STRUCTURER DES PROGRAMMES SELON UN ALGORITHME
MAÎTRISER LES ÉLÉMENTS D'UN LANGAGE POUR ÉCRIRE UN PROGRAMME
DÉBOGUER ET TESTER UN PROGRAMME
MAÎTRISER LA SYNTAXE DU LANGAGE PYTHON
ACQUÉRIR LES NOTIONS ESSENTIELLES DE LA PROGRAMMATION OBJET

PYTHON, INITIATION À LA PROGRAMMATION

SUPPORT DE FORMATION

FRÉDÉRIC GAURAT

Python, Initiation à la Programmation

Frédéric GAURAT

Version 1.0, 2022-06-10

Table des matières

Chapitre 1. Objectifs et Prérequis

1.1. Objectifs pédagogiques

Python est un langage de programmation multiplateforme permettant le développement d'une grande variété d'applications. Vous en maîtriserez sa syntaxe, ses principaux mécanismes et son paradigme Objet. Vous découvrirez les fonctionnalités de la bibliothèque de modules standards, implémenterez des interfaces graphiques, accéderez aux données d'une base tout en utilisant des outils permettant de tester et d'évaluer la qualité du code produit.

- Structurer des programmes selon un algorithme
- Maîtriser les éléments de lexique et de syntaxe d'un langage pour écrire un programme
- Compiler et exécuter un programme
- Déboguer et tester un programme
- Maîtriser la syntaxe du langage Python
- Acquérir les notions essentielles de la programmation objet

1.2. Participants

Toute personne devant apprendre à programmer avec Python.

1.3. Prérequis

Aucune connaissance particulière.

Chapitre 2. Introduction à la programmation

2.1. Qu'est ce qu'un programme ?

Un programme est un ensemble d'instructions qui s'exécutent dans un ordinateur [wprog]. - Un programme source est un code écrit par un informaticien dans un langage de programmation. Il peut être compilé vers une forme binaire ou directement interprété. - Un programme binaire décrit les instructions à exécuter par un microprocesseur sous forme numérique. Ces instructions définissent un langage machine.

Un programme fait généralement partie d'un logiciel que l'on peut définir comme un ensemble de composants numériques destiné à fournir un service informatique. Un logiciel peut comporter plusieurs programmes. On en retrouve ainsi dans les appareils informatiques (ordinateur, console de jeux, guichet automatique bancaire…), dans des pièces de matériel informatique, ainsi que dans de nombreux dispositifs électroniques (imprimante, modem, GPS, téléphone mobile, machine à laver, appareil photo numérique, décodeur TV numérique, injection électronique, pilote automatique…).

L'objectif d'un programme est de réaliser une tâche spécifique. Cette tâche est définie par un algorithme. Un algorithme est une suite de instructions qui s'exécutent dans un ordinateur.

2.2. Qu'est ce qu'un algorithme ?

Un algorithme est une suite finie et non ambiguë d'instructions et d'opérations permettant de résoudre une classe de problèmes. [walg]

Le mot algorithme vient d'Al-Khwârizmî, nom d'un mathématicien persan du ixe siècle.

Le domaine qui étudie les algorithmes est appelé l'algorithmique. On retrouve aujourd'hui des algorithmes dans de nombreuses applications telles que le fonctionnement des ordinateurs, la cryptographie, le routage d'informations, la planification et l'utilisation optimale des ressources, le traitement d'images, le traitement de textes, la bio-informatique, etc.

2.3. Paradigmes de programmation

2.3.1. Définition

Le paradigme de programmation est la façon (parmi d'autres) d'approcher la programmation informatique et de formuler les solutions aux problèmes et leur formalisation dans un langage de programmation approprié [wpparad]. **Ce n'est pas de la méthodologie contenant une méthode**; cette dernière organise le traitement des problèmes reconnus dans l'écosystème concerné pour aboutir à la solution conceptuelle et programme exécutable.

2.3.2. Les différents paradigmes

Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme en situation d'exploitation. - Programmation orientée objet : les développeurs peuvent considérer le programme comme une collection d'objets en interaction. - Programmation fonctionnelle : un programme peut être vu comme une suite d'évaluations de fonctions sans états.

Chapitre 3. Introduction à la construction d'un algorithme

3.1. Avant de coder, le pseudo-code

Comme indiqué plus haut, un algorithme est une suite d'instructions qui va décrire la résolution d'un problème. Il ne s'agit pas ici, de proposer une soltion avec un langage spécifique, mais bien de décrire la façon dont le problème va être résolu. C'est le pseudo-code qui va permettre de décrire, simplement, les différentes étapes de résolutions.

Il n'existe cependant pas de réelle convention pour le pseudo-code [pscod] mais, on peut citer les principaux mots clés qui font l'objet d'un relatif consensus : - L'affectation représentée par le signe: = - L'alternative représentée par la structure: SI (condition) ALORS (instruction) SINON (instruction) - La répétition: REPETER (nombre de fois) (instructions) - La répétition conditionnelle: TANT QUE (condition) FAIRE (instruction) - La fonction: FONCTION (nom) (paramètres) (suite d'instruction) - La séquence d'instruction: DEBUT (instructions) FIN Il existe bien d'autres conventions mais en général une douzaine de mots clés suffisent pour décrire la plupart des algorithmes et les rendre compréhensibles.[wpscod]

Exemple de pseudo-code pour lire et afficher un nombre:

```
DEBUT
    AFFICHER("Entrez un nombre")
    LIRE(n)
    AFFICHER("Le nombre est ", n)
FIN
```

Exemple de pseudo-code pour calculer la somme des nombres entre 1 et 100:

```
DEBUT
    somme = 0
    i = 1
    TANT QUE (i <= 100) FAIRE
        somme = somme + i
        i = i + 1
    FIN
    AFFICHER("La somme des nombres entre 1 et 100 est ", somme)
FIN
```

Chapitre 4. Introduction au langage Python

4.1. Un bref historique

Python est un langage de programmation **interprété**, **multi-paradigme** et **multiplateformes**. - interprété : un langage interprété se caractérise par le fait que le programme sera analysé à chaque exécution. Avec un compilateur,cette opération est réalisée une seule fois : au moment de la compilation. [winterp] - multi-paradigme : Python supporte plusieurs "formes" de programmation : **Fonctionnelle** et **Orientée objet** - multiplateformes : Une application développée avec Python peut s'exécuter sur plusieurs plateformes (Window, Linux, Unix) pour peu que la plateforme ait un interpréteur.

Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk et Tcl. [wpy]

4.2. Les versions de Python

La toute première version de Python date du 20/02/1991 et la dernière du 19/02/2021. [wiversion] Parmi les évolutions majeures, on trouve le support d'Unicode pour la version 2.0.

4.3. Python Enhancement Proposals (PEP)

Les PEPs sont des propositions d'améliorations du langage. Elles peuvent définir des évolutions techniques ou simplement de bonnes pratiques de programmation (PEP08). Elles sont toutes listées ici : https://www.python.org/dev/peps/

4.4. L'interpréteur Python

L'interpréteur Python peut être utilisé de 2 façons : - pour exécuter des instructions au fil de l'eau - pour exécuter un script

Exécution de l'interpréteur Python

```
$ python

Python 3.9.2 (default, Feb 19 2021, 17:43:04)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Exécution d'un premier exemple

```
$ python

Python 3.9.2 (default, Feb 19 2021, 17:43:04)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> python_est_bien = True
>>> if python_est_bien:
...     print("C'est pas faux")
...
C'est pas faux
>>>
```

4.5. Introduction à la syntaxe du langage

Les commentaires

```
# Ceci est un commentaire
une_valeur = 1  # Un autre commentaire en fin de ligne
                # et un troisième qui forme un bloc de commentaires
text = "# Ceci n'est pas un commentaire puisque dans une chaîne."
```

4.6. Utilisation de Python pour des traitements simples

Il est donc possible d'utiliser Python comme une simple calculatrice ou comme un "bac à sable". L'idée ici est de permettre au développeur de tester des bouts de code sans avoir à créer des scripts.

Il existe aussi un produit : https://jupyter.org/ qui offre la possibilité de créer des "blocs notes". C'est un outil simple qui permet de produire du code comme nous le ferions dans un carnet de notes. Il ne fait pas partie de la distribution standard de

Python mais on peut l'installer très facilement. Il s'intègre très bien dans Visual Studio Code .

4.7. Travailler avec des nombres [pynum]

Utilisation des nombres dans l'interpréteur Python

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 2 # Une division renvoie toujours un nombre à virgule flottante
4.0
>>> 8 / 3
2.6666666666666665
>>> 17 // 3  # Une division entière
5
>>> 17 % 3  # Le reste (modulo)
2
>>> 5 ** 2  # 5 élevé au carré
25
>>> 2 ** 7  # 2 élevé à la puissance 7
128
a = 2 # Stocker une valeur dans une variable
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined # Une erreur se produit si une variable
n'existe pas
```

4.8. Travailler avec des chaînes [pystr]

Utilisation des chaînes dans l'interpréteur Python

```
>>> 'Python est très bien'  # simple quotes
'Python est très bien'
>>> 'Python c\'est très bien'  # utilisation du \' pour permettre la '
"Python c'est très bien"
>>> "Python c'est très bien"  # ...double quotes
"Python c'est très bien"
>>> s = "Python c'est très bien"
>>> print(s)
"Python c'est très bien"
```

Manipuler les caractères de contrôle

```
>>> print('C:\nom\trivial') # ici nous avons deux caractères de contrôle :
\n (new line) et \t (tabulation)
C:
om  rivial
>>> print('C:\\nom\\trivial') # on peut doubler le caractère "\" pour
annuler son rôle
C:\nom\trivial
>>> print(r'C:\nom\trivial') # on peut préfixer la chaîne avec 'r' pour le
mode raw ( brut )
C:\nom\trivial
```

Chaîne de caractère sur plusieurs lignes

```
>>> print("""   # les triples quotes (simple ou double) sont utilisées
pour délimiter une chaîne de caractère multiligne.
... ligne 1     # on peut utiliser
... ligne 2
... ligne 3
... """)
```

Concaténation de Chaînes de caractère

```
>>> 'Py' 'thon'     # concaténation sur une ligne
>>> ('Py'           # concaténation sur plusieurs lignes
... 'thon')
'Python'
>>> a = 'Py'        # concaténation avec l'opérateur '+'
>>> b = 'thon'
>>> a+b
'Python'
```

Les chaînes de caractère sont des listes de caractères

```
>>> p = 'Python'
>>> p[0]        # Le premier caractère de la chaîne :  le premier
caractère d'une chaîne est celui à la position 0
'P'
>>> p[5]        # Le sixième caractère de la chaîne
'n'
```

Autre utilisation des chaînes de caractère comme des listes

```
>>> p[-1]       # Le dernier caractère peut être obtenu en utilisant
l'indice '-1'
'n'
>>> p[-2]       # L'avant-dernier caractère peut être obtenu en utilisant
l'indice '-2'
'o'
>>> p[-3]       # L'antépénultième caractère peut être obtenu en utilisant
l'indice '-3'
'h'
```

Pour résumer :

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
   0   1   2   3   4   5
  -6  -5  -4  -3  -2  -1
```

Il est aussi possible d'accéder à des segments de la chaîne. .Segment de la chaîne

```
>>> p[0:2]  # Tout les caractères de la position 0 (inclus) à 2 (exclue)
'Py'
>>> p[2:5]  # Tout les caractères de la position 2 (inclus) à 5 (exclue)
'tho'
```

Le début du segment est toujours inclus et la fin toujours exclue. Il est aussi possible de simplifier cette notation en utilisant des valeurs implicites .Valeur par défaut

```
>>> p[:2]  # Du début à la position 2 (exclue)
'Py'
>>> p[2:]  # de la position 2 (inclus) à la fin
'thon'
```

Les chaînes de caractères, en Python, ne peuvent pas être modifiées. On dit qu'elles sont **immuables**. Affecter une nouvelle valeur à un indice dans une chaîne produit une erreur.

4.9. Travailler avec des listes (une première approche) [pylst]

La liste est une structure de donnée permettant de regrouper plusieurs valeurs.Les éléments de la liste sont placés entre et sont séparés par des virgules. Une liste peut contenir des valeurs de différent type, mais dans la réalité ce n'est pratiquement jamais le cas.

Exemple de liste

```
>>> l = ["valeur 1", "valeur 2","valeur 3","valeur 4"]
>>> l
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4']
```

Les éléments d'une liste ont des indices

```
>>> l[0]
'valeur 1'
```

On peut également travailler sur des segments (slices)

```
>>> l[1:3]
['valeur 2', 'valeur 3']
```

Toutes les opérations par tranches renvoient une nouvelle liste contenant les éléments demandés. On parle de copie superficielle (shallow copy)

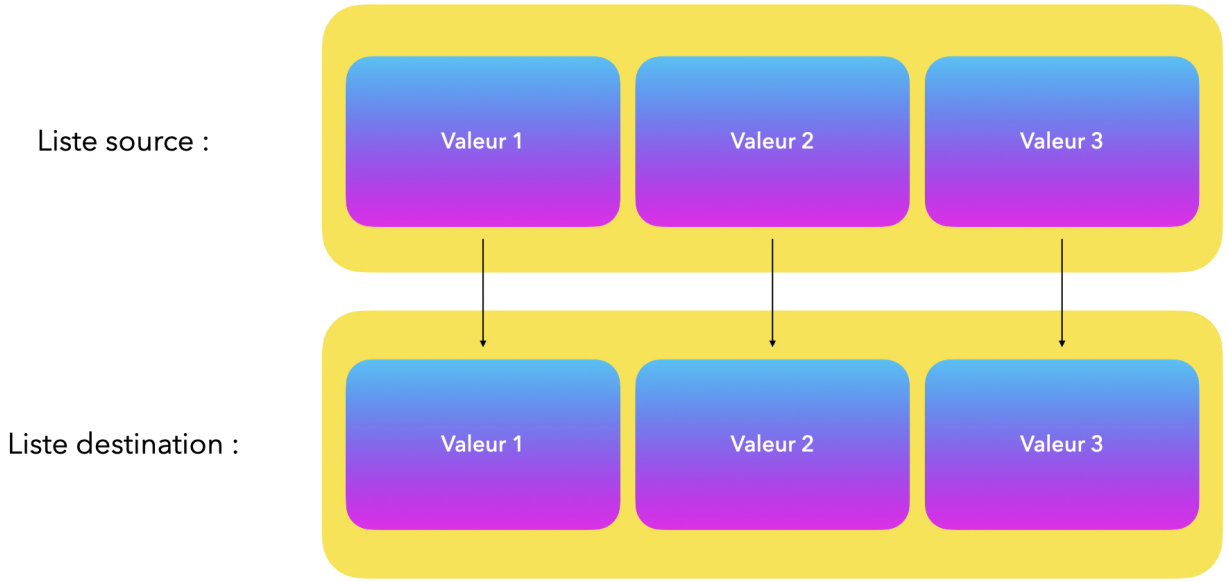


Figure 1. Copie superficielle simple (shallow copy)

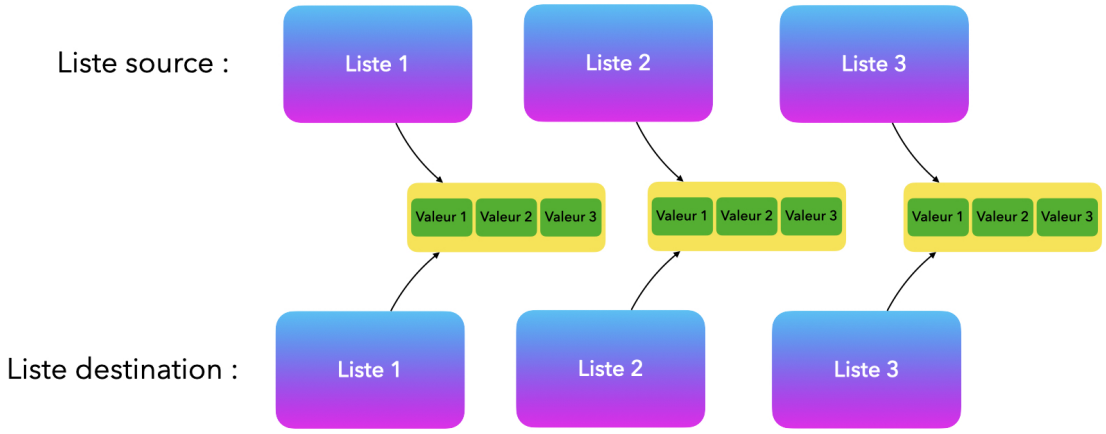


Figure 2. Copie superficielle complexe (shallow copy)

Exemple de liste

```
>>> l = ["valeur 1", "valeur 2","valeur 3","valeur 4"]  # liste source
>>> l
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4']
>>> l2 = l[:]                                           # Copie de la
liste source
>>> l2
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4']
>>> l2[0] = "autre valeur"                              # Modification de
la liste destination
>>> l
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4', 'valeur 7'] # La liste
source n'a pas changé
>>> l2
['autre valeur', 'valeur 2', 'valeur 3', 'valeur 4', 'valeur 7']
```

Comme pour les chaînes, il est possible d'effectuer des opérations de concaténation avec + ou la méthode `append()` La concaténation avec + renvoie une nouvelle liste. La méthode `append()` ajoute l'élément à la liste courante

Concaténation avec + et `append()`

```
>>> l+['valeur 5', 'valeur 6']              #  Ajout de plusieurs valeurs
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4', 'valeur 5', 'valeur 6']
>>> l.append('valeur 7')                    # Ajout d'une valeur
>>> l
['valeur 1', 'valeur 2', 'valeur 3', 'valeur 4', 'valeur 7'] # 'valeur 5',
'valeur 6' ne sont pas dans la liste
```

4.10. Un premier programme

Voici un premier exemple reprenant les éléments déjà vu la suite de Fibonacci [pyfibo].

Suite de Fibonacci

```
>>> a, b = 0, 1         # Double affectation
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

Chapitre 5. Contrôler le flot d'exécution d'un programme

Dans ce chapitre nous allons aborder les éléments permettant le contrôle du flot d'exécution du programme.

5.1. If

La structure d'un `if` est la suivante:

Structure du if

```
>>> a = 1
>>> if a<0:
...     print("inf à 0")
... elif a>0:
...     print("sup à 0")
... else:
...     print(" == 0")
...
>sup à 0
```

5.2. For

L'instruction `for` est différente de ce qu'il est possible de trouver dans d'autres langages. En Python, le `for` est plus proche d'un `foreach`, il est conçu pour faciliter l'itération sur des séquences de valeur.

Structure de la boucle `for`

```
>>> l = ["valeur 1", "valeur 2","valeur 3","valeur 4"]
>>> for e in l:
...     print(e)
...
valeur 1
valeur 2
valeur 3
valeur 4
```

5.3. La fonction range()

Pour retrouver le `for` du C (ou d'autres langages), Python introduit la fonction `range()`. La fonction `range()` génère des suites arithmétiques. L'appel de la fonction range est documenté ainsi :

Documentation de la fonction `range()`

```
range(start, stop[, step])
```

La fonction `range`

```
stop = 3
>>> for i in range(stop): # itération de 0 à 3 (exclue)
...     print(i)
...
0
1
2

start = 1
stop = 3
>>> for i in range(start,stop): # itération de 0 à 3 (exclue)
...     print(i)
...
1
2

>>> start = 1
>>> stop = 10
>>> step = 2
>>> for i in range(start,stop,step):  # itération de 0 à 10 (exclue) par
pas de 2 (de 2 en 2)
...     print(i)
...
1
3
5
7
9
```

Convertir un range en une list

```
>>> list(range(4))
[0, 1, 2, 3]
```

5.4. Utilisation de break, continue et else dans une boucle

Le rôle de l'instruction break, est d'interrompre la boucle `for` ou `while`. L'instruction `else` permet l'exécution de code si la boucle n'a pas été interrompue par une instruction break. [pyloop]

C'est une mauvaise pratique algorithmique que d'interrompre une boucle. Mieux vaut déterminer la condition d'arrêt proprement.

5.5. L'instruction pass

L'instruction pass ne fait rien. Elle n'est présente que pour garantir une syntaxe correcte.

Utilisation de pass

```
>>> a = 1
>>> if a==2:
...     pass    # Pas de traitement pour ce bloc if (pour le moment)
...             # pass permet d'exécuter le code tout de même
>>> print(a)
1
```

Chapitre 6. Les fonctions

Les fonctions permettent de factoriser du code afin de le réutiliser à différents endroits de votre application.

Il existe une différence entre **fonction** et **procédure** : une **fonction** renvoie une valeur, une **procédure** n'en renvoie pas .

La suite de Fibonacci sous forme de procédure

```
>>> def fib(n):    # Suite de  Fibonacci jusqu'à n
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ') # la fonction print peut prendre un second
paramètre end indiquant le caractère à utiliser à la fin de l'affichage
...         a, b = b, a+b
...     print()
...
>>> fib(100) # Appel de la fonction
0 1 1 2 3 5 8 13 21 34 55 89
```

La suite de Fibonacci sous forme de fonction

```
>>> def fib2(n):    # Suite de  Fibonacci jusqu'à n
...     result = []
...     a, b = 0, 1
...     while a < n:
...          result.append(a)
...         a, b = b, a+b
...     return result
...
>>> fib2(100) # Appel de la fonction
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

6.1. Portée des variables

Une variable déclarée dans une fonction est dite **locale**, elle n'est visible que dans la fonction. Une variable déclarée hors du bloc d'une fonction est dite **globale**, elle est visible partout mais ne sera accessible qu'en lecture à l'intérieur des fonctions.

Une variable globale

```
>>> a = 1
>>> def f():
...     print(a)
...
>>> f()
1
```

Une variable globale avec tentative de modification

```
>>> a=1
>>> def f():
...     a=2         #ici on redéfinit une nouvelle variable à l'intérieur
de la fonction, impossible donc de modifier la variable globale
...     print(a)
...
>>> f()
2
```

Il est possible de déclarer explicitement que nous souhaitons modifier une variable globale :

```
>>> a=1
>>> def f():
...     global a # On déclare que nous souhaitons travailler avec la
variable globale a
...     a=2
...
>>> f()
>>> print(a)
2
```

6.2. Les paramètres par défaut

Python permet de définir des valeurs par défaut pour les paramètres des fonctions.

Paramètres par défaut de la suite de Fibonacci

```
>>> def fib2(n=100):    # Suite de  Fibonacci jusqu'à n (100 est la valeur
par défaut)
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)
...         a, b = b, a+b
...     return result
...
>>> fib2() # Appel de la fonction sans préciser la valeur de la suite
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fib2(200) # Il est toujours possible de passer un paramètre à la
fonction
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Les valeurs par défaut sont évaluées lors de la définition, pas lors de l'exécution. Autrement la valeur par défaut est évaluée **une seule fois**

Evaluation des valeurs de paramètres par défaut [pydef]

```
>>> def f(a, L=[]): # L est initialisé avec une liste vide, une seule
fois, tous les autres appels
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

Evaluation des valeurs de paramètres par défaut [pydef]

```
>>> def f(a, L=None):
...     if L is None:
...             L = [] # Pour éviter ce comportement, on déclare
localement la liste
...     L.append(a)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[2]
>>> print(f(3))
[3]
```

6.3. Les paramètres par position et par mot-clef

Il est possible de passer des paramètres de deux façons en Python : - En respectant la position des paramètres tel que définie par la fonction, ce sont les paramètres positionnels. - En donnant leur nom lors de l'appel, ce sont les paramètres nommés.

Passage de paramètres par position ou par nom

```
>>> def direBonjour(nom,prenom):
...     print('Bonjour '+nom+' '+prenom)
...
>>> direBonjour('DURAND','Robert') # passage par position
Bonjour DURAND Robert

>>> direBonjour(nom='DURAND',prenom='Robert') # passage par nom
Bonjour DURAND Robert

>>> direBonjour(prenom='Robert',nom='DURAND')# avec le passage par nom,
inutile de respecter l'ordre,
Bonjour DURAND Robert

>>> direBonjour('DURAND',prenom='Robert')# on peut mélanger les 2 mais les
premiers paramètres doivent être passés par position
Bonjour DURAND Robert

>>> direBonjour('Robert',nom='DURAND')# ici, on passe le premier argument
par position (le nom) et on passe un second argument nommé qui est aussi
le nom => erreur
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: direBonjour() got multiple values for argument 'nom'
```

6.4. Nombre variable de paramètres

Il est possible de déclarer une fonction afin qu'elle puisse recevoir un nombre variable de paramètre. Toutefois, comme il est possible de passer des paramètres par position ou par nom, Python nous propose 2 façons de passer un nombre de paramètres variable.

6.4.1. Nombre variable de paramètres par position

Les paramètres passés par position sont comparables aux listes qui elle-même conserve des éléments en leur attribuant une position.

Passage d'un nombre de paramètres variable par position

```
>>> def direBonjour(nom,prenom):
...     print('Bonjour '+nom+' '+prenom)
...
>>> l = ['DURAND','Robert']
>>> direBonjour(*l) # ici le *l permet de déstructurer la liste en une
suite de valeur => direBonjour('DURAND','Robert')
Bonjour DURAND Robert
```

6.4.2. Nombre variable de paramètres par mot-clef

Les paramètres passés par position sont comparables aux dictionnaires (liste associative) qui associe une clef conserve des éléments en leur attribuant une position.

Passage d'un nombre de paramètres variable par nom

```
>>> def direBonjour(nom,prenom):
...     print('Bonjour '+nom+' '+prenom)
...
>>> d = {'nom':'DURAND','prenom':'Robert'} # Ceci est un dictionnaire, il
sera abordé plus tard, il permet de faire des associations de paires clef
=> valeur
>>> direBonjour(**d) # ici le **l permet de dèstructurer la liste en une
suite de paire, clef : valeur => direBonjour(nom=DURAND',prenom='Robert')
Bonjour DURAND Robert
```

6.5. Les expressions lambda

Les expression `lambda` permettent de définir des fonctions anonymes : des fonctions qui ne seront utilisées qu'à un seul endroit sans volonté d'être réutilisée.

Pour illustrer les fonctions `lambda`, nous allons utiliser la fonction `map` qui permet de parcourir une liste pour la transformer en une autre liste. Pour ce faire, nous allons passer à la fonction `map` une autre fonction qui va se charger de faire les opérations de transformation sur chacun des éléments de la liste d'origine.

Dans cet exemple nous allons simplement doubler chaque valeur de liste l et obtenir une nouvelle liste

La fonction map

```
```map(function, iterable, [iterable 2, iterable 3, ...])```
```
```

Première solution (sans map et sans lambda)

```
>>> l = [1,2,3,4]
>>> d = []
>>> def mult2(a):
...     return a*2
...
>>> for i in l:
...     d.append(mult2(i))
>>> d
[2, 4, 6, 8]
```

Deuxième solution (avec map et sans lambda)

```
>>> l = [1,2,3,4]
>>> d = []
>>> def mult2(a):
...     return a*2
...
>>> d = map(mult2,l) # la fonction map appel mult2 pour chaque élément de
l et ajoute la valeur retournée dans d
>>> list(d) # On transforme la map en une list
[2, 4, 6, 8]
```

Dans ce dernier exemple, on peut se rendre compte que la fonction mult2 ne sera certainement jamais utilisée ailleurs que dans cet exemple. On peut donc l'anonymiser et la passer en paramètre à map. C'est le rôle des `lambda`

Troisème solution (avec map et avec lambda)

```
>>> l = [1,2,3,4]
>>> d = []
>>> d = map(lambda v: v*2,l) # la fonction map appel mult2 pour chaque
élément de l et ajoute la valeur retournée dans d
>>> list(d) # On transforme la map en une list
[2, 4, 6, 8]
```

6.6. Documenter automatiquement vos fonctions

Les commentaires sont importants dans un script Python, il en est de même pour la documentation. Python a une solution simple pour documenter facilement vos fonctions. les Docstring. [pydocstring] Les Docstrings sont des chaînes de caractères multilignes placées au tout début d'une fonction. Python va ensuite initialiser une propriété de votre fonction (la propriété *doc*) afin de permettre à certains outils ([doxygen]) de générer la documentation de votre script.

Documentation d'une fonction

```
>>> def une_function():
...     """Ceci est de la
...     documentation
...     très utile
...     """
...     pass
...
>>> print(une_function.__doc__)
Ceci est de la
    documentation
    très utile

>>>
```

Chapitre 7. Les structures de données

7.1. Les listes [listes]

Une liste chaînée est une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments de même type, dont la représentation en mémoire de l'ordinateur est une succession de cellules faites d'un contenu et d'un pointeur vers une autre cellule. De façon imagée, l'ensemble des cellules ressemble à une chaîne dont les maillons seraient les cellules.[wlistes]

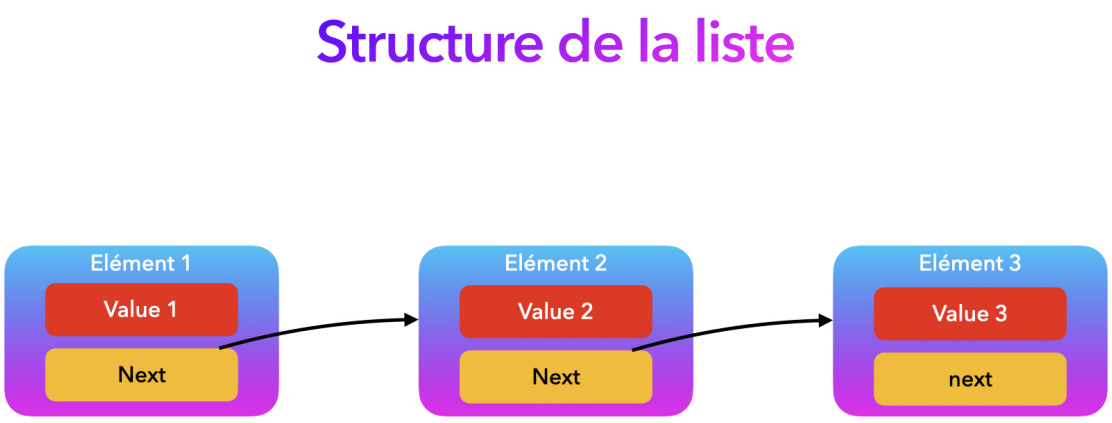


Figure 3. Structure d'une liste

Les listes sont riches de nombreuses fonctionnalités fort bien documenté, voici ici quelques méthodes importantes issues de la documentation.

- **list.append(x)**: Ajoute un élément à la fin de la liste. Équivalent à a[len(a):] = [x].
- **list.extend(iterable)** : Étend la liste en y ajoutant tous les éléments de l'itérable. Équivalent à a[len(a):] = iterable.
- **list.insert(i, x)** : Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc a.insert(0, x) insère l'élément en tête de la liste et a.insert(len(a), x) est équivalent à a.append(x).

- **list.remove(x)** : Supprime de la liste le premier élément dont la valeur est égale à x. Une exception ValueError est levée s'il n'existe aucun élément avec cette valeur.
- **list.pop([i])** : Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, a.pop() enlève et renvoie le dernier élément de la liste (les crochets autour du i dans la signature de la méthode indiquent que ce paramètre est facultatif et non que vous devez placer des crochets dans votre code ! Vous retrouverez cette notation fréquemment dans le Guide de Référence de la Bibliothèque Python).
- **list.clear()** : Supprime tous les éléments de la liste. Équivalent à del a[:].
- **list.index(x[, start[, end]])** Renvoie la position du premier élément de la liste dont la valeur égale x (en commençant à compter les positions à partir de zéro). Une exception ValueError est levée si aucun élément n'est trouvé.

Les arguments optionnels start et end sont interprétés de la même manière que dans la notation des tranches et sont utilisés pour limiter la recherche à une sous-séquence particulière. L'indice renvoyé est calculé relativement au début de la séquence complète et non relativement à start.

- **list.count(x)** Renvoie le nombre d'éléments ayant la valeur x dans la liste.
- **list.sort(*, key=None, reverse=False)** Ordonne les éléments dans la liste (les arguments peuvent personnaliser l'ordonnancement, voir sorted() pour leur explication).
- **list.reverse()** Inverse l'ordre des éléments dans la liste.
- **list.copy()** Renvoie une copie superficielle de la liste. Équivalent à a[:].

7.2. Diverses utilisations des Listes : les piles

Une pile (stack) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (**LIFO** pour **last in**, **first out**), donc, le dernier élément, ajouté à la pile, sera le premier à en sortir.[wstack]

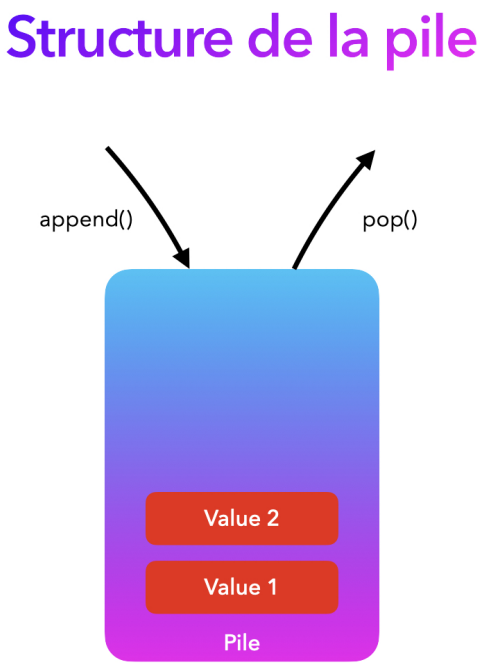


Figure 4. Structure d'une pile

Exemple de l'utilisation d'une pile

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

7.3. Diverses utilisations des Listes : les files

Une file dite aussi file d'attente (queue) est une structure de données basée sur le

principe « premier entré, premier sorti » ou FIFO (**first in**, **first out**) : les premiers éléments ajoutés à la file seront les premiers à en être retirés.[wsqueue]

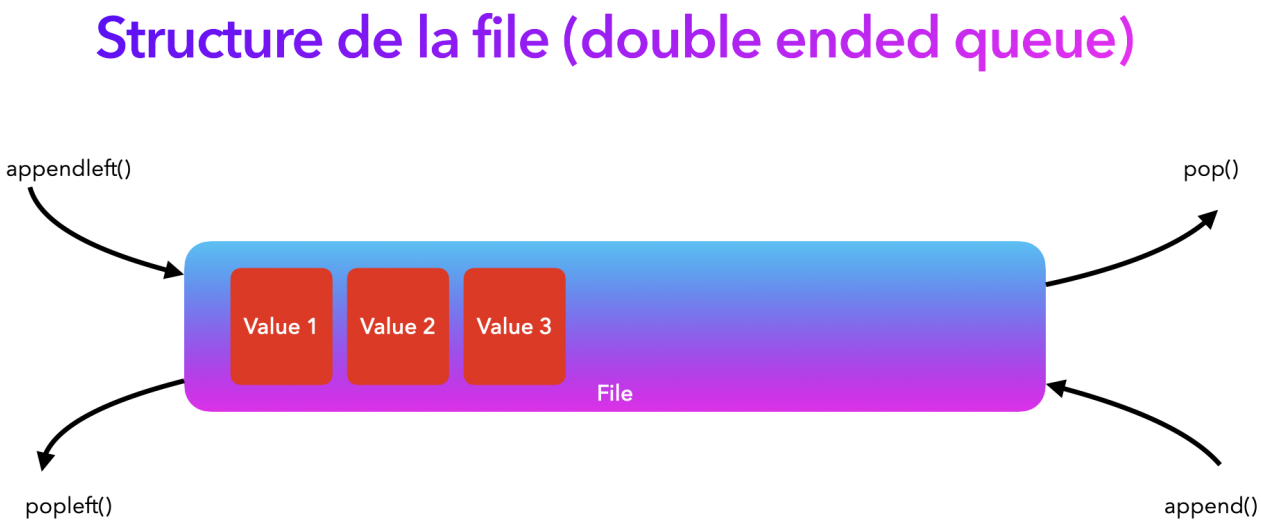


Figure 5. Structure d'une file

Exemple de l'utilisation d'une file

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

7.4. Les compréhensions de liste (ou liste en compréhension)

Python a un moyen simple d'initiliser les listes : **les compréhensions**. L'idée ici est de nous permettre de fournir des valeurs initiales basées sur des traitements sans avoir à utiliser de multiples boucles `for`.

Pour initialiser une `list` **d** à partir de la `list` **l** avec une lambda:

Rappel map et lambda

```
>>> l = [1,2,3,4]
>>> d = []
>>> d = map(lambda v: v*2,l)
>>> list(d)
[2, 4, 6, 8]
```

Pour initialiser une `list` **d** à partir de la `list` **l** avec une compréhension:

Initialisation avec une compréhension de liste

```
>>> l = [1,2,3,4]
>>> d = [x*2 for x in l]
>>> d
[2, 4, 6, 8]
```

7.5. D'autres structures de données : Les `tuples` et les séquences

Il existe également un autre type standard de séquence : le tuple[wstuples]. Les tuples ressemblent beaucoup au liste mais ils ont quelques caractéristiques qui les distinguent. - les tuples sont toujours affichés entre parenthèses - les tuples sont immuables - les tuples peuvent contenir des éléments de types différents.

Exemple d'utilisation des tuples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Les tuples peuvent être imbriqués
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Les tuples sont immuables:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # Mais peuvent contenir des objets mutable
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

7.6. Les ensembles mathématiques avec `set`

Les ensembles représentent la notion d'ensemble mathématique à savoir : une collection non ordonnée sans élément dupliqué. Les ensembles sont pratiques pour toutes les opérations ensemblistes, et aussi pour la suppression de doublons.

Exemple d'utilisation des sets

```
>>> i = {1,2,3,1,4,2,5,6}
>>> j = {1,2,5,6}
>>> i
{1, 2, 3, 4, 5, 6}
>>> 1 in i
True
```

7.6.1. La Différence

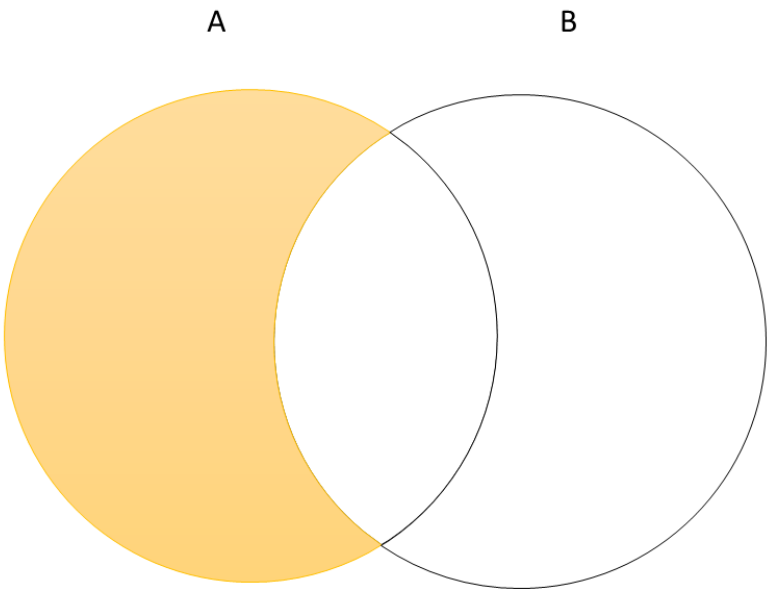


Figure 6. Différence

Exemple d'utilisation des sets : La Différence

```
>>> i-j                 # Différence on renvoie un nouvel ensemble avec
les éléments de i qui ne sont pas dans j
{3, 4}
```

7.6.2. L'Union

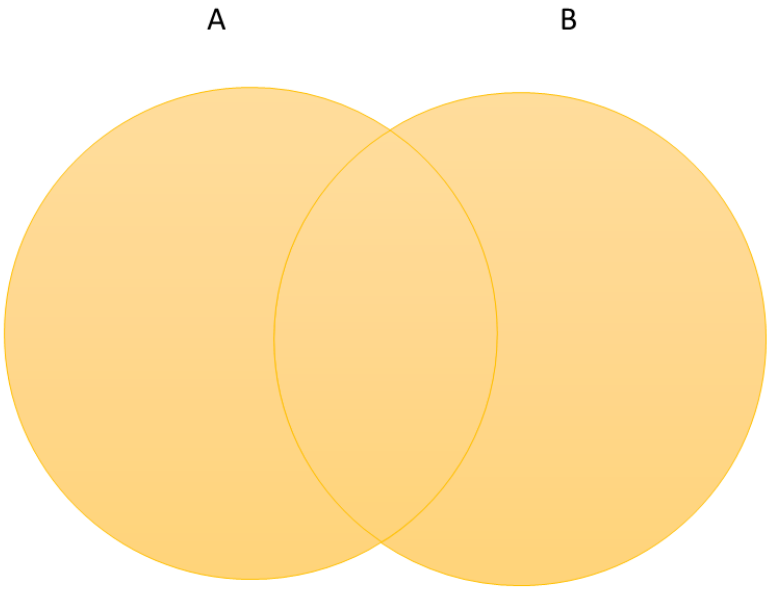


Figure 7. Union

Exemple d'utilisation des sets : L'Union

```
>>> i|j                 # Union de i et de j => i.union(j). Renvoie les
éléments de i et de j
{1, 2, 3, 4, 5, 6}
```

7.6.3. L'Intersection

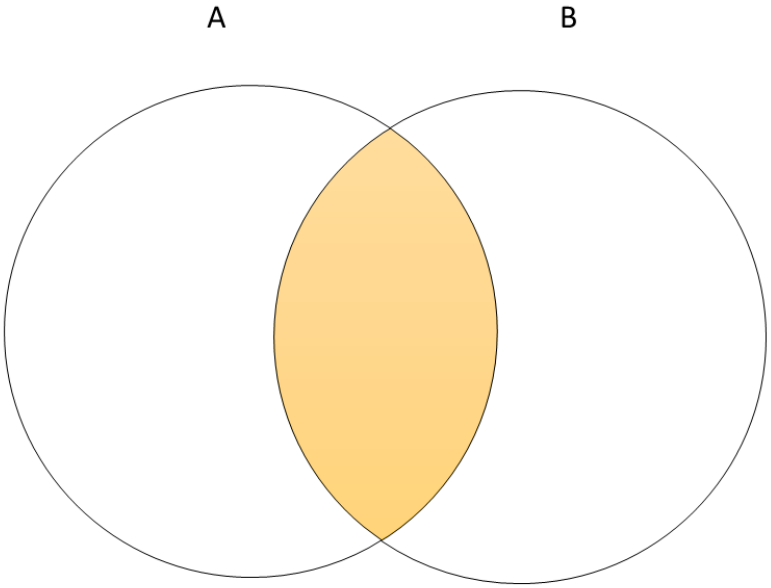


Figure 8. Intersection

Exemple d'utilisation des sets : L'Intersection

```
>>> i & j              # intersection
{1, 2, 5, 6}
```

7.6.4. La Différence Symétrique

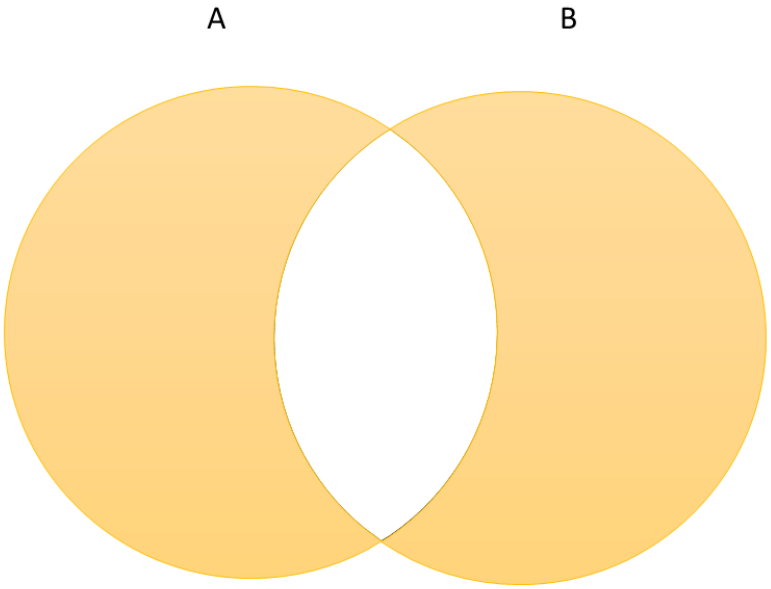


Figure 9. La Différence Symétrique

Exemple d'utilisation des sets : La Différence Symétrique

```
>>> i ^ j                         # renvoie les lettres qui sont dans
i ou j, mais pas les deux
{3, 4}
```

7.7. Une structure associative avec les dictionnaires

Les dictionnaires sont parfois appelés tableaux associatifs dans d'autres langages. À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des clés, qui peuvent être de n'importe quel type immuable : (chaînes de caractères, nombres). Les dictionnaires sont des ensembles de paires clés: valeur, les clés devant être uniques dans le dictionnaire.

Exemple de dictionnaire

```
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

7.7.1. Parcourir un dictionnaire

Exemple de parcours de dictionnaire

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Chapitre 8. Organiser son code en module

L'interpréteur Python ne permet pas de réutiliser son code. Nous allons devoir faire en sorte que nos programmes Python deviennent des scripts afin d'être réutilisés et évoluer.

En Python un fichier contenant du code est appelé un module et les définitions d'un module peuvent être importées dans un autre module. le module porte le nom de son fichier à savoir : le module **fibo** est dans un fichier **fibo.py**. La variable *__name__* renvoie le nom du module. Il existe un module particulier qui ne porte pas le nom de son fichier, c'est le module qui est exécuté. Lui porte un autre nom : *__main__*

8.1. Exécuter des modules : les scripts

La suite de Fibonacci transformée en module

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Utilisation du module

```
>>> import fibo
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__ # Affiche le nom du module
'fibo'
```

8.2. Les différentes façons d'importer un module

importer un module ou ce qu'il contient

```
>>> import fibo                       # importe la totalité du module fibo
>>> fibo.fib(1000)

>>> from fibo import fib, fib2        # à partir du module fibo importe la
fonction fib et fib2
>>> fib(1000)
>>> fib2(1000)

>>> from fibo import *                # à partir du module fibo importe la
totalité de ce qu'il contient
>>> fib(1000)
>>> fib2(1000)

>>> import fibo as lemodule           # ajout d'un alias sur le module
(évite les collisions de nom)
>>> lemodule.fib(1000)

>>>from fibo import fib as mafonction # ajout d'un alias sur une fonction
(évite les collisions de nom)
```

8.3. Exécuter des scripts

Pour être certain que notre module est exécuté (et pas importé), nous allons devoir nous assurer que le script porte le nom *`__main__`*.

Exécution d'un script Python shell

```
python fibo.py
```

Exécution d'un script Python code

```
import fibo

if __name__ == "__main__": ## ce code ne sera exécuté que si le module est
exécuté.
    fibo.fib(1000)
```

8.4. plus loin avec les modules : les Packages

Il est possible d'organiser efficacement une application complexe. Pour cela, nous n'allons pas nous contenter de fichier mais aussi de répertoire. Avec Python, un répertoire qui contient des modules et un fichier *`__init__.py`* est un package.

Exemple d'architecture

```
pack01/
      __init__.py
      pack02/
              __init__.py
              mod01.py
              mod02.py
      pack03/
              __init__.py
              mod01.py
      pack04/
              __init__.py
              mod03.py
```

Exemple d'import

```
import pack01.pack02.mod01

from pack01.pack02 import mod01

from pack01.pack02.mod01 import function01
```

Chapitre 9. Les entrées et sorties

L'affichage de données est essentiel en Python, c'est la raison pour laquelle il est fondamental de maîtriser les différentes techniques de formatage à l'affichage. De la même façon nous allons devoir lire et écrire à partir d'un fichier dans différents format (json, ...)

9.1. Formater des chaînes de caractère

9.1.1. Expression formatée

Pour utiliser une expression formatée, on préfixe la chaîne de caractère avec un `f` ou un `F` et on insère des variables entre `{}`

Exemple du formatage d'une chaîne de caractère avec une expression formatée

```
>>> quand = 2021
>>> quoi = 'une Formation Python'
>>> f'Ceci est {quoi} en {quand}'
'Ceci est une Formation Python en 2021'
```

Il est aussi possible d'utiliser des expressions à l'intérieur des chaînes formatées. [specfor]

Utilisation d'une expression dans une chaîne formatée

```
>>> pi = 3.14159265358979323846264433
>>> print(f"valeur de pi {pi:.3f}")
valeur de pi 3.142
>>>
>>>
>>> for i in range(5):
...     print(f"valeur de {str(i):10} => {i:10}")
...
valeur de 0          =>          0
valeur de 1          =>          1
valeur de 2          =>          2
valeur de 3          =>          3
valeur de 4          =>          4
>>>
>>> for i in range(5):
...     print(f"valeur de {i:10} => {i:10}")
...
valeur de          0 =>          0
valeur de          1 =>          1
valeur de          2 =>          2
valeur de          3 =>          3
valeur de          4 =>          4
```

9.1.2. La méthode `format()`

Il est aussi possible d'utiliser la méthode `str.format()`

```
>>> quand = 2021
>>> quoi = 'une Formation Python'
>>> 'Ceci est {} en {}'.format(quoi, quand)
'Ceci est une Formation Python en 2021'
>>> 'Ceci est {0} en {1}'.format(quoi, quand)
'Ceci est une Formation Python en 2021'
>>> d = {'clef1':'valeur1','clef2':'valeur2','clef3':'valeur3'}
>>> print("clef1 : {0[clef1]}".format(d)) # Paramètre par position
>>> print("clef1 : {clef1}".format(**d)) # Paramètre par nom
```

Dans tous les cas il suffit de convertir vos valeurs en chaîne de caractère pour les afficher en utilisant la fonction `str()`. Le rôle de la fonction `str()` est de convertir une donnée en quelque chose de compréhensible humainement. Pour des cas plus particuliers, il est aussi possible de convertir une valeur en une autre valeur interprétable pour Python avec `repr()`

Exemple str() et repr()

```
>>> s = """
... ceci
... est
... une
... chaîne
... """
>>> str(s)
'\nceci\nest\nune\nchaîne\n'
>>> repr(s)
"'\\nceci\\nest\\nune\\nchaîne\\n'"
```

9.2. Travailler avec des fichiers

On utilise la fonction `open()` pour récupérer un objet fichier. Le fonction `open()` attend 2 paramètres : - le nom du fichier - le mode (lecture, écriture, ...)

Ouverture d'un fichier

```
>>> f1 = open('lefichier.txt', 'w') # ouverture en écriture
>>> f2 = open('lefichier.txt','r') # ouverture en lecture
>>> f3 = open('lefichier.txt') # ouverture en lecture (mode par défaut)
```

Vous devez toujours fermer un fichier après l'avoir utilisé.

Fermeture d'un fichier

```
>>> f1.close()
>>> f2.close()
>>> f3.close()
```

Afin d'éviter un oubli ou ne pas pouvoir fermer le fichier si une erreur s'est produite, il est préférable de laisser à Python la responsabilité de la libération de la ressource. On utilise la syntaxe `with`

Ouverture d'un fichier avec with

```
>>> with open('lefichier.txt', 'w') as f: # ouverture en écriture
...
...
>>> with open('lefichier.txt','r') as f2: # ouverture en lecture
...
...
>>> with open('lefichier.txt') as f3: # ouverture en lecture (mode par
défaut)
...
...
```

9.2.1. Travailler avec les fichiers

f.readline() :lit une seule ligne du fichier f.read() : lit la totalité du fichier f.readlines() : lit la totalité du fichier dans une liste f.write("Ceci est une chaîne\n") : écrit une ligne dans un fichier

Contenu du fichier lefichier.txt

```
Ligne 1
Ligne 2
Ligne 3
Ligne 4
Ligne 5
```

Contenu du fichier lefichier.txt

```
>>> with open('lefichier.txt') as f:
>>> ... l = f.readline()
>>> ... print(l)
>>> Ligne 1

with open('lefichier.txt') as f:
    l = f.read()
    print(l)

>>> Ligne 1
>>> Ligne 2
>>> Ligne 3
>>> Ligne 4
>>> Ligne 5

>>> with open('lefichier.txt') as f:
...     l = f.readlines()
...    print(l)

>>> ['Ligne 1\n', 'Ligne 2\n', 'Ligne 3\n', 'Ligne 4\n', 'Ligne 5']
```

9.3. Notion de sérialisation

La sérialisation et une opération visant à rendre une donnée en mémoire transportable dans un flux d'entrée/sortie. L'objectif est de mettre en "série" (comme dans d'un montage électrique) les octets constituant cette donnée. Il existe beaucoup de moyens permettant cette opération : le XML, le JSON,...

9.3.1. Le format JSON

Le JSON est un format chaîne de caractère permettant de représenter les données

Python. Ce format est comparable aux dictionnaires.

Ecrire un dictionnaire dans un fichier au format JSON

```
import json

d = {'clef1':'valeur1','clef2':'valeur2','clef3':'valeur3'}

with open('lefichier.json','w') as f:
    f.write(json.dumps(d))
```

Contenu du fichier lefichier.json

```
{"clef1": "valeur1", "clef2": "valeur2", "clef3": "valeur3"}
```

Lecture d'un fichier au format JSON

```
import json

with open('lefichier.json','r') as f:
    d = json.load(f)
    print(d)

>>> {'clef1': 'valeur1', 'clef2': 'valeur2', 'clef3': 'valeur3'}
```

9.3.2. Le module pickle

Il existe un autre moyen de sérialiser des données Python, le format `pickle` proposé dans le module pickle. Ce module est propre à Python et permet de sérialiser des éléments directement en binaire. [pickle]

Chapitre 10. La programmation orientée objet (POO)

La programmation orientée objet (POO) consiste en l'interaction d'éléments logiciels appelées **objets**. Un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne (des propriétés) et un comportement (des méthodes), et il sait interagir avec ses pairs. La programmation orientée est donc un moyen de représenter des objets et leurs relations.[wpoo]

Pour modéliser une application s'appuyant sur la programmation orientée objet, on utilise UML (Unified Modeling Language)

10.1. La définition d'un objet (état, comportement, identité).

Un objet est un conteneur ayant des informations et des mécanismes concernant un sujet, manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. Il est créé à partir d'un modèle appelé **classe** ou **prototype** (Javascript), dont il **hérite** les comportements et les caractéristiques.[wobj]

10.2. La notion de classe, d'attributs et de méthodes.

Une classe regroupe des membres, méthodes et propriétés (attributs) communes à un ensemble d'objets.

La classe déclare, d'une part, des attributs représentant l'état des objets et, d'autre part, des méthodes représentant leur comportement.

Une classe représente donc une catégorie d'objets. Elle apparaît aussi comme un moule ou une usine à partir de laquelle il est possible de créer des objets ; c'est en quelque sorte une « boîte à outils » qui permet de fabriquer un objet. On parle alors d'un objet en tant qu'instance d'une classe (création d'un objet ayant les propriétés de la classe).

Il est possible de restreindre l'ensemble d'objets représenté par une classe A grâce à un mécanisme d'héritage. Dans ce cas, on crée une nouvelle classe B liée à la classe A et qui ajoute de nouvelles propriétés. [wcls]

10.3. L'encapsulation des données

EnL'encapsulation désigne le principe de regrouper des données brutes avec un ensemble de routines permettant de les lire ou de les manipuler. Ce principe est souvent accompagné du masquage de ces données brutes afin de s'assurer que l'utilisateur ne contourne pas l'interface qui lui est destinée. L'ensemble se considère alors comme une boîte noire ayant un comportement et des propriétés spécifiés.

L'encapsulation est un pilier de la programmation orientée objet, où chaque classe définit des méthodes ou des propriétés pour interagir avec les données membres. [wencap]

10.4. La communication entre les objets.

La communication entre objets se fait par le biais de méthodes. Un objet Définit des méthodes **publiques** (visible par tous) et l'ensemble de ces méthodes est appelé une **interface**. [winter] L'appel de la méthode d'un objet revient à lui envoyer un message.

10.5. L'héritage, transmission des caractéristiques d'une classe.

L'héritage est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une classe mère . La classe héritant de ces caractéristiques devient la classe fille. [wherit]

10.6. La notion de Polymorphisme.

En informatique et en théorie des types, le polymorphisme, est le concept consistant à fournir une interface unique à des entités pouvant avoir différents types. Par exemple, un carré, un rectangle et un cercle sont de types différents mais ils proposent tous une méthode de calcul de surface : **la même interface**. [wpoly]

10.7. Les interfaces.

Une interface est un ensemble de signatures de méthodes publiques d'un objet. Il s'agit donc d'un ensemble de méthodes accessibles depuis l'extérieur d'une classe, par lesquelles on peut modifier un objet, ou plus généralement communiquer avec lui. [winter]

10.8. Présentation d'UML.

Unified Modeling Language ou Langage de Modélisation Unifié est un langage de modélisation graphique à base de pictogrammes conçus comme une méthode normalisée de visualisation dans les domaines du développement logiciel et en conception orientée objet.

L'UML est une synthèse de langages de modélisation objet antérieurs : Booch, OMT, OOSE. Principalement issu des travaux de Grady Booch, James Rumbaugh et Ivar Jacobson, UML est à présent un standard adopté par l'Object Management Group (OMG). UML 1.0 a été normalisé en janvier 1997; UML 2.0 a été adopté par l'OMG en juillet 20051. La dernière version de la spécification validée par l'OMG est UML 2.5.1 (2017)2.

UML définit différents diagrammes permettant une représentation sous différents angles du même projet. [wuml]

10.9. Quelques diagrammes UML

10.9.1. Le diagramme de classes

Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que leurs relations. Ce diagramme fait partie de la partie statique d'UML, ne s'intéressant pas aux aspects temporels et dynamiques.

Une classe décrit les responsabilités, le comportement et le type d'un ensemble d'objets. Les éléments de cet ensemble sont les instances de la classe. [wuml_cls]

10.9.2. Diagramme d'activité

Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multi-threads ou multi-processus). Le diagramme d'activité est également utilisé pour décrire un flux de travail (workflow). [wuml_act]

10.9.3. Diagramme états-transitions

Un diagramme états-transitions est un schéma utilisé en génie logiciel pour

représenter des automates déterministes. Il fait partie du modèle UML et s'inspire principalement du formalisme des statecharts et rappelle les grafcets des automates. S'ils ne permettent pas de comprendre globalement le fonctionnement du système, ils sont directement transposables en algorithme. En effet, contrairement au diagramme d'activité qui aborde le système d'un point de vue global, le diagramme états-transitions cible un objet unique du système. Tous les automates d'un système s'exécutent parallèlement et peuvent donc changer d'état de façon indépendante.

[wuml_et]

10.10. Notion de modèle de conception (Design Pattern).

Un patron de conception est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un **problème de conception** d'un logiciel. Il décrit une **solution standard**, utilisable dans la conception de différents logiciels.

Les patrons de conception décrivent des procédés de conception généraux et permettent en conséquence de capitaliser l'expérience appliquée à la conception de logiciel. Ils ont une influence sur l'architecture logicielle d'un système informatique.

Les patrons de conception ont été formellement reconnus en 1994 à la suite de la parution du livre Design Patterns: Elements of Reusable Software, co-écrit par quatre auteurs : Gamma, Helm, Johnson et Vlissides (Gang of Four - GoF ; en français « la bande des quatre »). Il définit 23 design patterns, en voici quelques-uns : - Singleton: vise à assurer qu'il n'y a toujours qu'une seule instance d'une classe - Iterator: permet d'accéder séquentiellement aux éléments d'un ensemble sans connaître les détails techniques du fonctionnement de l'ensemble - ...

[wuml_pat]

Chapitre 11. Programmation Objet en Python

Python mélange beaucoup de concept récupérer dans différents langages (C++, Smalltalk, Modula-3)

Les classes fournissent toutes les fonctionnalités standards de la programmation orientée objet : l'héritage de classes autorise les héritages multiples, une classe dérivée peut surcharger les méthodes de sa ou ses classes de base et une méthode peut appeler la méthode d'une classe de base qui possède le même nom.

Les objets peuvent contenir n'importe quel nombre ou type de données. De la même manière que les modules, les classes participent à la nature dynamique de Python : elles sont créées pendant l'exécution et peuvent être modifiées après leur création.

Dans la terminologie C++, les membres des classes (y compris les données) sont publics (sauf exception, voir Variables privées) et toutes les fonctions membres sont virtuelles.

Comme avec Modula-3, il n'y a aucune façon d'accéder aux membres d'un objet à partir de ses méthodes : **une méthode est déclarée avec un premier argument explicite représentant l'objet** et cet argument est transmis de manière implicite lors de l'appel.

Comme avec Smalltalk, les classes elles-mêmes sont des objets. Il existe ainsi une sémantique pour les importer et les renommer. Au contraire de C++ et Modula-3, les types natifs peuvent être utilisés comme classes de base pour être étendus par l'utilisateur.

Enfin, comme en C++, la plupart des opérateurs natifs avec une syntaxe spéciale (opérateurs arithmétiques, indiçage, etc.) peuvent être redéfinis pour les instances de classes.

11.1. L'écriture de classes et leur instanciation.

Définition d'une classe et création d'un objet.

```
# Création d'une classe
class Rectangle:
...
...
...

r = Rectangle() # Création d'un objet r à partir de la classe Rectangle
```

11.2. Les constructeurs et les destructeurs.

Le contructeur et la méthode permettant d'initiliaser les propriétés d'un objet en python c'est la méthode `__init__(self)` Le destructeur et la méthode appelée lorsque l'objet est détruit

```
class Rectangle:
    def __init__(self,longueur=0,largeur=0):
        self._longueur = longueur
        self._largeur = largeur

    def __del__(self):
        print('le destructeur')
```

11.3. La protection d'accès des attributs et des méthodes.

Les membres "privés", qui ne peuvent être accédés que depuis l'intérieur d'un objet, n'existent pas en Python. Nous utilisons une convention : un nom préfixé par un tiret bas (comme _spam) doit être considéré comme une partie non publique de l'API (qu'il s'agisse d'une fonction, d'une méthode ou d'un attribut 'données'). [pypriv]

Déclaration de 2 membres privés

```
class Rectangle:
    def __init__(self,longueur=0,largeur=0):
        self._longueur = longueur
        self._largeur = largeur

    def __del__(self):
        print('le destructeur')
```

11.4. La nécessité du paramètre self.

Self est une convention. Il est le premier paramètre passé à l'ensemble des méthodes d'un objet. On peut le comparer au `this` d'autres langage orienté objet

[pyself]

11.5. L'héritage simple, l'héritage multiple, le polymorphisme.

Héritage

```
from Rectangle import Rectangle

class Carre(Rectangle):

    def __init__(self,cote=0):
        super().__init__(cote,cote)
        print("def __init__(self,cote=0):")
        self._cote = cote

    def __str__(self):
        return f"Carre cote : {self._cote}"
```

11.6. L'implémentation des interfaces.

Une interface ne contient aucune implémentation de méthode, uniquement les déclarations.

Ici nous souhaitons imposer l'implémentation d'une méthode à la classe Rectangle et provoque une erreur si cette méthode n'est pas implémentée.

Interface

```
class ICalcMath:
        def calc_surface(self):
            raise NotImplementedError("calc_surface NotImplemented !")

class Rectangle(ICalcMath):

    _cpt = 0

    def __init__(self,longueur=0,largeur=0):
        print("def __init__(self,longueur=0,largeur=0)")
        self._longueur = longueur
        self._largeur = largeur
        RectangleP._cpt+=1

    def __str__(self):
        return f"Rectangle : longueur : {self._longueur}, largeur : {self
._largeur}"

    def calc_surface(self): ## Méthode surcharge celle de l'interface
ICalcMaths
        return self._largeur*self._longueur
```

Chapitre 12. Les erreurs et les Exceptions

Il existe plusieurs type d'erreurs en Python, celles liées au langage (syntaxe,...) et celles liées à l'exécution de votre code.

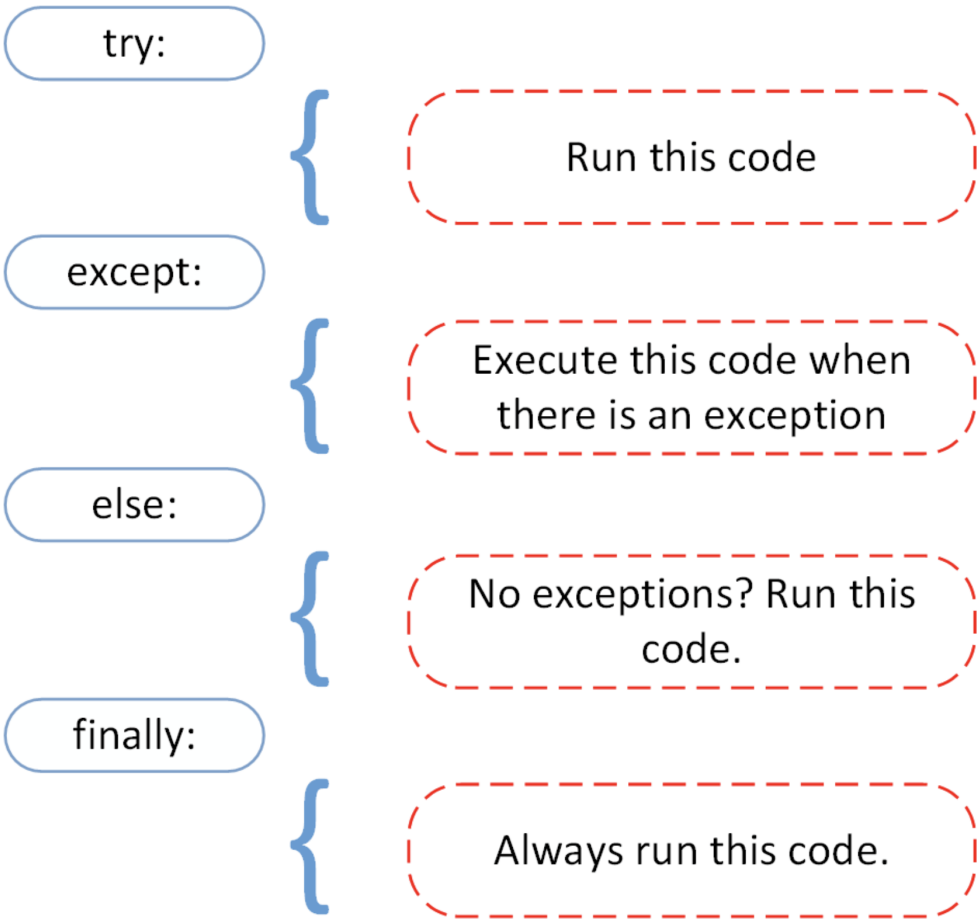
Erreur de syntaxe

```
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Erreur générer par l'exécution du code

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

12.1. Gestion des Exceptions



Gestion des Exceptions

```
class DivBy12Exception(Exception):

    def __init__(self):
        super().__init__("Division par 12")


    try:
        c = calc.div(1,12)
        print(c)
    except ZeroDivisionError as e:
        print("erreur",e)
    except DivBy12Exception as e:
        print("DivBy12Exception erreur",e)
        raise Exception() #Déclenchement d'une exception (la faire
remonter)
    finally:
        print("après erreur
```

[pyexc01] - [pyexc02]

Chapitre 13. Utilisation StdLib

Python est naturellement riche de beacoup de bibliothèques. Il est "livré avec des piles" [pypiles]

13.1. Les arguments passés sur la ligne de commande.

Il est possible de simplement passé par le module `sys` mais il existe une autre module : `argparse` beaucoup plus sophistiqué.

Paramètres passés à ligne de commande avec sys

```
import sys
print(sys.argv)

python monscript.py one two three

['monscript.py', 'one', 'two', 'three']
```

Paramètres passés à ligne de commande avec argparse

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
    description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()

print(args)
```

13.2. L'utilisation du moteur d'expressions régulières Python avec le module "re"

Les expressions régulières permettent d'extraire d'un contenu basé sur des chaînes de caractères. Pour ce faire nous nous devons utiliser un autre "langage" : les expressions régulières. Les expressions régulières sontdans le module `re`.

Exemple simple d'utilisation des regexp

```
import re

#Vérifie si la chaîne démarre par "The" eet se finit par "Spain":

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)

if x:
  print("Oui")
else:
  print("Non")
```

[pyre]

13.3. La manipulation du système de fichiers

Exemple simple d'utilisation de os et shutil

```
import os
import shutil
import glob

os.getcwd()      # Return the current working directory
os.chdir('/server/accesslogs')   # Change current working directory
os.system('mkdir today')   # Run the command mkdir in the system shell

shutil.copyfile('data.db', 'archive.db')
shutil.move('/build/executables', 'installdir')

l = glob.glob('*.py')
print(l)
['primes.py', 'random.py', 'quote.py']
```

[pyos]

13.4. Création d'environnements virtuels

Pour éviter les conflits de versions de Python ou de bibliothèque, il est important d'isoler ses projets.

Création d'un environnement virtuel python3

```
mkdir monprojetpython3
cd monprojetpython3
python3 -m venv .venv
source .venv/bin/activate # Init des variables d'environnement
...
deactivate
```

13.5. Installation d'une bibliothèque Python.

Même si Python est très généreux en termes de fonctionnalité, il est parfois nécessaire de récupérer des bibliothèques externes. Les bibliothèques sont répertoriées ici : pypi.org

Création d'un environnement virtuel python3 et récupération d'un bibliothèque

```
mkdir monprojetpython3
cd monprojetpython3
python3 -m venv .venv
source .venv/bin/activate   # Init des variables d'environnement
pip install requests        # Installation d'une bibliothèque
...
deactivate
```

Liste des dépendances

```
pip list
```

Sauvegarde des dépendances dans un fichier requirements

```
pip freeze > requirements.txt
```

Restauration des dépendances

```
source .venv/bin/activate
pip install -r requirements.txt
```

13.6. Les accès aux bases de données relationnelles, le fonctionnement de la DB API.

La DB API est une API visant à uniformiser les accès aux bases de données. .Exemple avec SQLite

```
import sqlite3
conn = sqlite3.connect('ma_base.db')

cursor = conn.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS users(
     id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE,
     name TEXT,
     age INTERGER
)
""")
conn.commit()

cursor.execute("INSERT INTO users(name, age) VALUES(?, ?)", ("olivier",
30))
conn.commit()

cursor.execute("""SELECT name, age FROM users""")
user1 = cursor.fetchone()
print(user1)

db.close()
```

[pydbapi_sqlite] - [pydbapi]

Chapitre 14. Outils QA

14.1. Les outils d'analyse statique de code (Pylint, Pychecker).

Les linters sont des outils qui ont pour objectif de vérifier votre code afin de faire respecter des specs, des normes, et vous proposent des suggestions pour l'améliorer.

Exemple de script

```
#!/usr/bin/env python
import string

def hello(name):
    print("Hello "+name)

hello('Fred')
```

Suggestion de PyLint

```
pylint main.py
************* Module main
main.py:9:0: C0304: Final newline missing (missing-final-newline)
main.py:1:0: C0114: Missing module docstring (missing-module-docstring)
main.py:4:0: C0116: Missing function or method docstring (missing-
function-docstring)
main.py:2:0: W0611: Unused import string (unused-import)

-------------------------------------------------------------------
Your code has been rated at 0.00/10 (previous run: 10.00/10, -10.00)
```

14.2. Le développement piloté par les tests.

Python est livré avec des outils permettant les tests unitaires : le module `unittest`
En pratique, les tests s'écrivent avant le code.

Un exemple simple

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Chapitre 15. Création IHM TkInter

Python offre nativement la possibilité de réaliser des interfaces graphiques avec TkInter (basé sur Tk)

Hellow World Tkinter

```
class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

Chapitre 16. Conclusion

Python est un langage facile à comprendre et facile à lire. Il est présent dans de nombreux projets : machine learning, Web, script d'admin système ou réseaux.

Vous devriez maintenant pouvoir commencer ou faire évoluer de nombreux projets basés sur Python.

Réferences

- [walg] Wikipedia Algorithme
- [wprog] Wikipedia Programme
- [wpparad] Wikipedia Paradigme de programmation
- [wpscod] Wikipedia Pseudo-code
- [pscod] Pseudo-code
- [wpy] Wikipedia Python
- [winterp] Wikipedia interpréteur
- [wiversion] Historique des versions
- [pynum] Python, les nombres
- [pystr] Python, les chaînes
- [pylst] Python, une première approche des listes
- [pyfibo] Exemple de la suite de Fibonacci
- [pyloop] Les instructions break, continue et les clauses else au sein des boucles
- [pydef] Valeur par défaut des arguments
- [pydocstring] PEP 257 — Docstring Conventions
- [doxygen] Doxygen
- [listes] Les listes
- [wlistes] Les listes Wikipedia
- [wstack] Les piles Wikipedia
- [wsqueue] Les files Wikipedia
- [wstuples] Les tuples, N-uplet Wikipedia
- [specfor] Mini-langage de spécification de format
- [pickle] Le module Pickle
- [wpoo] La programmation orienté objet
- [wobj] Les objets
- [wcls] Les classes
- [wencap] Encapsulation

- [winter] Les Interfaces
- [wherit] L'héritage
- [wpoly] Polymorphisme
- [wuml] UML
- [wuml_cls] UML, Diagramme de classes
- [wuml_act] UML, Diagramme d'activité
- [wuml_et] UML, Diagramme états-transitions
- [wuml_pat] Design Patterns
- [pycla] Les classes en Python
- [pypriv] Les membres privés
- [pyself] Self
- [pyexc01] Error et Exception 01
- [pyexc02] Error et Exception 02
- [pypiles] Piles fournies
- [pyre] Regexp
- [pyos] Operating System Interface
- [pydbapi] DB API
- [pydbapi_sqlite] DB API SQLite