_with_tva(24, lambda x: x * 1.18)
You bill is 28.32
```

Les expressions génératrices

Lorsqu'une fonction génère une collection de données, le défaut est que cette collection est générée dans son ensemble avant d'être retournée. Ceci peut avoir deux défauts :

- Le système va générer un certain volume de données qui vont occuper autant d'espace en mémoire
- L'appelant doit attendre que la fonction ai fini pour commencer à traiter les données.

Les expressions génératrices répondent à ces contraintes. Elles génèrent les données à la demande et les calcule à la volée. Elles sont donc appelées dans un iterateur.

Une expression génératrice est une fonction définie classiquement. La différence est qu'elle retourne une donnée avec l'instruction yield au lieu de return.

La présence de cette instruction yield transforme une fonction en expression génératrice. Ainsi, un appel à une fonction de ce type ne retourne pas le retour de la fonction mais un objet de type generator.

Il est alors possible d'itérer sur les retours de cette expression.

```
In [13]: def get_knights():
    ...:     for knight in ['Arthur', 'Lancelot', 'Robin']:
    ...:         yield knight.upper()
    ...:

In [14]: knights = get_knights()

In [15]: for knight in knights:
    …:     print(knight)
    ...:
ARTHUR
LANCELOT
ROBIN

In [16]: for knight in knights:
    ...:     print(knight)
    ...:

In [17]:
```

Par rapport aux listes, les expressions génératrices ont deux contraintes :

- Elles ne peuvent être parcourues qu'une seule fois
- On ne peut accéder à un élément par un indice

L'intérêt est que l'espace mémoire occupé est celui d'un objet de type générateur.

Le modèle sémantique des comprehension lists peut aussi créer un générateur, il suffit de remplacer les crochets par des parenthèses.

```
In [18]: knights = ['Arthur', 'Lancelot', 'Robin']

In [19]: KNIGHTS = (knight.upper() for knight in knights)
```

La programmation orientée objet

Les paradigmes de programmation

Il s'agit des différentes façons de raisonner et d'implémenter une solution à un problème en programmation. Nous pouvons citer trois grandes familles :

- La programmation impérative
  paradigme originel et le plus courant
- La programmation orientée objet (POO)
  consistant en la définition et l'assemblage de briques logiciel appelées objets
- La programmation déclarative
  consistant à déclarer les données du problème, puis à demander au programme de le résoudre

Le concept d'objet

Le problème à traiter va être modélisé par des objets. Un objet est une entité caractérisée par :

- Un état, représenté par des variables que nous appellerons des *attributs*.
- Des comportements, représentés par des fonctions intégrées à l'objet que nous appelleront des *méthodes*.

Un objet est donc un composant qui regroupe au sein d'une même entité des données et des traitements.

Vous avez déjà manipulé des objets. Si nous prenons les listes, l'état correspond aux données de la liste et les comportements sont les méthodes comme sort, split, append…

Un objet sera un composant du traitement de notre problème et plusieurs objets devront interagir pour traiter notre problème. Un principe pour la conception des objets est que **chaque objet devra avoir une seule responsabilité**. Par exemple, un objet qui gérera un compte ne s'occupera pas de son affichage. C'est justement le cas avec le type list.

La notion de classe

En programmation orientée objet, nous utilisons des objets. La structure de ces objets est défini dans des classes. Une classe peut être vue comme le moule qui permet de produire les objets. Dans le vocabulaire orienté objet, les objets sont des instances d'une classe. Les objets étant en soi des données, ils'ont un type qui est leur classe.

Représentation : présentation d'UML

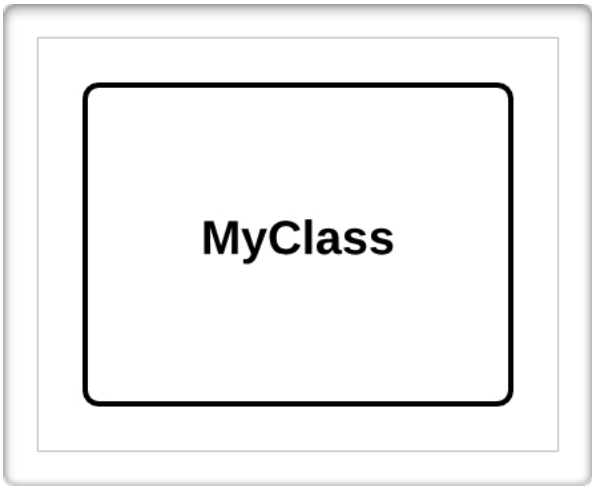
Afin de pouvoir représenter les notions de classes et objets, nous allons utiliser UML, Unified Modeling Language. UML est un langage de modélisation graphique basé sur des diagrammes et des pictogrammes. Il est standardisé et se veut être une synthèse des méthodes de modélisation. Pour cela, en version 2.3, il propose 14 type de diagrammes qui vont bien au delà de la modélisation objet.

Si UML permet de représenter beaucoup de choses sous forme de diagrammes, il n'est pas suffisant. Les diagrammes ne permettent pas de représenter toutes les contraintes. Nous exprimerons donc certaines contraintes en langage naturel.

Il existe cependant un langage formel, OCL (Object Constraint Language), depuis la version 1.1 d'UML. Le langage naturel est évidemment le plus facile à mettre en œuvre, à formuler et à comprendre. Mais il est également source d'ambiguïtés qu'OCL essaye d'éviter.

Déclarer une classe

La classe la plus basique peut être représentée par le diagramme ci-contre. Cette classe n'a ni état (pas d'attributs) ni de comportement (pas de méthode). Notez que le nom de la classe correspond à la règle de nommage de la PEP8 et est donc en CamelCase.



Ce diagramme va nous permet de déclarer la classe la plus basique en Python.

En Python, une classe est déclarée par la directive `class`. Elle est suivie du nom de la classe puis de deux points. Le corps de la classe doit évidemment être indenté.

Une classe est un *callable*. Un appel à cette classe retournera une instance de cette classe et pour maintenir une référence à cette instance, nous allons l'affecter a une variable.

Le code suivant permet de déclarer et d'instancier cette classe.

```
In [21]: class MyClass:
    ...:     pass
    ...:

In [22]: myInstance = MyClass()
```

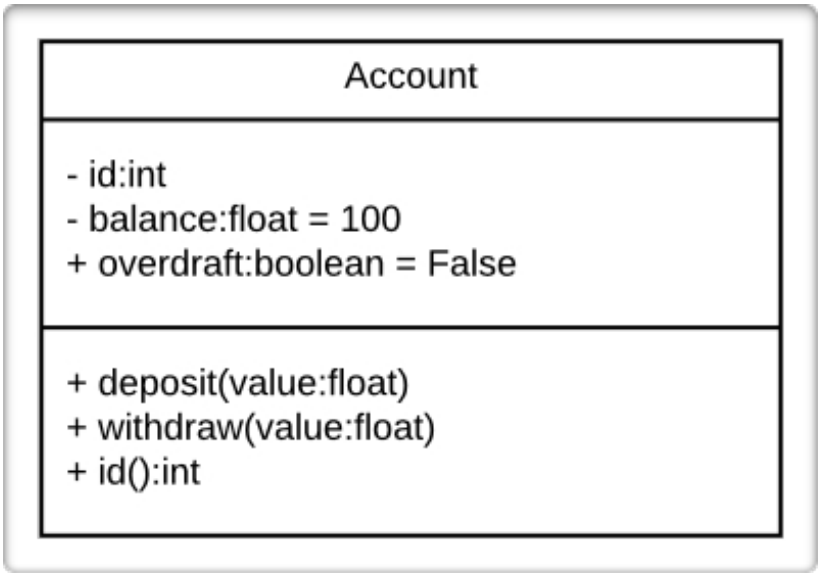
Cette classe et ces objets sans état et sans comportement ne servent pas à grand chose.

Attributs et méthodes : représentation

Pour illustrer le concept de l'orienté objet, nous utiliserons la notion de compte bancaire. Un compte bancaire possède un identifiant et un solde. Nous utiliserons également un paramètre *overdraft* pour définir si le compte est autorisé à un découvert ou non.

Le solde évolue par des dépôts et des retraits.

UML permet de représenter dans un diagramme de classe ses attributs et méthodes. Le diagramme suivant représente la notion de compte.



Une classe est représentée en 3 blocs. Le premier est le nom de la classe, le second est l'ensemble des attributs de la classe et la troisième les méthodes.

Pour les attributs, il est possible de spécifier le type et la valeur par défaut.

Les méthodes spécifient les paramètres attendus entre parenthèses ainsi que le type du retour.

Attributs et méthodes sont précédés d'un symbole qui représente leur visibilité. UML permet de représenter 4 visibilité :

- **Public** représenté par le symbole **+**. Un attribut ou une méthode publique pourra être accédé par n'importe quel composant.
- **Package** représenté par le symbole **~**. Un attribut ou une méthode package pourra être accédé par n'importe quel composant déclaré dans le même package.
- **Protected** représenté par le symbole **#**. Un attribut ou une méthode protégée ne pourra être accédé que par le composant qui l'a déclaré ou par tout composant spécialisé (voir notions d'héritage)
- **Private** représenté par le symbole **-**. Un attribut ou une méthode privée ne pourra être accédé que par le composant qui l'a déclaré.

La visibilité est un concept important de la conception objet. Dans notre exemple, l'identifiant et le solde (balance) sont des attributs privés car ils ne doivent pas être modifiés par un composant extérieur. Le solde évoluera par appel aux méthodes permettant le dépôt et le retrait. Afin d'accéder à la donnée identifiant, il faut fournir une méthode, ici la méthode `id()`.

Vous avez remarqué que ce modèle ne permet pas de consulter le solde.

Les méthodes

En Python, les méthodes sont des fonctions. Il s'agit donc des traitements. Lors de la conception d'un objet, les méthodes doivent être caractérisées par des verbes et représentent des actions.

Les méthodes peuvent recevoir des paramètres et retourner une donnée. Deux caractéristiques vont faire qu'une fonction est une méthodes :

- Les méthodes sont définies dans les classes.
- Vous devez déclarer un premier paramètre nommé par convention self. Ce paramètre est destiné à recevoir l'instance en cours. Vous pouvez déclarer d'autres paramètres en plus de ce paramètre self.

Pour appeler une méthode, il faut la préfixer par le nom de l'objet à laquelle elle appartient en ignorant le paramètre attendu self.

```
In [1]: class MyClass:
   ...:     def doSomething(self):
   ...:         print("I did something")
   ...:
   ...:     def doSomethingWith(self, value):
   ...:         print("I will do something with {}".format(value))
   ...:

In [2]: myObject = MyClass()

In [3]: myObject.doSomething()
I did something

In [4]: myObject.doSomethingWith('a value')
I will do something with a value
```

La nécessité de ce paramètre self est abordé dans la partie dédiée.

Les attributs

Un attribut est une donnée associée à un objet. En soi, c'est une variable qui contient donc une donnée. Lors de la conception d'un objet, les attributs sont caractérisés par des noms.

Pour manipuler un attribut au sein d'un objet, il faut faire précéder le nom de l'attribut par le mot-clef self (ce n'est pas qu'une convention). Pour manipuler un attribut en dehors de l'objet, il faut le préfixer par la référence à son instance.

```
In [10]: class Account:
    ...:     def set_balance(self, value):
    ...:         self.balance = value
    ...:     def get_balance(self):
    ...:         return self.balance
    ...:

In [27]: myAccount = Account()
In [28]: myAccount.set_balance(100)
In [29]: print(myAccount.get_balance())
100

In [30]: print(myAccount.balance)
100
```

Dans cet exemple, balance est un attribut public qui peut être accède par le code qui manipule l'objet comme le montre la ligne 30.

En Python, un objet n'a que des attributs

Si vous consultez la documentation de la fonction dir(), vous verrez qu'elle liste tous les attributs d'un objet. Or, dans cette liste, il y a des méthodes.

Rappelons qu'en Python, tout est objet. Une méthode peut donc être considérée comme un attribut qui a la particularité d'être un callable, donc de pouvoir être exécuté. Cette propriété permet la mise en œuvre de l'introspection.

Les méthodes et le paramètre self

Lorsque vous écrivez une méthode, vous êtes tenus de déclarer un premier paramètre qui par convention, aura pour nom self. Ce paramètre est destiné à recevoir l'instance. Explication…

Si les objets sont des instances de classes et que le principe de la programmation orientée objet consiste à définir des entités qui regroupent un état et des comportements. Il paraît évident de se représenter les objets comme des composants construits sur le modèle d'une classe et qui embarquent données (les attributs) et comportements. Mais en Python où l'implémentation de la POO est par prototype, ce n'est pas tout à fait ça.

Commençons par définir une classe et créons une instance.

```
In [10]: class Account():
    ...:     def __init__(self, balance):
    ...:         self.balance = balance
    ...:     def deposit(self, value):
    ...:         self.balance += value
    ...:         return self.balance
    ...:

In [11]: a = Account(100)
```

Puisque les méthodes sont des fonctions, si nous faisons appel à `a.deposit`, nous devons avoir en retour une référence à la méthode (fonction) `deposit` de l'objet `a`.

Mais nous avons vu qu'en Python, tout est objet. `Account(100)` retourne donc une instance de la classe Account mais Account seul retourne une référence vers la classe Account. Nous pouvons donc référencer les méthodes des classes.

La différence est importante, pour un objet, nous avons une *méthode liées* alors que pour une

```
In [12]: a.deposit
Out[12]: <bound method Account.deposit of <__main__.Account object at
0x104fbf278>>

In [13]: Account.deposit
Out[13]: <function __main__.Account.deposit>
```

classe, il s'agit d'une fonction.

Si vous faites appel à la fonction `Account.deposit()`, cela lève une exception car il manque 2 arguments. D'après la stack-strace, `a.deposit()` n'en attends qu'un seul. Ce second argument, c'est évidemment ce paramètre `self`.

Le paramètre `self` attends en effet une instance.

```
In [14]: Account.deposit(a, 100)
Out[14]: 200

In [15]: print(a.balance)
200
```

Nous illustrons ici ce qui se passe réellement avec l'implémentation du modèle objet en Python : on écrit `a.deposit(100)` et l'interpréter exécute `Account.deposit(a, 100)`.

Méthode particulière : le constructeur

Pour créer un objet, une méthode particulière est appelée. Cette méthode est habituellement appelée un constructeur (nous allons voir une particularité en Python). Cette méthode est appelée à l'instanciation de l'objet et permet de définir son état initial (voir après les attributs). Elle est disponible pour toutes les classes et par défaut ne définit aucun état.

Python a une particularité : lors de l'instanciation de l'objet, deux méthodes sont appelées successivement :

- La méthode `__new__(self)` qui est le **constructeur** et s'assure de la création technique de l'objet
- La méthode `__init__(self)` qui est un **initialiseur** et qui définit l'état initial de l'objet.

En Python, l'état initial d'un objet et plus généralement tout son comportement fonctionnel à l'initialisation sont définis dans l'initialiseur. Par abus de langage, on parle régulièrement de constructeur.

Le constructeur de base est le suivant.

```
In [10]: class UselessClass:
    ...:     def __init__(self):
    ...:         pass
    ...:
```

Le comportement de cet initialiseur est le comportement par défaut, il est inutile d'écrire la fonction `__init__()` de ce type. En son absence, un initialiseur équivalent par défaut est appelé.

Nous pouvons tester le comportement du constructeur par le code suivant.

Nous pouvons voir que le message est affiché à l'instanciation et donc que cette méthode est bien appelée à ce moment.

```
In [11]: class InitilizedClass:
    ...:     def __init__(self):
    ...:         print('I have been initilized')
    ...:

In [12]: InitilizedClass()
I have been initilized
Out[12]: <__main__.InitilizedClass at 0x1047cb5f8>
```

Méthode particulière : le destructeur

Il existe une autre méthode particulière communément appelée *destructeur*. En fait, c'est une méthode appelée juste avant la destruction de l'objet et elle est donc destinée à faire le ménage dans l'objet. Son comportement peut être illustré par le code suivant.

```
In [10]: class InitilizedClass:
    ...:     def __init__(self):
    ...:         print('I have been initilized')
    ...:     def __del__(self):
    ...:         print('I will be destroyed')
    ...:

In [11]: c = InitilizedClass()
I have been initilized

In [12]: del c
I will be destroyed
```

Cette méthode doit donc réaliser toutes les actions avant la destruction de l'objet (comme s'assurer que les connexion à une base de données soient coupées pour un objet de connexion à une base).

Constructeur, avancé

La construction d'un objet peut être paramétrée. Les paramètres qui sont attendus à la construction doivent être déclarés dans la méthode __init__(). Ainsi, le code suivant permet d'instancier un compte bancaire selon la spécification du diagramme UML.

Note : `id` est le nom d'une fonction native en Python, nous ne pouvons pas utiliser ce nom de variable sans masquer la fonction dans cet espace de nommage. Pour cette raison, le paramètre utilisé s'appelle `nid`.

```
In [10]: class Account:
    ...:     def __init__(self, nid, balance, overdraft):
    ...:         self._id = nid
    ...:         self._balance = balance
    ...:         self.overdraft = overdraft
    ...:
```

Cette implémentation ne correspond pas encore tout à fait à la spécification du diagramme car ce dernier déclarait des valeurs par défaut. Les méthodes étant des fonctions, il est évidemment possible de spécifier cas paramètres par défaut comme nous l'avons vu dans la partie sur les fonctions.

l'initialiseur est une méthode appelée à l'instanciation de l'objet et dont le rôle est de s'assurer de l'état initial de l'objet. C'est au sein de cette méthode que vous devez vous assurer de l'intégrité de l'objet. Par principe, mettez-y le tout le code assurant l'intégrité de l'objet. Déclarez également tous les attributs dont vous aurez besoin. Ainsi, nous pourrons implanter les méthodes

```
In [34]: class Account:
    ...:     def __init__(self, nid, balance=100, overdraft=False):
    ...:         self._id = nid
    ...:         self._balance = float(balance)
    ...:         self.overdraft = bool(overdraft)
    ...:
    ...:     def deposit(self, value):
    ...:         self._balance += float(value)
    ...:
    ...:     def withdraw(self, value):
    ...:         self._balance -= float(value)
    ...:
    ...:     def id(self):
    ...:         return self._id
    ...:
```

Visibilité en Python

En Python, tous les attributs et les méthodes ont une visibilité publique. Il existe cependant une convention : lorsqu'un nom d'attribut ou de méthode commence par le caractère souligné, l'underscore, il doit être considéré comme privé. Ainsi :

- `mon_attribut` est un attribut publique
- `_mon_attribut` est un attribut privé.

Il est important dans la conception objet de ne pas confondre visibilité et obfuscation. Python en particulier ne permet pas l'obfuscation du code.

Lors de la conception d'un objet, vous devez penser à comment cet objet interagit avec les autres composants et comment vous devez gérer son état. Un attribut public est destiné à être accédé par un composant extérieur. Un attribut privé soit parce qu'il n'a pas en soi de sens pour l'extérieur soit parce que sa modification doit être contrôlé par des règles métier. La visibilité en Python est fortement liée à la notion d'encapsulation qui est abordée ci-dessous.

Cette convention est très forte. Ainsi, par principe, n'accède pas directement à un attribut privé et si vous le faites, il s'agit de votre responsabilité.

La classe Account proposée précédemment suit cette convention.

Principe d'encapsulation

L'encapsulation est le principe consistant à masquer l'implémentation de données. Elle résulte du principe qu'un objet communique avec ses attributs et méthodes publiques. L'objectif est avant tout de maitriser l'évolution de l'implémentation.

Dans un système de gestion de compte, nous pouvons ajouter la notion de *client*. Cet objet possède la notion d'ancienneté, entier en nombre d'années depuis que la personne est devenue client. L'attribut géré par l'objet est donc une date et l'ancienneté est retournée par une méthode qui calcule cette ancienneté en fonction de la date actuelle.

```
In [1]: class client:
   ...:     def __init__(self):
   ...:         self._join = datetime(2015, 12, 16)
   …:
   ...:     def seniority(self):
   ...:         return (datetime.now() - self._join).days // 365
   ...:
```

L'usage de datetime est abordé dans la partie gestion des dates.

Le principe de l'encapsulation permet donc de laisser libre le développeur de la classe de choisir l'implémentation indépendamment de l'interface de la classe. Mais cette indépendance va aussi permettre de faciliter l'évolution. En effet, nous avons une méthode qui retourne l'ancienneté actuelle, mais si nous voulons avoir l'ancienneté à une date donnée, il faut avoir une méthode qui prend un paramètre date. En Python, avec la propriété des paramètres par défaut, nous pouvons faire évoluer cette même méthode sans aucune conséquence pour le code existant.

```
   ...:     def seniority(self, for_date=datetime.now()):
   ...:         return (for_date - self._join).days // 365
```

L'encapsulation permet de mettre en œuvre le principe Open/Closed : ouvert à l'extension, fermé à la modification.

Les properties

La visibilité des attributs d'une classe est *absolue* : elle définit tous les accès à l'attribut. Il n'est pas possible de définir un attribut qui sera par exemple accédé de manière publique en lecture mais privée en écriture. Dans notre exemple, c'est le besoin de l'attribut `balance` de la classe BankAccount et ce type de besoin est finalement assez courant. C'est pour cette raison que les attributs ont la visibilité la plus restreinte (privé dans notre implémentation) et si il y a besoin d'un autre type d'accès, c'est à travers des méthodes dédiées. Mais cette implémentation entraine que certains attributs sont obtenus par appel à l'attribut et d'autres par appel à des méthodes. Dans beaucoup d'autres langages, l'uniformisation consiste à définir tous les attributs en privé et à définir des méthodes dites *accesseurs*.

Python permet le respect du principe objet en proposant un descripteur de données, la fonction `property()`.

Pour plus d'informations que le document suivant, voir :
https://docs.python.org/3/howto/descriptor.html#properties

La signature de cette classe est la suivante :

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

Les attributs attendus sont :

- fget : une fonction appelée pour accéder à l'attribut
- fset : une fonction appelée pour définir l'attribut
- fdel : une fonction appelée pour détruire l'attribut
- doc : crée le docstring de l'attribut

L'usage général est donc le suivant :

L'affectation à la ligne 4 appelle en réalité la méthode `setx(self, value)` et la donnée est

```
In [2]: class C(object):
   ...:     def getx(self): return self.__x
   ...:     def setx(self, value): self.__x = value
   ...:     def delx(self): del self.__x
   ...:     x = property(getx, setx, delx, "I'm the 'x' property.")
   ...:

In [3]: c = C()

In [4]: c.x = 10

In [5]: c.x
Out[5]: 10
```

passée en tant que paramètre `value`. L'accès à l'attribut `x` ligne 5 appel en réalité la méthode `getx(self)` et retourne le retour de cette méthode.

Nous pouvons utiliser la fonction `property()` dans notre cas pour contrôler l'accès à l'attribut `_balance`. La seule action que nous acceptons est l'accès à la donnée, l'implémentation est la suivante.

```
class Account:
   def __init__(self, nid, balance=100):
        self._id = nid
        self._balance = float(balance)


    def _get_balance(self):
        return self._balance

    balance = property(_get_balance)
```

Avec cette implémentation, le solde peut être accédé comme un attribut mais ne pourra pas être modifié par affectation (cela lèvera un `AttributeError`). Le descripteur property permet la mise en œuvre du principe de la Programmation Orientée Objet en permettant un contrôle de l'accès aux attributs et en mettant en œuvre le principe de l'encapsulation. Ainsi, le solde est accédé par la property et évolue par appel aux méthodes `deposit` et `withdraw`.

Les méthodes spéciales

Nous avons vu jusqu'ici une méthode spéciale : __init__(self). Les méthodes spéciales sont des méthodes qui sont encadrés de double-underscores. Leur particularité est qu'elle ne sont jamais (rarement) appelées directement. __init__ est appelée par l'interpréteur lors de l'instanciation d'un objet. Python propose un ensemble de méthodes spéciales :

- `__init__(self)` : initialiseur appelé juste après l'instanciation d'un objet
- `__del__(self)` : destructeur, appelé juste avant la destruction de l'objet
- `__str__(self)` -> str : est appelé par la fonction de conversion de type `str()` et par la fonction `print()`. Elle doit donc retourner une chaine de caractères représentant l'objet.
- `__repr__(self)` -> str : est appelé par la fonction `repr()` et doit retourner une chaine de caractères contenue entre des chevrons et contenant non, type de l'objet et informations additionnelles.

Il est conseillé de redéfinir (surcharger, vois l'héritage) ces méthodes lorsque vous créez vos classes.

Python propose également des méthodes de comparaison entre objets. Celles-ci sont appelées par les opérations de comparaison correspondantes. Dans la colonne Opération du tableau suivant, x représente l'objet courant et y l'objet *other* avec qui il est comparé.

Méthode	Opération
__lt__(self, other)	x < y
__le__(self, other)	x <= y
__eq__(self, other)	x == y
__ne__(self, other)	x != y
__ge__(self, other)	x >= y
__gt__(self, other)	x > y

Ces méthodes ne doivent être redéfinies que si elles sont un sens. Si vos objets doivent être triés, vous devez redéfinir la méthode `__gt__(self, other)`.

Python propose également des méthodes spéciales qui permettent des opérations mathématiques entre les objets.

Méthode	Opération
__neg__	-x
__add__	x + y
__sub__	x - y
__mul__	x * y
__div__	x / y

Python propose d'autres méthodes spéciales, pour plus d'informations, vous pouvez consulter :

https://docs.python.org/3/reference/datamodel.html

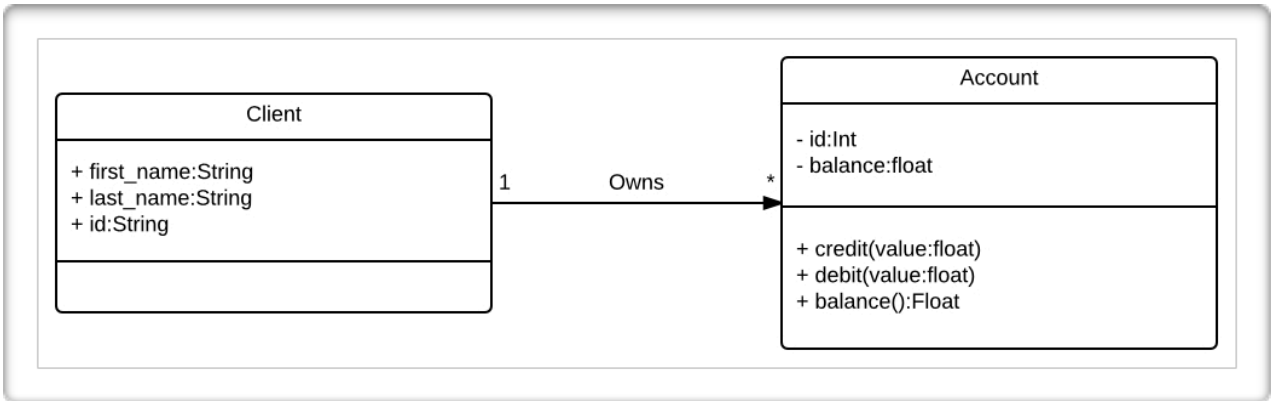
Les relations

Lorsque nous définissons des objets, ceux-ci sont liés les uns aux autres. Nous allons parler de relations.

UML permet de représenter des relations entre objets. La relation est symbolisée par un trait, en général horizontal, reliant les deux classes. Une flèche permet d'orienter la relation.

UML nous permet également de préciser la cardinalice de cette relation. Les indications possibles sont :

- n où n est un nombre signifie exactement n instances
- * signifie un nombre indéterminé d'instances
- m..n où m est un entier et n est soit un entier soit * et signifie de m à n instances.



Dans notre diagramme, un client peut avoir 0, 1 ou plusieurs comptes sans limite suppérieur. Un compte par contre doit appartenir à un client et à un seul. Un compte ne peut donc pas exister sans être associé à un client.

La flèche ainsi que la définition de la relation indiquée au dessus de cette flèche nous renseigne sur l'implémentation attendue. Ainsi, puisque le modèle précise qu'un client possède des comptes, l'implémentation de cette relation consistera à ajouter un attribut de type liste (de comptes) au sein de la classe Client. Pour implémenter ce modèle, nous n'avons pas besoin de toucher à la classe Account et nous pouvons simplement définir la classe Client suivante.

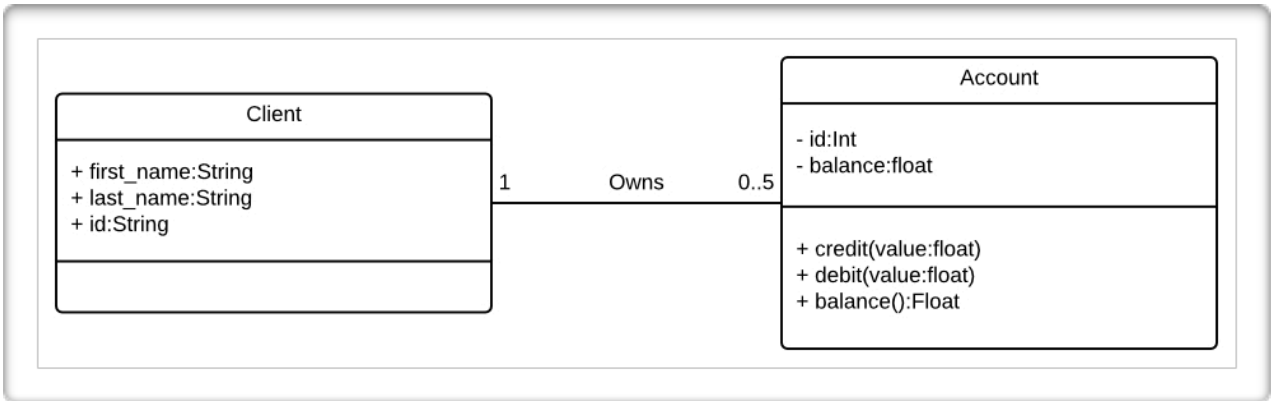
Il faut donc bien interpréter un diagramme UML avant d'implanter le modèle. Tous les attributs ne

```
In [20]: class Client:
    ...:     def __init__(self, nid, first_name, last_name):
    ...:         self.nid = nid
    ...:         self.first_name = first_name
    ...:         self.last_name = last_name
    ...:         self._accounts = []
    ...:
```

sont donc pas listés dans le pictogramme de la classe, les relations en définissent d'autres. La cardinalité de la relation indique sur le type de variable à implanter : si la cardinalité représente au maximum un élément, l'attribut contiendra un élément. Si plus de 1, ce sera une collection.

La cardinalité est représentée sur les deux cotés d'une relation, cependant, cela ne signifie pas qu'il est nécessaire d'ajouter une référence à chaque coté d'une relation. Faire ce choix complexifiera grandement la maintenance et la logique de traitement.

Nous pouvons complexifier la relation en imposant un nombre limité de comptes comme dans le diagramme suivant :



L'implémentation est la même que précédemment, c'est à dire que les comptes seront gérés par une liste. Mais il faut limiter la taille de la liste à 5 éléments. Les listes en Python n'étant pas limitées, cette contrainte sera assurée par la méthode `add_account` qui alimente cet attribut. L'implantation est donc la suivante :

```
def add_account(self, account):
    if len(self._accounts < 5):
        self._accounts.append(account)
```

Les limites de la représentation graphique

Nous avons vu que nous pouvons représenter beaucoup d'informations sous forme de diagrammes en UML. Mais la représentation graphique a des limites. Si nous définissons une relation entre Client et Account avec une cardinalité de 0..5, un client peut avoir jusqu'à 5 de n'importe quel compte. Le métier nous impose qu'un client ne peut avoir qu'un compte de type SavingAccount, spécialisation de Account. Il est impossible de représenter cette contrainte dans le diagramme.

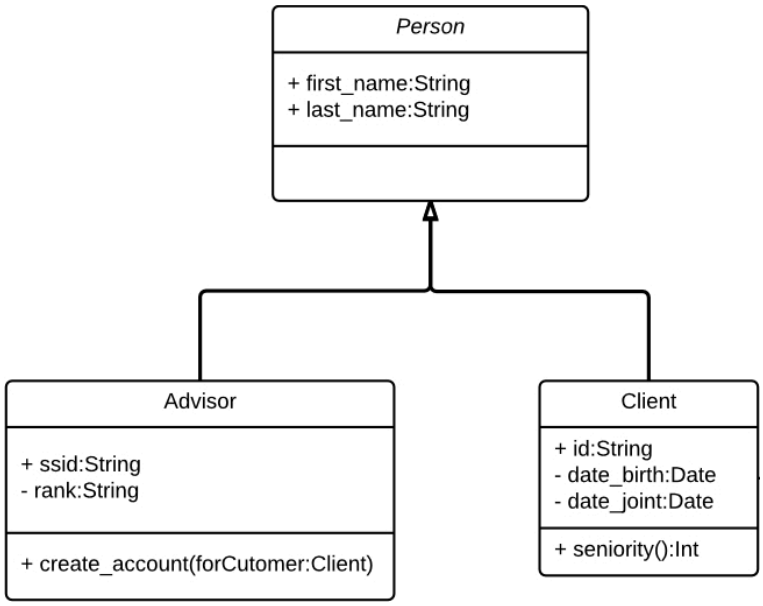
Cette contrainte doit donc être formulée dans un langage qui permet de la définir. C’est ce que propose OCL (Object Constraint Language). Cette formation n’est pas une formation UML/OCL, ce sujet ne sera pas abordé ici.

Héritage et abstraction

L'héritage

L'héritage est la propriété de généraliser ou spécialiser des états et/ou des comportements. La généralisation consiste à avoir une définition unique et éviter les duplications. La spécialisation consiste à adapter les caractéristiques et les comportements.

En UML, l'héritage est représenté par un trait vertical entre deux classes. Le diagramme suivant montre un héritage où `Person` est une généralisation de `Client` et `Advisor` et où `Client` et `Advisor` sont des spécialisation de `Person`.



L'héritage permet la mise en œuvre du concept de l'abstraction. L'abstraction identifie et regroupe des caractéristiques et traitements communs applicables à des entités ou concepts variés. Une représentation abstraite commune de tels objets permet d'en simplifier et d'en unifier la manipulation.

Héritage simple et héritage multiple

Le schéma précédent présente un héritage simple : chaque classe spécialisée n'hérite que d'une seule classe générale. Certains langages permettent l'héritage multiple, c'est à dire qu'une classe peut hériter de plusieurs classes. Elle acquiert alors les caractéristiques de l'ensemble des classes héritées.

En général, l'héritage multiple est difficile à mettre en œuvre efficacement car il est source d'erreurs et est assez difficile à maintenir. Même si le langage le permet, évitez un héritage multiple.

Héritage et type des objets

Le type d'un objet est sa classe. Mais dans le mécanisme de l'héritage, un objet est également du type d'une classe *parente*. Ainsi, une instance d'Advisor est aussi bien de type Advisor que Person.

Cette propriété est fondamentale dans les langages déclaratifs afin de manipuler plusieurs types spécialisés pour leur caractéristique commune. C'est le premier pas vers les notions d'abstraction.

Abstraction : classes abstraites et classes concrètes

Dans l'exemple précédent, la classe *Person* est représentée en italique ce qui définit une classe abstraite. Une classe abstraite est une classe qui ne peut être instanciée. Elle ne peut être que spécialisée.

Cette propriété permet de s'assurer qu'une classe qui n'a pas de sens dans notre système ne pourra exister en elle même. Mais qu'il sera possible de la spécialiser.

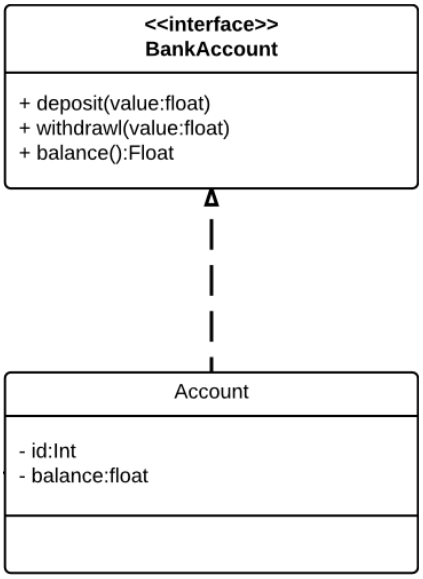
Abstraction : les interfaces

Une interface peut être définie de la manière la plus simple comme un contrat.

Il s'agit d'un composant abstrait qui ne fait que définir des méthodes publiques que les classes qui implémentent l'interface doivent définir. Une interface garanti donc un comportement commun entre des objets de types distincts sans aucune filiation. Une interface est déclarée par un trait vertical discontinu.

L'autre propriété d'une interfaces est que, même si elle n'est pas une classe, elle confèrent son type à la classe qui l'implémente. Dans l'exemple ci-contre, la classe Account est également de type BankAccount.

La notion d'interface est apparue avec les langages déclaratifs interdisant l'héritage multiple. Elles permettent un comportement et un type commun entre objets sans filiation.



Python et les abstractions

Par rapport à ce qui a été présenté ci-dessus, Python

- permet l'héritage
- supporte l'héritage multiple
- ne possède pas la notion de classes abstraites
- ne possède pas la notion d'interfaces

L'héritage en Python

Pour définir qu'une classe est une spécialisation d'une autre en Python, on l'indique dans la déclaration de la class en faisant suivre le nom de la classe par des parenthèses contenant le ou les nom(s) de la ou des classe(s) héritée(s).

```
class Client(Person):
    pass
```

Spécialiser une classe signifie lui donner un comportement spécifique, mais cela peut également signifier lui attribuer des comportements complémentaires. Dans ce dernier cas, une classe spécialisée peut déléguer à la classe générale ce qu'elle sait gérer. Lors de l'instanciation d'une

classe Client, la classe Person sait gérer l'initialisation de nom et prénom. Nous pouvons lui

```
class Client(Person):
    def __init__(self, nom, prenom, nid)
        Person.__init__(self, nom, prenom)
        self._identifiant = nid
        self._accounts = []
```

déléguer cette gestion dans le constructeur.

Les attributs créés par l'initialiseur de la classe Person seront évidemment accessibles par l'objet Client.

Notions de polymorphisme

Le polymorphisme est la capacité d'un système à choisir l'implémentation réelle d'une méthode. Nous la mettons déjà en œuvre en surchargeant la méthode `__init__(self)`.

Spécialiser une classe peut ainsi conduire à redéfinir le comportement de certaines méthodes. Si nous avons un nouveau type de compte `DeferedAccount` qui est compte à débit différé, l'opération de débit ne peut être la même que la classe Account dont il hérite. Nous allons donc implémenter cette classe de la manière suivante :

```
class DeferedAccount(Account):

    def __init__(self, nid, balance=100):
        Account.__init__(self, nid, balance)
        self._operations = []

    def withdraw(self, value):
        self._operations.append(value)
```

Pour les objets de type DeferedAccount, l'interpréter ne fera pas appel à la méthode withdraw de la classe parent mais bien à celle redéfinie (surchargée) dans la déclaration de DeferedAccount.

Polymorphisme et duck typing

En Python, la notion de polymorphisme repose sur la notion de duck typing.

> Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard

Ce qui nous intéresse en Python, ce n'est pas quel est le type réel de l'objet mais est-ce qu'il a la caractéristique nécessaire à l'action. Nous avons déjà vu la mise en œuvre de ce principe. Si nous définissons la fonction suivante :

```
In [10]: def triple_and_one(a, b):
    ...:     print(a * 3 + b)
    ...:
```

Nous pouvons l'utiliser avec des entiers

```
In [11]: triple_and_one(2, 5)
11
```

Mais aussi avec des chaines de caractères

```
In [12]: triple_and_one("Hip ", "Hourra")
Hip Hip Hip Hourra
```

Ce principe peut être résumé par cette citation :

> Don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with. If the argument fails this speci c-ducklyhood-subset-test, then you can shrug, ask « why a duck? »
>
> *Alex Martelli*

Ce qui nous conduit à un principe clef en Python :

> It's easier to ask for forgiveness than permission

La gestion des exceptions

La gestion des exceptions n'est pas exactement un traitement des erreurs. C'est un mécanisme qui va interrompre l'exécution du programme pour informer qu'un bloc appelé est dans un état qu'il ne sait pas gérer et que ce bloc délègue à l'appelant la gestion de cet état. La gestion des exception est donc avant tout un système de messages.

Prenons l'exemple du compte bancaire : si le paramètre `value` des méthodes de crédit ou de débit reçoit une valeur négative, cela reviendra à réaliser l'opération inverse. Créditer d'une somme négative reviendra à réaliser un débit… Vous devez donc vérifier au sein des méthodes de crédit et débit que la valeur passée est positive. La responsabilité des méthodes de crédit et de débit est limitée à réaliser ce crédit ou débit, de ce fait, si le paramètre ne permet pas de réaliser cette opération (cas d'une valeur négative), la méthode doit pouvoir en informer le code appelant et lui déléguer la gestion de cet état. Il n'appartient pas à la méthode de débit ou crédit de d'interpréter le traitement de cette valeur.

C'est là que le mécanisme des exceptions entre en action : en levant une exception, les méthodes de débit ou crédit vont pouvoir informer le code appelant aussi bien qu'elle n'a pas pu s'accomplir mais également de la raison pour laquelle elle ne s'est pas achevée. C'est donc à ce code appelant de gérer la situation.

Le mécanisme des exceptions fait qu'une portion de code déclare si elle peut gérer ou non un type d'exception. Si l'exception n'est pas gérée, elle est à nouveau transférée au code appelant. Si il n'y a plus de code appelant, le programme est stoppé et une *stack trace* est affichée.

Les points importants du mécanisme des exceptions sont :

- Interruption de l'exécution *normale* du programme
- Délégation à l'appelant de la gestion de l'état

Vous avez avez déjà été confronté à une stack trace comme dans les exemples ci-dessous (l'import d'un module inexistant et l'accès à une clef inexistante dans un dictionnaire) :

```
In [10]: import nawak
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-47-c948b8798146> in <module>()
----> 1 import nawak

ImportError: No module named 'nawak'

In [11]: d = {'clef': 'valeur'}
In [12]: d['no key']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-49-3d990c75721f> in <module>()
----> 1 d['no key']

KeyError: 'no key'
```

Dans une stack trace, la dernière ligne vous informe de l'exception qui a été transmise. Une exception est un objet et vous soyez dans ces deux exemples que deux types différents ont été retournés : `ImportError` et `KeyError`. Le type de l'exception est également une information.

Les acteurs

Le mécanisme de gestion des exceptions met en jeu deux acteurs :

- Un composant qui va *lever* une exception.
- Un composant qui, d'une manière ou d'une autre fait appel au composant précédent et qui va *gérer* l'exception.

Nous allons mettre en place l'ensemble du mécanisme.

Lever une exception

N'importe quelle portion de code peut lever une exception avec la directive `raise` suivie de l'exception à lever. L'exception lancée peut être de n'importe quel type héritant de la classe `Exception`.

Python propose un ensemble d'exceptions natives. Leur liste et arborescence est détaillé dans la documentation : https://docs.python.org/3/library/exceptions.html

Pour gérer le cas d'une valeur négative, nous pouvons utiliser une exception de type ValueError. Le code de la méthode de dépôt sera alors le suivant, celui de la méthode de retrait sera équivalent.

```
def deposit(self, value):
    if float(value) >= 0:
        self._balance += float(value)
    else:
        raise ValueError("Negative value")
```

Si vous faites un appel à la méthode `deposit` en fournissant une valeur négative, une exception de type `ValueError` sera levée. En exécutant l'appel à cette méthode dans un shell interactif, vous déclencherez la levée de l'exception et l'affichage de la stack trace.

```
In [11]: account.deposit(-200)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-5de152979884> in <module>()
----> 1 account.deposit(-200)

<ipython-input-15-56ce7bdd18ca> in deposit(self, value)
      4             self._balance += float(value)
      5         else:
----> 6             raise ValueError("Negative value")
      7

ValueError: Negative value
```

Dans un contexte normal d'exécution d'un programme, celui-ci ne doit pas s'interrompre et afficher la stock trace. Il faut donc capturer l'exception et la traiter.

La capture des exceptions

Pour capturer une exception, il suffit que le code qui lèvera l'exception soit dans un bloc dont l'en-tête est l'instruction `try:`. Ce bloc est suivi d'une instruction `except`. Le bloc `except` déclare les exception qu'elle gère et est en elle même un en-tête pour le bloc d'instructions appelé si l'exception est levée.

Ainsi, le code qui fait appel à la méthode `deposit` devrait être dans un bloc `try` suivi d'un `except` déclarant la gestion des `ValueError`. Ce code n'affiche plus de stack trace et le programme n'est plus arrêté brutalement.

```
In [13]: try:
    ...:     account.deposit("dix")
    ...: except ValueError:
    ...:     print("Negative value cant be used")
    ...:
Negative value cant be used
```

Le mécanisme des exceptions est un mécanisme d'information. Le contenu d'un message est donc très important pour informer correctement. Si au lieu d'avoir une valeur négative en paramètre nous avons une chaine de caractères non numérique, le comportement sera le suivant.

Passer une chaine de caractères non numérique à une fonction comme `float` ou `int` lève également une `ValueError`. Le message affiché ne correspond plus à ce qui s'est passé. Il nous faut donc discriminer les deux sources d'exception.

```
In [12]: try:
    ...:     account.deposit(-200)
    ...: except ValueError:
    ...:     print("Negative value cant be used")
    ...:
Negative value cant be used
```

Nous avons deux possibilités : essayer de différencier l'origine des deux exceptions ou créer notre propre type d'erreur. Dans les deux cas, nous exploiterons le fait que les exceptions sont des objets.

Manipuler les exceptions

Il est possible d'intercepter les exceptions dans la clause except avec l'instruction as qui l'affecter à une variable. Nous pouvons alors accéder aux attributs de l'exception. `ValueError` n'a qu'un attribut, `args`. Si vous voulez manipuler d'autres types d'erreurs, consultez leur documentation avec la fonction `help()`.

Cet attribut args contient un tuple des arguments qui ont été passé au constructeur.

```
In [14]: try:
    ...:     account.deposit("dix")
    ...: except ValueError as e:
    ...:     print("args:", e.args)
    ...:     print(e)
    ...:
args: ("could not convert string to float: 'dix'",)
could not convert string to float: 'dix'

In [15]: try:
    ...:     account.deposit(-200)
    ...: except ValueError as e:
    ...:     print("args:", e.args)
    ...:     print(e)
    ...:
args: ('Negative value',)
Negative value
```

Nous pouvons donc différencier le contenu de l'exception sur le contenu de ce tuple.

```
In [16]: try:
    ...:     account.deposit(-200)
    ...: except ValueError as e:
    ...:     if "Negative value" in e.args:
    ...:         print(e)
    ...:
Negative value
```

Attention, avec le code ligne 16, les autres erreurs de type `VelueError` sont silencieuses… Faites donc très attention à la gestion des exception dans le bloc `except`.

Créer ses propres exceptions

Les exceptions sont des objets, ou plutôt, des classes. Python propose déjà un certain nombre d'exceptions qui, vous l'aurez remarqué, héritent toutes de la classe `BaseException`. Nous pouvons donc créer notre propre exception qui devra hériter de `Exception`.

```
In [20]: class AccountValueError(Exception):
    ...:     pass
    ...:

In [21]: class Account:
    ...:     def deposit(self, value):
    ...:         if float(value) >=0:
    ...:             self._balance += float(value)
    ...:         else:
    ...:             raise AccountValueError("Negative value")
    ...:

In [22]: account = Account()

In [23]: try:
    ...:     account.deposit(-200)
    ...: except AccountValueError as e:
    ...:         print("Valeur négative")
    ...:
Valeur négative

In [24]: try:
    ...:     account.deposit("dix")
    ...: except AccountValueError as e:
    ...:         print("Valeur négative")
    ...:
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-e38d58c7cd72> in <module>()
      1 try:
----> 2     account.deposit("dix")
      3 except AccountValueError as e:
      4         print("Valeur négative")
      5

<ipython-input-17-a053ff71b0b9> in deposit(self, value)
      1 class Account:
      2     def deposit(self, value):
----> 3         if float(value) >=0:
      4             self._balance += float(value)
      5         else:

ValueError: could not convert string to float: 'dix'
```

N'oubliez pas que dans le mécanisme d'héritage, une classe est également du type de celle dont elle hérite. Si nous définissons nos propres exceptions, il faut spécialiser les exceptions existantes au plus porche de celles-ci. Une `AccountValueError` peut donc être considéré comme une spécialisation de la `ValueError` et l'implémentation la plus pertinente est :

```
In [25]: class AccountValueError(ValueError):
    ...:     pass
    ...:
```

Ainsi, une clause `except` qui déclare le type général ValueError interceptera cette exception ainsi que les spécialisations.

```
In [26]: try:
    ...:     account.deposit(-200)
    ...: except ValueError as e:
    ...:         print("Une erreur de valeur")
    ...:
Une erreur de valeur
```

C'est évidemment une manière efficace de capturer plusieurs types d'erreurs. Car évidemment, un même bloc peut lever plusieurs types d'exceptions.

Traiter plusieurs exceptions

Dans un bloc `try`, plusieurs exceptions peuvent être levées. Le bloc `except` peut gérer ces différentes exceptions soit en les cumulant soit avec plusieurs clauses `except`.

Nous allons définir une nouvelle erreur, `BusinessError` qui hérite de `Exception`. Cette exception est destinée à signaler une erreur *métier*. La méthode `withdraw` obéit aux mêmes règles que le dépôt en ce qui concerne les valeurs négatives. Mais un compte ne peut avoir un solde négatif. Si le paramètre value entraine un solde négatif, une `BusinessError` doit être levée. L'implémentation de la méthode est la suivante.

```
In [27]: class AccountValueError(ValueError):
    ...:     pass
    ...:

In [28]: class BusinessError(Exception):
    ...:     pass
    ...:

In [29]: class Account:
    ...:     def withdraw(self, value):
    ...:         value = float(value)
    ...:         if value < 0:
    ...:             raise AccountValueError("Negative value")
    ...:         if value > self._balance:
    ...:             raise BusinessError("Insuficent balance")
    ...:         self._balance -= value
    ...:
```

Si le comportement de l'appelant est le même pour les deux exceptions, alors nous pouvons les cumuler dans un tuple.

```
In [30]: try:
    ...:     account.withdraw(some_value)
    ...: except (AccountValueError, BusinessError):
    ...:     print("Wrong value")
    ...:
```

Si le comportement est différent, alors nous devons déclarer deux clauses `except`.

```
In [31]: try:
    ...:     account.withdraw(some_value)
    ...: except AccountValueError:
    ...:     print("Wrong value")
    ...: except BusinessError:
    ...:     print("Wrong account state")
    ...:
```

Si une exception est levée, l'interpréter parcourt les clauses `except` dans l'ordre et exécute la première qui correspond au type d'exception levée. Les suivantes sont ignorées. Faites donc attention de ne pas écrire un code qui ressemblerait au suivant :

```
In [32]: try:
    ...:     account.withdraw(some_value)
    ...: except ValueError:
    ...:     print("Some wrong value")
    ...: except AccountValueError:
    ...:     print("Negative value")
    ...:
```

Dans ce cas, le `except AccountValueError` n'est jamais exécuté car si une `AccountValueError` est levée, elle est traitée par le bloc du `except ValueError`. Si vous déclarez plusieurs clauses `except`, assurez-vous de déclarer les exceptions les plus spécialisées avant les plus génériques.

Si vous déclarez une clause `except` en omettant le nom d'exception capturée, la clause attrapera toutes les exception. C'est une mauvaise pratique !

Par principe, capturez toujours les exceptions au plus proche de la logique de votre programme. Si vous capturez une exception simplement pour éviter qu'elle ne remonte *planter* votre programme, vous mettez le système dans un état incohérent et le programme risque de planter *plus loin*. Les actions correctives seront alors plus compliquées.

Relancer une exception

Une exception peut être lancée de n'importe où, y compris dans la clause `except`. Cette possibilité est utilisée pour traiter partiellement une exception et informer le bloc appelant de ce qu'il reste à faire. L'exception relancée peut être du même type ou d'un autre type. N'oubliez pas qu'une exception, c'est un type et que le type est l'information principale transmise.

Supposons que le code appelant les méthodes `deposit` et `withdraw` sait qu'il peut lever des erreurs de type `ValueError`. Lui même est appelé par un code pour si ce type n'a pas de sens mais à juste besoin d'une erreur de type `BusinessError`. Nous pouvons écrire le code suivant qui intercepte les ValueError pour relancer des `BusinessError`.

```
In [33]: try:
    ...:     account.withdraw(some_value)
    ...: except ValueError:
    ...:     print("Some wrong value")
    ...:     raise BusinessError("Usefull info")
    ...:
```

Finally et else

La gestion des exceptions peut être complétée par deux autres mots-clef : `else` et `finally`.

Else définit un bloc qui sera exécuté si aucune exception n'est levée.

```
In [34]: try:
    ...:     account.withdraw(some_value)
    ...: except ValueError:
    ...:     print("Some wrong value")
    ...: else:
    ...:     print("Balance have changed")
    ...:
```

Finally définit un bloc qui sera exécuté après que tous les blocs précédents, qu'il y ai eu une exception ou non. Ce bloc doit contenir toutes les instructions qui doivent garantir que le système est dans un état *propre* après les instructions courantes. Les instructions de fermeture de fichier ou de connexion à une base de donnée sont toujours dans un bloc `finally`.

```
In [35]: try:
    ...:     account.withdraw(some_value)
    ...: except ValueError:
    ...:     print("Some wrong value")
    ...: finally:
    ...:     print("Balance have changed")
    ...:
Balance have changed
```

Quand utiliser les exceptions

Le mécanisme des exceptions sont à utiliser avant tout pour gérer les erreurs dont on n'a pas le contrôle et pour déléguer la gestion au bloc appelant. C'est un mécanisme qui évite de faire des tests qui la plupart du temps ne servent à rien. Par exemple, ouvrir un fichier est réalisé par la fonction open. Avant d'ouvrir le fichier, il faudrait :

- vérifier que le fichier existe
- vérifier qu'il s'agit d'un fichier (et non d'un répertoire)
- vérifier que j'ai les permissions d'accès au fichier
- vérifier que ce fichier n'est pas déjà ouvert en écriture.

L'ensemble des ces vérifications est lourde aussi bien à écrire qu'à maintenir. Et le temps que vous réalisiez les différents tests, l'état peut changer (race conditions). Avec les exceptions, il suffit d'écrire

```
In [36]: try:
    ...:     fichier = open('/tmp/file', 'w')
    ...: except (IOError, OSError):
    ...:     print('problem with file')
    ...:
```

En Python, les exceptions sont rapides contrairement à Java par exemple où le principe est d'éviter autant que possible d'avoir recourir aux exceptions qui alourdissent l'exécution du programme. Ainsi, en Python, il est courant d'utiliser les exceptions pour proposer un comportement par défaut.

Si nous devons récupérer un élément d'une liste mais si la liste est vide récupérer `None`, l'approche classique est de tester la liste.

```
In [37]: if myList and len(myList) > index_value:
    ...:     myValue = myList[index_value]
    ...: else:
    ...:     myValue = None
    ...:
```

C'est la philosophie « Watch before you leap ».

En Python, la philosophie est « Its better to ask for forgiveness than permission » et nous utiliserons.

```
In [38]: try:
    ...:     myValue = myList[index_value]
    ...: except IndexError:
    ...:     myValue = None
    ...:
```

Le mot clef With

Certaines situations d'exception sont régulières. Pour cela, certains Context Managers gèrent les cas les plus courants. Au lieu de gérer l'ouverture de fichier comme ceci :

```
In [39]: try:
    ...:     fichier = open('/tmp/fichier', 'w')
    ...: except(IOError, OSError):
    ...:     pass # Don't pass IRL
    ...: finally:
    ...:     try:
    ...:         fichier.close()
    ...:     except NameError:
    ...:         pass # file wasn't opened
    ...:
```

On peut simplifier par :

```
In [40]: try:
    ...:     with open('/tmp/fichier', 'w') as fichier:
    ...:         pass
    ...: except (IOError, OSError):
    ...:     pass # Don't pass IRL
    ...:
```

Notez qu'en Python 3, `IOError` et `OSError` héritent tous deux de `EnvironmentError`.

Pour plus d'informations sur les Context Manager, consultez la partie qui leur est dédiée.

Les tests

Cette partie traitera principalement des tests unitaires.

Pourquoi tester ?

Les tests ne font pas parti du code *fonctionnel*. Ils sont toutefois fondamentaux dans le développement logiciel. Les tests permettent de

- Montrer que le code fonctionne
- Montrer que le code répond aux attentes
- Illustrer l'usage du code
- Montrer que le code fonctionne toujours

Python propose deux outils pour les tests : doctest et unittest.

Doctest, la documentation auto-testée

Doctest est un test inclus dans la documentation proposé par le module doctest. Le principe est d'illustrer par un code simple le composant documenté.

```
def add(a, b):
    """
        :Example:

        >>> add(1, 1)
        2
        >>> add(2.1, 3.4)
        5.5

    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Dans cet exemple, l'exécution du module exécutera doctest qui recherchera tout code dans la documentation, l'exécutera et comparera le retour obtenu avec le retour attendu. Doctest ne signale que les erreurs.

Doctest est un outil pratique permettant d'illustrer l'usage mais limité à des exemples simples. N'oubliez pas que ces exemples vont se trouver dans la documentation.

Les tests unitaires avec unittest

La documentation complète est disponible à l'adresse :
https://docs.python.org/3/library/unittest.html

Le principe des tests unitaires est de tester un comportement et un seul. Pour cela, le système testé doit être isolé au maximum afin qu'aucun composant externe n'influe sur le résultat du test.

Python intègre une bibliothèque de tests, unittest, qui permet l'isolation et l'indépendance des tests.

Le principe est le suivant :

- On crée une classe héritant d'unittest par fonctionnalité testée (fonction, méthode …)
- Au sein de cette classe, on crée une méthode de test par cas testé. Le nom de ces méthodes doit commencer par test, unittest détermine par introspection si une méthode est une méthode de test.
- Unittest met à disposition deux méthodes, setup et teardown qui sont exécutées respectivement avant et après chaque méthode de test et qui sont destinés à préparer le test et à nettoyer l'environnement après le test.

Un test (implanté dans une méthode de test) consiste à exécuter un appel à la fonctionnalité et à faire une assertion sur l'état après cet appel.

Classes et méthodes doivent avoir un nom sans ambiguïté sur leur comportement. En cas d'échec d'un test, le reporting indiquera le nom de ces classes et méthodes. Un nom adapté facilitera la correction.

Les méthodes setup et teardown vous permettent de garantir l'indépendance des tests. La première doit s'assurer de la préparation de l'environnement pour le test. La seconde s'assure de nettoyer l'environnement pour le test suivant. N'oubliez pas que **vous ne pouvez faire aucune spéculation sur l'ordre d'exécution des tests**.

Le code suivant test la création d'un compte.

```
import unittest
from training.projects.bank import bank

class TestCreateAccount(unittest.TestCase):

    def testBasicAccount(self):
        account = bank.BankAccount('012345')
        self.assertEqual(100, account.balance)

    def testAccountWithBalance(self):
        account = bank.BankAccount('012345', 500)
        self.assertEqual(500, account.balance)

    def testAccountWithBalanceAsString(self):
        account = bank.BankAccount('012345', '500')
        self.assertEqual(500, account.balance)
```

Le code suivant test le dépôt. Tester un dépôt nécessite l'existence d'un objet de type compte, donc sa création qui est réalisée dans la méthode setUp. Cet objet étant modifié par l'appel à la méthode deposit, il est détruit par la méthode tearDown.

```
class TestDeposit(unittest.TestCase):

    def setUp(self):
        self.account = bank.BankAccount('012345', 500)

    def testBasicDeposit(self):
        self.account.deposit(100)
        self.assertEqual(600, self.account.balance)

    def tearDown(self):
        del self.account
```

Le bon fonctionnement de la méthode deposit nécessite que l'objet soit correctement créé. La bonne création de l'objet est assurée par les tests précédents.

Les principales assertions sont :

Methode	Vérifie que
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Dans le cas de notre compte bancaire, effectuer un dépôt (ou retrait) négatif conduit à la levée d'une exception. Il existe une assertion spécifique pour ce cas, `assertRaises` qui prend en paramètre l'exception attendue. Elle est utilisée de la manière suivante.

```
def testNegativeDeposit(self):
    with self.assertRaises(ValueError):
        self.account.deposit(-100)
```

Pour discriminer l'origine de la levée de l'exception, nous nous baserons sur les arguments et paramètres de l'exception. La source de l'exception doit pouvoir être identifiée non pas pour des questions de *testabilité* mais de facilité de maintenance.

```
def testNegativeDeposit(self):
    with self.assertRaises(ValueError) as context:
        self.account.deposit(-100)
        self.assertTrue('Negative value' in context.exception)
```

L'approche test-first

Il est indispensable pour la qualité du code que celui-ci soit accompagné de tests. La démarche la plus habituelle consiste à écrire son code en fonction des spécifications puis les tests. Cette démarche à plusieurs défauts :

- Reportés à la fin de la tâche, les tests ne sont finalement pas écrits. Après tout, le code fonctionne et il est urgent de passer à la tâche suivante.
- Les tests sont écrits en fonction du code existant et montrent surtout que le code répond aux attentes.
- Cette stratégie confirme le reproche fait à l'écriture des tests qui est que cela prends du temps.

L'idéal serait d'écrire les tests en même temps que le code en diminuant au maximum le temps nécessaire à leur écriture. C'est ce que permet l'approche TDD pour Test Driven Development.

La mise ne œuvre du TDD peut être décrit par 3 règles à respecter scrupuleusement :

1. Vous ne pouvez écrire du code de production que pour corriger un test en échec
2. Vous ne devez écrire qu'un minimum de code de test pour que celui-ci soit en échec
3. Vous ne devez pas écrire plus de code de production que pour faire passer un test en échec

Il est impossible de décrire plus cette technique dans un document qui n'y est pas dédié. Aussi, référez-vous à la démonstration faite en formation. Vous y verrez que

- Les intentions sont écrites avant leur réalisation
- Les intensions sont exhaustives
- Les tests sont plus exhaustifs que lorsqu'écrits après
- Le temps passé à écrire les tests est compensé par l'absence d'écriture de code inutile.

Trier les listes

Le type `list` possède une méthode `sort()` qui permet de trier les éléments selon leur ordre naturel. La méthode `sort()` trie la liste sur laquelle elle est appliquée et sa valeur de retour est `None`.

Il existe une fonction `sorted(iterable)` qui prend en paramètre un *itérable* et retourne une copie triée de ce dernier selon l'ordre naturel de ses éléments. L'itérable original n'est pas modifié.

> Note : nous voyons ici une convention forte : si une fonction qui transforme un objet (ici en le triant) retourne cet objet transformé, alors il s'agit d'une copie et l'original n'est pas modifié ; si elle ne retourne rien (`None`), alors l'objet original est modifié. Vous retrouverez régulièrement ce principe. Pensez à le suivre lorsque vous créez vos fonctions.

Les deux fonctions implémentent l'Algorithme Timsort[9].

L'ordre dit naturel dépends de l'implémentation de la méthode `__lt__(other)` des objets triés. Si vous voulez que vos objets puissent être triés selon leur ordre naturel, vous n'avez à surcharger que cette méthode.

Les deux fonctions possèdent deux paramètres optionnels : `key` et `reverse`.

Pour illustrer le tri, nous utiliserons en exemple une liste de comptes bancaires. Considérez que la classe `Account` surcharge la méthode `__lt__(other)` afin que l'ordre de tri naturel soit en fonction du numéro de compte.

```
def __lt__(self, other):
    return self._id < other.id()
```

[9] Voir : https://en.wikipedia.org/wiki/Timsort

Inverser l'ordre de tri

Le paramètre `reverse` est une valeur booléenne qui va indiquer si l'ordre de tri doit être inversé. Par défaut, ce paramètre est à `False`.

```
In [1]: accounts = []
In [2]: accounts.append(bank.BankAccount('cz002', 2000))
In [3]: accounts.append(bank.BankAccount('ac011', 1000))
In [4]: accounts.append(bank.BankAccount('bx511', 500))
In [5]: accounts
Out[5]: [BankAccount.cz002, BankAccount.ac011, BankAccount.bx511]

In [6]: sorted(accounts)
Out[6]: [BankAccount.ac011, BankAccount.bx511, BankAccount.cz002]

In [7]: sorted(accounts, reverse=True)
Out[7]: [BankAccount.cz002, BankAccount.bx511, BankAccount.ac011]
```

Trier selon un autre critère

Le paramètre `key` déclare une fonction qui extraira l'élément de comparaison. Cette fonction attends un paramètre (l'élément de la liste) et retourne l'élément de comparaison. Cet élément de comparaison peut être soit une transformation de l'élément de la liste soit l'extraction d'une caractéristique. On utilise habituellement une lambda pour le paramètre key mais vous pouvez évidemment définir une fonction pour un critère plus complexe.

```
In [10]: sorted(accounts, key=lambda x: x.balance)
Out[10]: [BankAccount.bx511, BankAccount.ac011, BankAccount.cz002]
```

Exécuter un programme Python

Il y a plusieurs méthodes pour exécuter un module. S'il est indépendant, vous pouvez le rendre exécutable en vous assurant auparavant de l'existence d'un shebang. La première ligne du module doit être celle-ci :

`#!/usr/bin/env python`

Autrement, la manière conseillée est d'exécuter l'interpréter à la racine de votre librairie avec l'option `-m` et en paramètre, le nom complet (avec les packages) du module principal. Les options suivant `-m` ne sont plus passées à l'interpreteur mais à votre programme. Ainsi, exécuter le module mediacli du projet doit être lancé par :

`python -m training.projects.mediamanager.mediacli`

L'appel au module de démonstration des arguments de la façons suivante :

`python -m training.cases.argparse_basic me 3 -v`

Ici, `-m` est une option de l'interpréter et `me`, `3` et `-v` de `argparse_basic`.

Les arguments de la ligne de commande

Chaque argument de la ligne de commande est ajouté à sys.argv qui est de type liste. Le programme suivant sert à démontrer le comportement.

```
import sys

if __name__ == '__main__':
    for arg in sys.argv:
        print(arg)
```

Voici le résultat suite à un appel avec des arguments.

```
$ basic_args.py Doctor -v -o output.txt
basic_args.py
Doctor
-v
```

Comme vous pouvez le constater, le nom du programme est également un argument de la ligne de commande. Ce nom est le nom utilisé pour lancer le programme.

Vous pouvez donc itérer sur cette liste pour en extraire les paramètres de la ligne de commande. Mais gérer les options peut être fastidieux.

L'approche historique avec getopt

https://docs.python.org/3.6/library/getopt.html

Python intègre une bibliothèque historique, getopt, qui comporte une fonction transposée de getopt() du C. Getopt est proposé pour les utilisateurs familiers avec cette fonction C. Cette partie est destinée à présenter l'usage de cet outil.

Getopt permet de déclarer les options exploitées par votre programme. La définition de la fonction est :

`getopt(args, options[, long_options]) -> opts, args`
Les paramètres sont les suivants :

- args est la liste qui va être parsé, typiquement sys.argv[1:]
- options est une chaine de caractères déclarant les options à un caractère (appel type -h). Si l'option attends un argument, le caractère est suivi par deux points (par exemple : 'o:')
- long_options est une liste des noms long des options (appel type —help). Si l'option attends un argument, le nom sera suivi d'un égal (par exemple: 'output=').

Exemple pratique : nous souhaitons utiliser les options h ou help pour afficher l'aide, v ou verbose pour afficher plus d'informations et o ou output pour définir un fichier de sortie. La déclaration sera la suivante :

```
getopt.getopt(sys.argv[1:], 'hvo:', ['help', 'verbose',
'output='])
```

La fonction getopt retourne deux éléments :

- une liste de listes de deux éléments, l'option et l'argument
- une liste des arguments *supplémentaires*

Le code suivant permet d'observer le comportement.

```
import getopt, sys

try:
    opts, args = getopt.getopt(sys.argv[1:], 'hvo:',
                               ['help', 'verbose', 'output='])
except getopt.GetoptError as err:
    print(str(err))
    sys.exit(2)

for option, argument in opts:
    print(option + ":" + argument)
print("-----")
for argument in args:
    print(argument)
```

```
$ understanding_getopt.py --verbose -o file.txt extra1 extra2
--verbose:
-o:file.txt
-----
extra1
extra2
```

Un résultat peut être :

Le comportement général est le suivant :

- Si une option ne fait pas parti des options déclarées, cela provoque une erreur
- Si un argument est associé à une option alors qu'elle était déclaré ne pas en attendre, cela provoque une erreur
- Si un argument était attendu pour une option mais n'est pas fourni, cela provoque une erreur
- Les options doivent être passées avant les arguments supplémentaires sinon cela provoque une erreur

Getopt est donc un outil simplifiant la validation des arguments de la ligne de commande. L'utilisateur doit ensuite itérer sur les options validées pour les traiter comme dans l'exemple ci—contre.

Getopt est présenté pour raison historique. De manière générale, il est préférable de s'orienter vers **argparse**.

```
for o, a in opts:
    if o in ('-h', '--help'):
        usage()
        sys.exit()
    elif o in ('-o', '--output'):
        output_filename = a
```

La méthode recommandée, argparse

https://docs.python.org/3/library/argparse.html

Argparse est un outil qui va permettre de définir et exploiter des arguments de la ligne de commande. Cette gestion passe par un objet unique de type **argparse.ArgumentParser**. Pour l'utiliser, il suffit d'instancier l'objet et appeler la méthode parse_args().

```
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(
        description="This script does something.")

    args = parser.parse_args()
```

Lancer ce programme ne fait rien. Le lancer avec l'argument -h par contre provoque le résultat suivant.

```
$ basic_argparse.py -h
usage: basic_argparse.py [-h]

This script does something.

optional arguments:
  -h, --help  show this help message and exit
```

Argparse gère le comportement par défaut et l'option la plus standard (help). Cet affichage pourra être enrichi grâce au paramètre help des différents arguments déclarés.

Arguments positionnels

Argparse permet de déclarer des arguments positionnels, c'est à dire des paramètres attendus par le programme qui vont être utilisés comme des données. Ainsi, pour un Hello World, on peut ajouter un paramètre *who*.

```
parser = argparse.ArgumentParser(
    description="This script does something.")

parser.add_argument("who")

args = parser.parse_args()

print("Hello " + args.who)
```

Un appel de ce programme sans argument provoquera une erreur.

Par défaut, le type de la donnée est traité en tant que chaine de caractères. Il est possible de spécifier le type. Ceci permettra également de valider la donnée.

```
parser = argparse.ArgumentParser(
    description="This script does something.")

parser.add_argument("who", help="Who are you ?")
parser.add_argument("many", type=int)

args = parser.parse_args()

for i in range(args.many):
    print("Hello " + args.who)
```

Les arguments optionnels

Les arguments optionnels, courts ou longs, sont spécifiés de manière similaire. La différence est que leur nom est préfixée par un tiret (pour les arguments courts soit une lettre) ou deux tirets (pour les arguments longs, soit des mots). Par défaut, ces options attendent un argument. Une option n'attendant pas d'augment peut être considérée comme un drapeau qui prend les valeurs True ou False.

```
parser = argparse.ArgumentParser(
    description="This script does something.")

parser.add_argument('-v', '--verbose', action='store_true')
parser.add_argument('-o', "--output",
                    help="Output file name")

args = parser.parse_args()

if args.verbose:
    print('lets do it')

if args.output:
    print("output file: " + args.output)
```

Les arguments positionnels et optionnels peuvent cohabiter. L'ordre des arguments positionnels doit évidemment être respecté, mais les arguments optionnels peuvent être placés dans n'importe quel ordre.

Manipuler les fichiers

Pour manipuler un fichier un fichier avec Python, aucune dépendance n'est nécessaire. La fonction `open` est une fonction de base de Python. Elle prends deux paramètres : le fichier à ouvrir et le mode. Le premier paramètre est le nom (absolu ou relatif) du fichier à ouvrir. Le mode est une chaine de caractère avec les options suivantes :

- `r` : lecture seule
- `w` : écriture, écrase un fichier existant, crée un fichier inexistant
- `x` : ouverture pour création (exception si fichier existe)
- `a` : écriture en mode ajout, écrit en fin de fichier, le crée si inexistant
- `b` : ajouté au précédent ouverture en mode binaire
- `t` : ajouté au précédent ouverture en mode texte (défaut)
- `+` : ajouté au précédent lecture et écriture

Il est important de toujours refermer un fichier ouvert par la fonction `close`. Une bonne pratique consiste à toujours fermer le fichier dans une clause `finally`.

```
In [1]: try:
   ...:     mfile = open('showsfiles.txt', 'r')
   ...: except OSError:
   ...:     if not mfile.closed():
   ...:         mfile.close()
   ...:

In [2]: mfile
Out[2]: <_io.TextIOWrapper name='showsfiles.txt' mode='r' encoding='UTF-8'>

In [3]: mfile.close()
```

Ouvrir un fichier peut se faire grâce à un context manager. Le context manager s'occupe automatiquement de la fermeture du fichier dès que l'on sort du bloc. Toute la manipulation du fichier devra dont être réalisée dans ce bloc.

```
In [7]: with open('showsfiles.txt', 'r') as mfile:
   ...:     mfile.read()
   ...:
```

Nous allons dans cette partie traiter uniquement des fichiers texte. Depuis Python 3, les fichiers text sont de type `io.TextIOWrapper`.

Lire et écrire dans un fichier

Nous avons plusieurs méthodes pour lire et écrire dans un fichier.

Écrire dans un fichier

Pour écrire dans un fichier, le fichier doit être ouvert en écriture. L'objet fichier propose les méthodes suivantes :

- `write(str)` : écrit la chaine `str` dans le fichier et retourne le nombre de caractères écrits.
- `writelines(sequences)` : écrit à la suite dans le fichier les chaines de la liste passée en paramètre. Attention, aucun caractère n'est inséré en plus du paramètre. Si les chaines ne sont pas terminées par des caractères saut de ligne, le résultat sera une ligne en fin de fichier.

Lire dans un fichier

Pour lire un fichier, celui-ci doit être ouvert en lecture. L'objet fichier propose plusieurs méthodes pour lire dans un fichier :

- `read([size])` : sans argument, retourne le contenu du fichier sous forme d'une chaine de caractères. L'argument optionnel est la taille maximum lue en nombre de caractères.
- `readline([size])` : retourne la ligne suivant sous forme d'une chaine de caractères. Si l'argument optionnel size est renseigné, la lecture sera limitée à ce nombre de caractères.
- `readlines([hint])` : Sans argument, retourne le contenu du fichier sous la forme d'une liste de chaines de caractères correspondant aux lignes contenues dans le fichier. Le paramètre hint spécifie le nombre de caractères lus. Le nombre total de caractères de la liste ne pourra donc pas être supérieur à ce paramètre.

L'objet fichier propose un itérateur qui facilite le parcours et le traitement d'un fichier ligne à ligne. Son usage est régulièrement préférable aux méthodes précédentes. L'usage est le suivant :

```
In [1]: for line in file:
   ...:     print(line[:-1])
```

Manipuler le curseur

Lorsqu'on parcourt un fichier, on déplace un curseur. Il est donc possible de manipuler le curseur sur un fichier. Pour cela, nous avons 2 méthodes :

- `tell()` : retourne la position du curseur
- `seek(offset[, whence])` : permet de déplacer le curseur de la valeur offset en fonction du paramètre whence. Le paramètre whence peut prendre trois valeurs :
  - `0` ou `os.SEEK_SET` : déplace le curseur par rapport au début du fichier. C'est le comportement par défaut. Le paramètre offset est un entier de valeur 0 ou positif.
  - `1` ou `os.SEEK_CUR` : déplace le curseur par rapport à la position courante du curseur. Pour un fichier texte, le paramètre offset ne peut avoir que la valeur de 0.
  - `2` ou `os.SEEK_END` : déplace le curseur par rapport à la fin du fichier. Pour un fichier texte, le paramètre offset ne peut avoir que la valeur de 0.

Le système de fichiers

Python propose plusieurs modules complet pour exploiter le système de fichier.

- os : https://docs.python.org/3/library/os.html
  Permet d'utiliser des fonctions dépendant du système d'exploitation
- os.path : https://docs.python.org/3/library/os.path.html#module-os.path
  Permet de manipuler les *chemins*. os.path est un module virtuel dont l'implémentation réelle est chargée en fonction du système sur lequel s'exécute le code.
- shutil : https://docs.python.org/3/library/shutil.html#module-shutil
  Permet des manipulations de haut-niveau des fichiers. Certaines fonctions peuvent appeler des fonctions du module os.

Exemples de fonctions du module os :

- `os.mkdir(chemin, mode)` : crée répertoire, mode UNIX
- `os.remove(chemin)` : supprime fichier
- `os.removedirs(chemin)` : supprime répertoires récursivement
- `os.rename(chemin_old, chemin_new)` : renomme fichier ou répertoire
- `os.renames(chemin_old, chemin_new)` : renomme fichier ou répertoire en créant les répertoires si ils n'existent pas
- `os.chmod(path, mode)` : change les permissions
- `os.chdir(chemin)` : change le répertoire de travail
- `os.getcwd()` : affiche répertoire courant

Exemples de fonctions du module os.path :

- `os.path.exists(chemin)` : est-ce que le fichier ou répertoire existe
- `os.path.isdir(chemin)` : est-ce un répertoire
- `os.path.isfile(chemin)` : est-ce un fichier
- `os.path.dirname(path)` : retourne l'arborescence de répertoires
- `os.path.basename(path)` : retourne le nom du fichier
- `os.path.split(path)` : retourne un tuple des deux précédents
- `os.path.splitext(path)` : retourne un tuple pour obtenir l'extension

Exemples de fonctions du module shutil :

- `shutil.move(src, dest)` : déplace ou renomme un fichier ou un répertoire
- `shutil.copy(src, dest)` : copie un fichier ou un répertoire
- `shutil.copy2(src, dest)` : copie un fichier ou un répertoire avec les métadonnées

Cas particulier du listing de fichiers d'un répertoire :

- `os.listdir(chemin)` : liste un répertoire
- `glob.glob(pattern)` : liste le contenu du répertoire en fonction du pattern et permet ainsi l'utilisation de wildcards.

Le module pickle

Pour plus d'informations : https://docs.python.org/3/library/pickle.html

Le module pickle est le module permettant la stérialisation et désérialisation d'objets en Python. Sérialiser un objet consiste à convertir un objet en flux d'octets ce qui permet de sauvegarder une donnée et son type. Pour éviter les confusions entre *serialization*, *marshalling*, et *flattening*, on parlera de *pickling* et *unpickling*.

Le module pickle propose deux fonctions principales :

- `dump(objet, file)` : sauvegarde la donnée
- `load(file)` : charge une donnée

Le module pickle propose également deux classes :

- `Pickler(file)` qui possède une méthode `dump(object)` qui sauvegarde une donnée
- `Unpickler(file)` qui possède une méthode `load()` qui charge une donnée

L'usage est le suivant :

```
pile_name = "ma pile"
pile_value = [42, 'answer']

import pickle

f = open("pile_backup", w)
pickle.dump(pile_name, f)
pickle.dump(pile_value, f)
f.close
```

```
import pickle

f = open("pile_backup", r)
pile_name = pickle.load(f)
pile_value = pickle.load(f)
f.close
```

Les fichiers CSV

Python propose un module permettant de gérer les fichiers CSV. Pour la documentation complète, voir : https://docs.python.org/3/library/csv.html

Le principe général de ce module est de fournir des objets permettant la lecture ou l'écriture de fichiers CSV. L'utilisation de base fera appel aux fonctions `csv.reader(csvfile)` et `csv.writer(csvfile)` qui retourneront un objet de type `csv.reader` et `csv.writer`. Un paramètre optionnel `delimiter` permet de spécifier le séparateur de colonnes et un paramètre optionnel `quotechar` permet de spécifier le caractère encadrant les chaînes. Un exemple de lecture est proposé dans le code suivant :

```
In [10]: with open('csvfile.csv') as csvfile:
    ...:     csvreader = csv.reader(csvfile, delimiter=';',
    ...:                            quotechar='"')
    ...:     for row in csvreader:
    ...:         print(row)
    ...:
```

Un exemple d 'écriture est proposé dans le code suivant :

```
In [24]: with open('csvfile.csv', 'w', newline='') as csvfile:
    ...:     csvwriter = csv.writer(csvfile, delimiter=';')
    ...:     for row in list_elements:
    ...:         csvwriter.writerow(row)
    ...:
```

Les bases de données

Cette partie va présenter les principes de base de la connexion aux bases de données, à l'envoi de requêtes et au traitement des données retournées. Cette partie traite des bases de données relationnelles. Les outils tel que les ORMs[10] ne sont pas abordés.

Le principe général indépendant de la base de données est :

- Établir une connexion
- Créer un *curseur* sur cette connexion
- Attribuer une requête à ce curseur
- Récupérer les données si il y a lieu
- Fermer la connexion

Connexion à MySql

Cette partie illustre le principe précédent à l'aide d'une base MySql. Cet exemple est évidemment transposable directement à MariaDb et plus généralement à tout système de gestion de bases de données (PostgreSql, Oracle…).

Nous aurons besoins d'un connecteur ou pilote. Pour MySql, il est disponible par pip ou sur le site d'Oracle.

Le code suivant montre comment récupérer la version de MySql.

```
import MySQLdb

try:
    conn = MySQLdb.connect(host='localhost', user='test user',
                           passed='test pass', db='test')
    cursor = conn.cursor()
    cursor.execute("SELECT VERSION()")
    row = cursor.fetchone()
    print('server version', row[0])

finally:
    if conn:
        conn.close()
```

Connexion à SQLite

Plutôt que d'utiliser MySql, nous préférerons SQLite. Les différences sont que :

- Le pilote SQLite est disponible dans l'installation standard de Python. Il n'est pas nécessaire d'installer d'autres dépendances.
- SQLite est une base de données fichier. Il n'y a pas de notion de base de donnée à laquelle se connecter ni d'utilisateurs.
- Lors de la connexion à la base, si elle n'existe pas, elle est créée.
- Il est possible de travailler avec une base en mémoire.

[10] Object-Relational Mapping, mapping objet-relationnel

SQLite est une base que vous pouvez utiliser à chaque fois que vous avez besoin de gérer des données de manière relationnelle. En tant que base en mémoire, SQLite est la base à utiliser lors des tests.

Le code suivant montre comment récupérer la version de SQLite.

```
import sqlite3 as lite

con = lite.connect('testdb.db')

try:
    cursor = con.cursor()
    cursor.execute("SELECT SQLITE_VERSION()")
    row = cursor.fetchone()
    print('server version', row[0])

finally:
    if con:
        con.close()
```

Le code suivant montre comment se connecter à une base en mémoire.

```
con = sqlite3.connect(":memory:")
```

Utilisation d'un Context Manager

Les connecteurs aux bases de données proposent des context managers. Celui-ci prends en charge la fermeture de la connexion. La contre-partie est que la connexion étant fermée à la sortie du bloc suivant le `with`, l'ensemble des traitements doit être réalisé dans ce bloc.

```
import sqlite3 as lite

con = lite.connect('testdb.db')

with con:
    cursor = con.cursor()
    cursor.execute('SELECT SQLITE_VERSION()')
    row = cursor.fetchone()
    print('server version', row[0])
```

Les requêtes

Les requêtes sont exécutées sur des curseurs.

Création de tables

Les requêtes de création de tables sont en général des requêtes prédéfinies. Ce sont donc des chaînes de caractères passées en paramètre au curseur par la méthode execute.

```
cur = con.cursor()
cur.execute("CREATE TABLE IF NOT EXISTS people (name_last, age)")
```

Insertion de données : requêtes paramétrées

De nombreuses requêtes attendent des paramètres (insertion, filtre…). Il faut alors utiliser la fonctionnalité des requêtes paramétrées. Pour cela, l'endroit de la requête SQL où un paramètre est attendu est marqué par un `?`. Les paramètres attendus seront ajoutés à l'appel de la méthode exécuté.

```
who = "Yoda"
age = 800

cur.execute("INSERT INTO people VALUES (?, ?)", (who, age))
```

Vous pouvez également utiliser les requêtes paramétrées avec une sémantique de dictionnaire.

```
who = "Yoda"
age = 800

cur.execute("select * from people where name_last=:who and age=:age",
            {"who": who, "age": age})
```

Utilisez toujours les requêtes paramétrées qui ont entre autre la fonction d'échapper tout caractère qui ferait parti d'une requête et ainsi limiter les risques d'injection SQL.

Récupérer les données

Une requête d'extraction de données retourne des enregistrements. Un enregistrement n'est autre qu'une liste de données. Pour obtenir ces enregistrements, nous avons plusieurs méthodes à not disposition après l'exécution de la requête :

- `fetchone()` : renvoi l'enregistrement suivant ou None si il n'y en a plus.

- `fetchmany([size])` : retourne un nombre d'enregistrements correspondant au paramètre size. En cas d'itération, size doit rester constant.
- `fetchall()` : envoi tous les enregistrements correspondant à la requête sous la forme d'une liste (dans la limite du paramètre `cursor.arraysize` de la base de données

Les transactions

Python gère évidemment la notion de transaction. Les transactions sont indispensables lorsque les données sont modifiées. Une transaction débute avec la première requête et se poursuit jusqu'à l'appel de la méthode `commit()`. Pour annuler une transaction, il faut faire appel à `rollback()`.

Le context manager gère les commit et rollback.

Organisation du code

L'organisation du code (connexion, exécution des requêtes, traitement des données…) dépendra évidemment de votre application. Une manière d'organiser son code de connexion à une base de données est de créer une classe de type *dao*[11]. Vous avez quelques exemples dans des modules du package training.projects (les modules se terminant par _db). Le principe général proposé dans ce code consiste à :

- Établir la connexion dans le constructeur. Pour une connexion à une base SQLite, les tables sont également créées dans le constructeur.
- Le destructeur s'occupe de fermer la connexion.
- Des méthodes encapsulent les requêtes et le chargement des données.

[11] Data Access Object

Manipuler les dates avec datetime

La manipulation des dates en Python se fait grâce au module datetime. Celui-ci fournit une classe, `datetime`, qui permet de manipuler la notion d'instant.

Création d'un instant

Les paramètres du constructeur sont : l'année, le mois, le jour, l'heure, les minutes, les secondes, les microsondes et le *timezone*. Les paramètres autre que le timezone sont des entiers et seuls l'année, mois et jour sont nécessaires.

```
In [1]: from datetime import datetime

In [2]: datetime(2015, 12, 16)
Out[2]: datetime.datetime(2015, 12, 16, 0, 0)
```

L'objet retourné est donc un instant à la date donnée à minuit.

Pour créer un objet représentant l'instant présent, la **classe** `datetime` fournit la méthode `now()`.

```
In [3]: datetime.now()
Out[3]: datetime.datetime(2015, 12, 16, 14, 30, 13, 835376)
```

L'objet datetime et la représentation texte

Plus habituellement, nous chargeons des données à partir de chaines de caractères et nous représentons ces données sous forme de chaines de caractères. La classe datetime fournit deux méthodes. Ces deux méthode auront besoin d'un paramètre format définissant la représentation de la structure de date. Python repose sur la bibliothèque C de votre plate-forme sur un format standardisé dont vous avez une copie sur le lien suivant :

https://docs.python.org/3.6/library/datetime.html#strftime-and-strptime-behavior

Pour lire une date à partir d'une chaine de caractères et générer un objet datetime, nous allons utiliser la méthode `strptime(string, format)`.

```
In [4]: date_object = datetime.strptime('Jun 1 2005  1:33PM', '%b %d %Y %I:%M%p')
```

Pour créer une chaine de caractères représentant un objet datetime, nous utiliserons la méthode `strftime(format)`.

```
In [5]: date_object.strftime('%Y-%m-%d %H:%M:%S.%f')
Out[5]: '2005-06-01 13:33:00.000000'
```

Par défaut, la date est en anglais. Pour travailler avec les informations de date en français, il faut modifier la locale :

```
In [6]: date_object.strftime('%a %d %B %Y')
Out[6]: 'Wed 01 June 2005'

In [7]: import locale

In [8]: locale.setlocale(locale.LC_TIME, 'fr_FR')
Out[8]: 'fr_FR'

In [9]: date_object.strftime('%a %d %B %Y')
Out[9]: 'Mer 01 juin 2005'
```

Les durées

Manipuler des dates, c'est manipuler les différences entre ces dates soit des durées. Elles sont représentées par des objets `datetime.timedelta`. La différence entre deux instants permet d'obtenir une durée.

```
In [10]: duree = datetime(2017, 12, 15) - datetime(2015, 12, 16)

In [11]: print(duree)
730 days, 0:00:00

In [12]: duree
Out[12]: datetime.timedelta(730)
```

On peut évidemment instancier un objet de type timedelta et ainsi déterminer des instants à partir d'autres instants.

```
In [13]: cuisson_oeuf = timedelta(seconds=180)

In [14]: datetime.now() + cuisson_oeuf
Out[14]: datetime.datetime(2017, 9, 10, 12, 12, 52, 776483)
```

Des opérations mathématiques sont donc possibles entre des objets de type datetime (soustraction), timedelta (addition et soustraction), datetime et timedelta (addition et soustraction) et timedelta et des valeurs numériques (multiplication et division).

Les objets de type timedelta ne sont pas très adaptés à la représentation des durées. Elles ne proposent pas de méthodes de parsing ou de formatage de la donnée. L'information de durée ne peut être extraite qu'en jour, secondes et microsecondes.

Dates et heures

Le module datetime propose des objets de type `datetime.date` et `datetime.time` pour représenter une date (jour) et un instant dans une journée. Il est possible soit d'instancier spécifiquement ces données, soit de les extraire d'un objet de type datetime.

```
In [15]: ep8 = datetime(2017, 12, 15)

In [16]: ep8.date()
Out[16]: datetime.date(2017, 12, 15)

In [17]: from datetime import time

In [18]: show_time = time(20, 30)

In [19]: print(show_time)
20:30:00
```

Des objets immuables

Les informations liées au temps sont des données critiques. Afin de ne pas les altérer par accident, les objets sont immuables, ils ne peuvent pas être modifiés. La classe datetime expose une méthode `replace()` qui permet de modifier un attribut de l'objet mais celle-ci retourne une copie de l'objet.

```
In [20]: ep8.replace(year=2019, day=20)
Out[20]: datetime.datetime(2019, 12, 20, 0, 0)

In [21]: print(ep8)
2017-12-15 00:00:00
```

Si vous souhaitez modifier un objet, il faut donc le faire explicitement.

Le module calendar

Le module calendar offre toutes les fonctionnalités pour gérer un calendrier ou une date dans un calendrier. Le module propose avant tout un certain nombre de constantes et de fonction afin d'accéder à des données usuelles.

```
In [22]: import calendar
In [23]: calendar.EPOCH
Out[23]: 1970
In [24]: calendar.MONDAY
Out[24]: 0
In [25]: calendar.TUESDAY
Out[25]: 1
In [26]: calendar.January
Out[26]: 1
```

Pour Python, lundi est donc représenté par la valeur 0 et janvier par 1. Pour éviter toute confusion, il est donc impératif d'utiliser ces constantes.

Le module calendar fourni également des constantes et méthodes permettant de manipuler les dates ou les nombres de jours :

```
In [27]: calendar.mdays
Out[27]: [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

In [28]: calendar.isleap(2018) # annee bissextile
Out[28]: False

In [29]: calendar.leapdays(2000, 2018)
Out[29]: 5

In [30]: calendar.weekday(2017, 12, 13)
Out[30]: 2
```

Le module calendar propose également de quoi mettre en forme des informations de calendrier

```
In [31]: calendar.prmonth(2017, 12)
   décembre 2017
Lu Ma Me Je Ve Sa Di
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Cette instruction est équivalente à `print(calendar.month(2017, 12))`.

Le module calendar permet également de manipuler un calendrier sous forme d'objet.

Les expressions rationnelles

Les expressions rationnelles, expressions régulières ou simplement RegEx sont un puissant moyen pour rechercher ou isoler une chaine de caractères. Elles permettent le recherche d'un motif ou la validation d'une chaine de caractères.

En Python, elles sont gérées par le module `re`

https://docs.python.org/3/library/re.html

Définir un motif

Utiliser les RegEx consiste avant tout à définir un *motif* décrivant la chaine de caractère à rechercher ou valider. Ce motif est une chaine de caractères qui va décrire ce qui est attendu. Le moteur de RegEx cherchera à identifier ce motif dans une chaine de caractères.

La suite de ce document présente une introduction à la définition d'un motif. La documentation citée en référence est bien plus exhaustive.

Un motif de base

Le motif le plus simple est une chaine de caractères *strict*. 'chat' par exemple peut être sera identifié dans la chaine de caractères 'chat' mais aussi 'chaton', 'achat' et 'un chat noir'.

La position d'un motif

Le symbol `^` en premier caractère est un marqueur indiquant que le motif doit être en début de chaine. Ainsi, '^chat' identifie la chaine 'chat' dans les chaines de caractères 'chat' et 'chaton' mais pas dans 'achat' ni dans 'un chat noir'.

De même, le symbol `$` en dernier caractère est un marqueur indiquant que le motif doit être en fin de chaine. Ainsi, 'chat$' identifie la chaine 'chat' dans les chaines de caractères 'chat' et 'achat' mais pas dans 'chaton' ni dans 'un chat noir'.

Variation sur un caractère (classes de caractères)

Les crochets permettent de définir quel caractère sera attendu parmi ceux entre crochet à l'emplacement des crochets. Dans les crochets, les caractères `-` et `^` ont un comportement spécial décrit ci-dessous.

- `[aeiouy]` signifie un caractère parmi ceux entre crochets
- `[a-z]` signifie un caractère entre a et z en minuscule.
- `[a-zA-Z]` signifie un caractère entre a et z en minuscule ou capitale.
- `[a-z-]` signifie un caractère entre a et z en minuscule ou un tiret.
- `[^aeiouy]` le `^` est une négation, ici, il est attendu n'importe quel caractère sauf un de ceux entre crochets.
- `.` signifie n'importe quel caractère.

Les crochets signifient donc ou.

Nombre d'occurrences

Un caractère ou une classe de caractères peut être attendu plusieurs fois. La définition d'un motif permet donc d'indiquer combien de fois un caractère est attendu en fonction du symbole suivant placé après :

- `*` : 0, 1 ou plusieurs occurrences
- `+` : 1 ou plusieurs occurrences
- `?` : 0 ou 1 occurrences
- `{m}` : strictement m occurrences
- `{m,n}` : entre m et n occurrences
- `{,n}` : de 0 à n occurrences
- `{m,}` : plus de m occurrences

Les groupes de caractères

Les parenthèses permettent de définir des groupes de caractères qui sont attendus dans leur ensemble. Les parenthèses définissent une relation et.

Un groupe peut être combiné à d'autres marqueurs :

- `(chat)+` permet d'identifier 'chatchat' dans 'tchatchatcha'.
- `([rf]i)+` identifie 'riri' et 'fifi' (et 'firi' ou 'rifi' ou 'rifiri'…)
- `(Android|iOs)` identifie soit 'Android' soit 'iOs'.

Utiliser le module re

Une fois le motif défini, il faut l'utiliser avec le module `re`.

Identifier la présence d'un motif

Le module re propose 3 fonctions principales :

- `re.search(pattern, seq)` recherche le pattern dans seq
- `re.match(pattern, seq)` est similaire à search mais recherche en début de chaine
- `re.fullmatch(pattern, seq)` est similaire à search mais la chaine complète doit correspondre au motif.

Si la fonction trouve une correspondance, un objet de type `_sre.SRE_Match`, communément appelé *Match Object*, est retourné sinon None.

Une RegEx peut également être utilisée comme un objet. La fonction `re.compile(pattern)` retourne un objet de type `_sre.SRE_Pattern` dédié à l'identification du motif `pattern`. Cet objet dispose de méthodes similaires aux fonctions précédentes.

Les informations d'un Match Object

Un Match Object expose 3 fonctions principales. Le paramètre optionnel `subgroup` est un entier ou une chaine de caractères.

- `group([subgroup])` : retourne le motif trouvé (sans paramètre ou pour une valeur 0) ou celui du sous-groupe.
- `start([subgroup])` : retourne l'indice de début du motif trouvé (sans paramètre ou pour une valeur 0) ou de celui du sous-groupe.
- `end([subgroup])` : retourne l'indice de fin du motif trouvé (sans paramètre ou pour une valeur 0) ou de celui du sous-groupe.

Le paramètre sous-groupe représente un des groupes définis par des parenthèses. Nous mettons ici à contribution la deuxième fonction des groupes : définir des sous-motifs d'intérêt. Ceux-ci permettent de définir l'information réelle que nous souhaitons obtenir ou sa localisation (indices).

L'exemple suivant définit un motif de recherche de nombres décimaux ayant une précision minimale de deux chiffres. Les groupes permettent d'extraire la partie entière et la partie décimale.

```
In [1]: import re

In [2]: float_regex = re.compile('([0-9]+)\.([0-9]{2,})')

In [3]: ma_chaine = 'pi vaut 3.14'

In [4]: resultat = float_regex.search(ma_chaine)

In [5]: if resultat:
   ...:     print(resultat.group())
   ...:     print(resultat.group(0))
   ...:     print(resultat.group(1))
   ...:     print(resultat.group(2))
   ...:
3.14
3.14
3
14

In [6]: print(resultat.start())
8

In [7]: print(resultat.end())
12

In [8]: print(resultat.start(2))
10
```

Comme illustré à la ligne 5, le pattern de traitement des résultat est de vérifier l'existence d'un résultat par la propriété booléenne du type None.

La numérotation des groupes étant pas toujours évidente, la sémantique de description des motifs permet d'ajouter des marqueurs. Ainsi, un groupe avec la sémantique suivante `(?P<name>…)` définira une étiquette `name` qui permettra d'identifier le groupe.

```
In [9]: float_regex = re.compile('(?P<whole>[0-9]+)\.(?P<fraction>[0-9]
{2,})')

In [10]: resultat = float_regex.search(ma_chaine)

In [11]: if resultat:
    ...:     print('1', resultat.group(1))
    ...:     print('whole', resultat.group('whole'))
    ...:     print('fraction', resultat.group('fraction'))
    ...:     print('start fraction', resultat.start('fraction'))
    ...:
1 3
whole 3
fraction 14
start fraction 10
```

Trouver plusieurs correspondances

Les méthodes précédentes ne permettent de trouver qu'une seule correspondance, la première du texte. Nous avons deux autres méthodes à disposition pour trouver toutes les occurrences.

- `findall(seq)` : envoi une liste de toutes les occurrences sous forme de chaine de caractères
- `finditer(seq)` : est un itérateur retournant des match objects

Les substitutions

Les RegEx peuvent être utilisées pour substituer du texte grâce à la méthode `sub(substr, str[, count])`. Le paramètre count permet de limiter le nombre de substitutions.

```
In [12]: ma_chaine = 'pi vaut 3.14 et e vaut 2.72'

In [13]: float_regex.sub('quelque chose', ma_chaine)
Out[13]: 'pi vaut quelque chose et e vaut quelque chose'
```

Mise en pratique

Le navires de la marine US sont identifiés dans un texte par le motif « USS » (pour United States Ship) suivi d'un tiret et de 1 à 4 chiffres. Ceci peut être représenté par le motif '`USS-[0-9]{1,4}`'.

Nous pouvons commencer par définir un objet qui identifiera ce motif.

```
In [1]: import re

In [2]: us_ship = re.compile('USS-[0-9]{1,4}')
```

Grâce à cet objet, nous pouvons valider le nom des prochaines références :

```
In [3]: def validate_ship_name(ship_name):
   ...:     if us_ship.fullmatch(ship_name):
   ...:         return 'ok'
   ...:     else:
   ...:         return 'wrong name'
   ...:

In [4]: validate_ship_name('USS-234')
Out[4]: 'ok'

In [5]: validate_ship_name('USS-23433')
Out[5]: 'wrong name'
```

Une validation doit se faire sur l'intégralité de la chaine, c'est la raison pour laquelle nous utilisons `fullmatch`. La méthode `search` retourne `True` pour le dernier cas car le motif est présent dans 'USS-234222'.

Nous devons travailler sur un texte contenant des noms de navires américains et britanniques. Ces derniers ont un nom similaire : à la place de USS, ils utilisent HMS (Her Majesty Ship). La RegEx permet d'extraire l'information de navire qui nous intéresse.

```
In [6]: text_abstract = 'The HMS-45 and the USS-1138 sailed north'

In [7]: us_match = us_ship.search(text_abstract)

In [8]: if us_match:
   ...:     print('US Ship is', us_match.group())
   ...:
US Ship is USS-1138
```

Nous pouvons complexifier cette RegEx pour identifier tous les bateaux, mais pas les sous-marins qui sont en SS-numéro. Nous voulons identifier tous les navires de surface et adapter la chaine en fonction qu'il soit américain ou britannique

```
In [9]: surface_ship = re.compile('(US|HM)S-[0-9]{1,4}')

In [10]: text_abstract = 'The HMS-45 and the USS-1138 sailed toward the SS-12'

In [11]: surface_ship = re.compile('(?P<cid>(US|HM)S)-[0-9]{1,4}')

In [12]: text_abstract = 'The HMS-45 and the USS-1138 sailed toward the SS-12'

In [13]: for match in surface_ship.finditer(text_abstract):
    ...:     if match.group('cid') == 'USS':
    ...:         print('US Ship', match.group())
    ...:     elif match.group('cid') == 'HMS':
    ...:         print('UK Ship', match.group())
    ...:     else:
    ...:         print('Something went wrong')
    ...:
UK Ship HMS-45
US Ship USS-1138
```

Les interfaces graphiques avec Tkinter

Tkinter est une bibliothèque permettant de créer des interfaces graphiques basée sur Tk. La bibliothèque fait parti de l'installation Python de base. Son grand avantage est donc la portabilité.

Cette partie est une introduction à Tkinter. Pour une documentation exhaustive, voir :

https://docs.python.org/3.6/library/tkinter.html

Hello World

```
from tkinter import *

fenetre = Tk()

label = Label(fenetre, text="Hello World")
label.pack()

fenetre.mainloop()
```

`Tk()` est la classe fondamentale qui va gérer les fenêtres. La méthode mainloop() va déclencher la boucle d'interaction.

Les widgets

Tkinter propose un ensemble de composants graphiques. Quelques uns seront présentés ici.

Leur utilisation consiste toujours en leur création puis leur positionnement. Le premier paramètre est toujours le conteneur auquel ils sont associés. D'autres attributs servent à les paramétrer.

Pour agencer les composants, il faudra faire appel à une de ces trois méthodes :

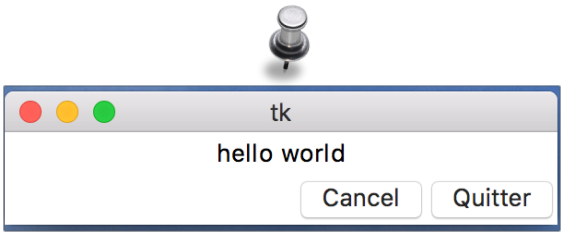
- `pack()` qui dispose les composants les un à la suite des autres
- `place()` qui permet un placement pixel-perfect et qui ne sera donc pas abordé ici
- `grid()` qui permet un placement sous forme de grille.

Un conteneur (fenêtre ou Frame) ne peut avoir qu'un type de gestion de widgets.

La méthode pack

La méthode `pack()` accepte un paramètre optionnel `side`. Celui-ci peut avoir pour valeur `TOP` (défaut), `BOTTOM`, `LEFT` et `RIGHT`.

Le principe de pack est de diviser l'espace en deux zones et de placer le widget du coté déclaré par `side`. L'autre coté est destiné à recevoir le composant suivant.




```
from tkinter import *
fen = Tk()
fen.minsize(300, 40)
label = Label(fen, text='hello world')
label.pack()

button_quit = Button(fen, text='Quitter')
button_quit.pack(side=RIGHT)

button_cancel = Button(fen, text='Cancel')
button_cancel.pack(side=RIGHT)

fen.mainloop()
```

La méthode grid

La méthode `grid()` nécessite deux paramètre : `row` et `column`. Il s'agit des indices de la position attendue du composant. Des indices de 0 placent le composant en haut à gauche.

Les composants n'ont pas à être placés dans l'ordre.


```
from tkinter import *

fenetre = Tk()

value = 1

for i in range(3):
    for j in range(3):
        Button(fenetre, text=value).grid(column=i, row=j)
        value += 1

fenetre.mainloop()
```

Les boutons : action de l'utilisateur

L'action d'un bouton est défini par la fonction associée au paramètre `command`. La classe `Tk` fourni une méthode pour quitter l'application.

```
from tkinter import *

fenetre = Tk()

bouton=Button(fenetre, text="Fermer", command=fenetre.quit)
bouton.pack()

fenetre.mainloop()
```

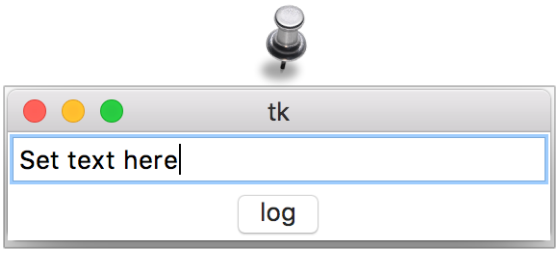
C'est bien évidemment la référence à la fonction qui est passée en paramètre. La fonction est exécutée par Tkinter lorsque le bouton est cliqué.

Nous pouvons évidemment définir des fonctions pour les boutons de notre application. Ces fonctions ont deux contraintes : elles ne peuvent pas avoir de paramètre et ne peuvent avoir aucun retour.

```
from tkinter import *

fenetre = Tk()

def log_text():
    print("hello world")

Button(fenetre, text='Log', command=log_text).pack()

fenetre.mainloop()
```

Les champs de saisie : information de l'utilisateur

Un composant de type Entry va afficher un champs de saisie. Mais il sera associé à un conteneur pour la donnée affichée (et donc saisie) par ce champs. Il y a plusieurs types pour ces conteneurs dépendant du type de donnée prise en charge.
Dans l'exemple suivant, un `StringVar()` valide une saisie de type chaine de caractères.




```
from tkinter import *

fenetre = Tk()

value = StringVar()
value.set('Set text here')

Entry(fenetre, textvariable=value, width=30).pack(side=TOP)

def log_value():
    print(value.get())

Button(fenetre, text='log', command=log_value).pack()

fenetre.mainloop()
```

Les cases à cocher : données prédéfinies

Une case à cocher est un composant qui a deux états : coché ou non. Le composant a donc 2 paramètres, `onvalue` et `offvalue`, auxquels on attribut la valeur que retournera le composant en fonction de son état.

Il y a donc évidemment des conteneurs de type `BoolVar`, mais ici, nous utilisons un `IntVar`.


```
from tkinter import *

fenetre = Tk()

checkValue = IntVar()
bouton = Checkbutton(fenetre, text='un label',
                     variable=checkValue,
                     onvalue=5, offvalue=0)
bouton.pack()

def get_value():
    if checkValue.get():
        print('value is', checkValue.get())
    else:
        print('not Checked')

Button(fenetre, text='log', command=get_value).pack()
```

Les Listbox : ajout et sélection

Une Listbox est un composant qui présente une collection d'information sous forme de liste.

Cette liste peut avoir des éléments initiaux mais doit accepter également l'ajout, la suppression et la sélection d'éléments. Le conteneur des données est un composant de type `Variable()`.

```
fenetre = Tk()

choices = Variable(fenetre, ('Java', 'Python', 'Swift'))
listbox = Listbox(fenetre, listvariable=choices,
                           selectmode='single')
listbox.pack()
```

Le paramètre `selectmode` permet de définir comment les valeurs peuvent être sélectionnées. Les valeurs sont :

- `SINGLE` : un seul choix
- `BROWSE` : un seul choix mais la sélection peut-être déplacée par lex flèches du curseur.
- `MULTIPLE` : plusieurs choix
- `EXTENDED` : plusieurs choix qui peuvent être étendus au clavier.

Pour ajouter un élément, il suffit de l'ajouter à la Listbox avec la méthode `insert()` en précisant l'indice. `0` ajoute en haut de la liste. La constante `END` ajoute en fin de liste. Si l'indice est supérieur au nombre d'éléments, l'élément est ajouté en fin de liste.

```
listbox.insert(END, 'Ruby')

listbox.insert(0, 'Pascal')

listbox.insert(10, 'Scala')
```

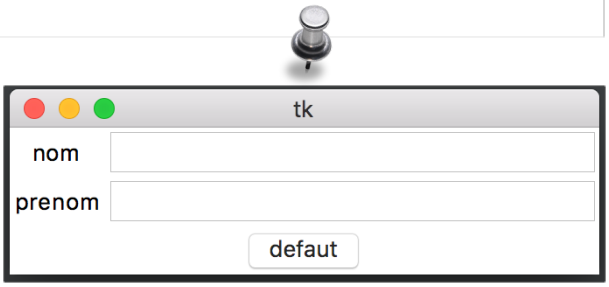
Pour récupérer la sélection, il faut commencer par interroger l'objet de type `Listbox`. La méthode `curselection()` retourne une liste des indices de tous les éléments sélectionnés. La méthode get() de l'objet de type Variable retourne la liste des éléments de la liste. On peut donc récupérer les éléments sélectionnés à partir des indices récupérés précédemment.

```
for indice in listbox.curselection():
    print(choices.get()[indice]
```

Organisation du code

Lorsque l'interface prend de l'ampleur, il est nécessaire de créer ses propres composants. Pour cela, nous allons regrouper des composants et leur logique dans des classes héritant de la classe `Frame`. La logique et la communication inter-comptants est implémentée dans des méthodes dédiées.

```
class UserInfo(Frame):
    def __init__(self, master=None, cnf={}, **kw):
        Frame.__init__(self, master, cnf, **kw)
        Label(self, text='nom').grid(column=0, row=0)
        Label(self, text='prenom').grid(column=0, row=1)

        self.last_name_value = StringVar()
        self.first_name_value = StringVar()

        Entry(self, textvariable=self.last_name_value, width=30)\
            .grid(column=1, row=0)
        Entry(self, textvariable=self.first_name_value, width=30) \
            .grid(column=1, row=1)

    def set_defaut_values(self):
        self.first_name_value.set('John')
        self.last_name_value.set('Doe')
from tkinter import *

root = Tk()
root.title('Demo')

user_frame = UserInfo(root)
user_frame.pack()

Button(root, text='defaut',
       command=user_frame.set_defaut_values).pack()

root.mainloop()
```



Ici, le bouton déclenche un comportement de la frame en appelant une méthode de cette dernière.

Si il est nécessaire d'extraire des données de la frame, il est nécessaire d'implanter un mécanisme de *callbacks* en passant une méthode (la *callback*) à la frame en paramètre.

La documentation

Il est important de documenter son code pour deux destinataires distincts : celui qui est appelé à l'utiliser et celui qui est appelé à le maintenir. En Python, il y a 3 outils :

- Les bonnes pratiques de nommage et d'organisation du code
- Les commentaires
- Les docstrings

Avec sa syntaxe basée sur l'indentation et l'incitation de l'usage des règles de la pep8, L'organisation du code est assez uniforme.

Les commentaires et les bonnes pratiques de nommage

En Python, un commentaire commence par un croisillon # (Qui n'est pas un dièse : ♯). Le terme dièse étant le plus répandu, on l'utilisera également.

Un commentaire peut être placé n'importe où dans le code et à pour but d'informer sur les lignes de code suivantes. Il doit surtout servir à expliquer pourquoi le code suivant a été écrit de cette manière.

Le destinataire des commentaires est le développeur qui fait de la maintenance.

L'usage des commentaires doit être limité au strict nécessaire. Leurs deux défauts sont :

- Ils sont une autre langue que votre code, ils sont donc une distraction lorsque le développeur lit le code.
- Ils sont rarement maintenus ce qui peut augmenter la source de confusion

De manière générale, favorisez la lisibilité de vote code par rapport à l'usage de commentaires.

- Utilisez des noms pour les variables (données)
- Utilisez des verbes pour les fonctions (actions)
- Utilisez des noms explicites et sans ambiguïté communs au contexte

Les docstrings

Les docstrings sont des chaînes de caractères encadrés de triple-double-cotes places à des endroits spécifiques : en tête de module, après la déclaration d'une classe et après la déclaration d'une fonction ou d'une méthode. Les conventions sont spécifiées dans la pep 257 :

https://www.python.org/dev/peps/pep-0257/

Les docstrings sont exploitées par certains outils : la méthode spéciale __doc__() et la fonction help() servent à afficher leur contenu. Les IDEs permettent leur affichage contextuel. Les outils de génération de documentation tel que Sphinx génèrent une documentation grâce aux doctests.

Les doctests doivent donc informer sur comment utiliser le module, la classe ou la fonction. Ils sont destinés au développeur qui utilise les composants et qui ne lira certainement pas les sources.

Le document de référence pour l'organisation d'une documentation est l'article de Thomas Cokelaer suivant : thomas-cokelaer.info/tutorials/sphinx/docstring_python.html

Utilisation du shell interactif (ou iPyhton)

Le shell interactif est un puissant outil qui va nous permettre de *tester* le comportement d'instructions Python. Vous pouvez le lancer par la commande `python` dans un terminal. Le shell interactif peut également être intégré à votre environnement de développement, ce qui est le cas avec PyCharm.

IPython est un shell interactif évolué. Il propose par exemple la complétion, le rappel d'instructions par blocs (au lieu de lignes) ou des *commandes magiques*. De manière générale, préférez iPython à chaque fois que vous le pouvez. PyCharm par exemple remplacera le shell interactif par iPython à chaque fois que cela est possible.

Testez vos ressources

Le shell interactif vous permet de tester le code de vos modules. Pour cela, vous devez importer le module comme si votre code était dans un autre module.

```
In [1]: from training.projects.bank import bank

In [2]: account = bank.BankAccount("cz002", 2000)

In [3]: account.balance
Out[3]: 2000.0

In [4]: account.deposit(200)

In [5]: account.balance
Out[5]: 2200.0
```

Modifiez vos ressources

Lorsque vous apportez des corrections à vos modules, ils doivent être **rechargés** (et non importés de nouveau) dans le shell interactif. Pour recharger un module, il faut utiliser une fonction du module `importlib`.

```
In [6]: import importlib

In [7]: importlib.reload(bank)
Out[7]: <module 'training.projects.bank.bank' from '/Some/path/training-
python/training/projects/bank/bank.py'>

In [8]: account = bank.BankAccount("cz002", 2000)
```

Attention, recharger un module permet d'exécuter par la suite le code à jour mais ne recharge pas les instances créées. Il faut donc instance à nouveau les objets utilisés.

Shell interactif et affichage des données

L'affichage d'une donnée est réalisée par la fonction `print()`.

Le shell interactif a un comportement particulier. À chaque fois que vous validez une instruction, le shell interactif affiche le retour de cette instruction. Ceci vous aide à comprendre ce qui est retourné mais prête également à confusion puisqu'une valeur est affichée alors que vous n'avez pas saisi d'instruction d'affichage.

```
In [20]: # Une affectation ne retourne pas de valeur
In [21]: a = 10
In [22]: # une variable retourne sa valeur
In [23]: a
Out[23]: 10
In [24]: # La fonction print affiche la valeur mais n'a pas de
valeur de retour
In [25]: print(a)
10
In [26]: # L'appel à une fonction peut retourner une valeur
In [27]: import math
In [28]: math.sqrt(5)
Out[28]: 2.23606797749979
In [29]: # L'affectation d'un retour de fonction n'a pas de valeur
de retour
In [30]: b = math.sqrt(5)
```

Faites donc attention à bien interpréter le comportement du shell interactif.

Ce comportement se produit également avec ipython et donc de manière plus particulière avec les Jupyter Notebooks car ils sont basés sur ipython. Ainsi, lorsque vous exécutez une cellule d'instructions, le Notebook vous affiche le retour de la dernière instruction.

Python 2, une version obsolète

Lorsque Python est installé par défaut sur certains systèmes comme Linux ou Os X, il s'agit malheureusement encore de la version obsolète Python 2. Python 2 est une version obsolète car :

- Depuis 2010, Python 2 ne reçoit plus d'évolution
- Depuis 2015, la maintenance de Python 2 est irrégulière et ne suit plus de calendrier
- Le support de Python 2 prendra fin en 2020.
- Certaines dépendances commencent à abandonner la prise en charge de Python 2 comme Django à partir de 2018.

En conséquence,

- tout nouveau projet doit être en Python 3
- toute maintenance de code Python 2 doit s'adapter à une syntaxe Python 3 grâce au package future afin de faciliter la transition.

Il existe de nombreuses incompatibilités entre Python 2 et 3 qui rendent la migration difficile. Le package Future a été créé afin de faciliter l'adaptation du code en vue de cette migration. Il a pour but de permettre d'écrire du code Python 2 compatible Python 3 afin que la migration se fasse *en douceur* au fil de l'eau. Voici un exemple avec la fonction `print()`.

Exemple avec la fonction print pour la compatibilité Python 2/3

En Python 3, `print` est une fonction alors qu'en Python 2, c'était également une instruction. Ainsi, en Python 2, il est possible de trouver des instructions de ce type.

```
In [1]: print "Hello World"
```

Celles-ci ne sont pas autorisées en Python 3. Pour assurer la portabilité du code Python 2, ajoutez à ces modules l'import suivant qui provoquera une erreur si il subsiste des instructions `print`.

```
In [1]: from __future__ import print_function
In [2]: print("Hello World")
Hello World
In [3]: print "Hello World"
  File "<ipython-input-3-085e5f98785e>", line 1
    print "Hello World"
                       ^
SyntaxError: Missing parentheses in call to 'print'
```

Une exception sera donc systématiquement levée, votre code ne pourra donc plus fonctionner tant qu'il ne sera pas mis à jour Python 3. Et sera ainsi prêt pour une migration.