Programmation en Python
Licence 1

Gaële Simon - Bruno Mermet

Janvier 2023

Table des matières

I. Introduction....4
1. Présentation générale....4
2. Exécuter du code python....4
II. Quelques caractéristiques....4
1. Les variables....4
2. Syntaxe de base....6
III. Syntaxe générale....7
A. Structures de contrôle....7
1. test....7
2. boucles....8
B. Types de données en Python....9
C. Définition de fonctions....10
1. Syntaxe de base....10
2. Valeurs par défaut des arguments....11
3. Arguments nommés....11
4. Utilisation de listes ou de tables associatives en guise d'arguments....12
5. Méthodes à nombres variable de paramètres....12
6. Lambda-fonctions....13
7. Typage explicite....13
IV. Manipulation de quelques types importants....13
A. Les Séquences....13
1. Accès à un élément....13
2. Accès à une tranche d'élément (renvoie une copie)....14
3. Concaténation....15
4. Comparaison....15
5. Longueur d'une séquence....15
6. Modification d'un séquence mutable (donc ni tuple, ni chaînes de caractères)....15
a. Les bases....15
b. Et encore….....17
B. Définition de listes en compréhension....17
C. Traitement de listes....18
1. Map....18
2. Filter....18
3. Réduction....18
D. Les ensembles....19
1. appartenance....19
2. Union....19
3. Intersection....19
4. Différence....20
5. Union disjointe....20
E. Tables associatives (dict)....20
1. Utilisation de base....20
2. Autres utilisations....21
a. Création à partir d'une liste de couples....21
b. Création à partir de paramètres nommés....21
c. Création en compréhension....21
d. Parcours d'un dictionnaire....21
V. Entrées-Sorties....21

A. Affichage à l’écran....21
1. La fonction print....21
2. Conversion d’une donnée en chaîne de caractères....22
3. Chaînes de caractères formatées....23
4. Utilisation de la méthode format()....23
B. Lecture au clavier....24
C. Lecture/Écriture dans des fichiers....24
1. Ouverture du fichier....24
2. Lecture depuis un fichier....25
3. Écriture dans un fichier....25
4. Fermeture du fichier....25
VI. Modules....25
VII. Exceptions....26
VIII. Classes et objets....26
A. Introduction....26
B. Variables de classe....27
C. Constructeur et variables d’instance....27
D. Méthodes d’instance....28
E. Affichage automatique sous forme de chaîne de caractères....29
F. Héritage....29
IX. Structure itérable, itérateur et générateur....30
A. Intérêt des notions d’itérable et d’itérateur....30
B. Exemple de définition d’une structure itérable et d’un itérateur....30
C. Générateur....31

I. Introduction

1. Présentation générale

Python est un langage à objets interprété (en fait compilé en *Just In Time*). Idéalement, son utilisation devrait concerner :

- le développement de scripts Système
- le prototypage

Cependant, les utilisations effectives de Python sortent largement de ce cadre. On trouve notamment :

- l'enseignement de l'informatique
- le calcul scientifique (car nombreuses bibliothèques)
- le traitement de données (car nombreuses bibliothèques)

Mais il faut savoir que:

- ce langage permet d'écrire du code très sale si l'on ne respecte pas certaines règles
- l'exécution est relativement lente : environ 20 fois plus lent que Java sur du calcul

Référence : https://docs.python.org/fr/

2. Exécuter du code python

Plusieurs solutions :

Lancer l'interpréteur python (en tapant « python ») puis taper son code

Passer à l'interpréteur python le nom du script à lancer « python hello.py » par exemple

Mettre un « # !/chemin/vers/python » au début de son fichier, faire un chmod 755 sur le fichier puis exécuter directement le fichier.

Remarques sur l'interpréteur :

une ligne réduite à une expression implique l'évaluation de l'expression et l'affichage de sa valeur

la valeur d'une telle expression est implicitement stockée dans la variable '_'.

II. Quelques caractéristiques

1. Les variables

Dans Python, les variables ne sont ni déclarées, ni typées. Leur nom doit commencer par une lettre ou un tiret de soulignement, et peut contenir lettres, chiffres et tirets de soulignement à volonté.

Les fonctions sont des objets comme les autres. Le nom d'une fonction est donc en fait une variable contenant la fonction.

Les variables sont stockées dans des *espaces de noms* (tables associatives nom/valeur) hiérarchiques.

La recherche de la valeur d'une variable est effectuée en recherchant dans les espaces de noms en remontant dans la hiérarchie. L'écriture dans une variable a toujours lieu dans une variable de l'espace de noms situé au bas de la hiérarchie.

L'espace de noms de plus haut niveau est appelé « global ». Cet espace de noms a un fils intitulé « builtins », qui contient toutes les fonctions prédéfinies. Ensuite, chaque fichier python « monFic.py » définit un module « monFic » ayant son propre espace de noms (fils de l'espace de noms « builtin »). Lorqu'on définit une fonction dans ce fichier, un nouvel espace de nom est associé à cette fonction (fils de l'espace de nom associé au module dans lequel est déclaré la fonction). De même, si on déclare une fonction à l'intérieur d'une autre fonction, cette fonction aura un espace de noms qui sera fils de l'espace de noms de la fonction englobante.

Ainsi, le code suivant :

```
nom = "titou"

print("1 " + nom)

def fonction1():
    print("2 " + nom)

def fonction2():
    nom = "polka"
    print("3 " + nom)

fonction1()
fonction2()

print("4 " + nom)
```

Produira l'affichage suivant :

```
1 titou
2 titou
3 polka
4 titou
```

À noter que pour éviter du code trompeur, la fonction suivante générera une erreur à l'exécution car le premier « print » dans la fonction pourrait faire référence à la variable « nom » du module, tandis que le deuxième print ferait référence à la variable « nom » de la fonction :

```
nom = "titou"

print("1 " + nom)


def fonction3():
    print(nom)
    nom = "polka"
    print(nom)

fonction3()

print("4 " + nom)
```

Pour pouvoir modifier la valeur d'une variable d'un espace de noms de niveau supérieur, il faut d'abord définir un « lien » vers cette variable au moyen d'un des 2 mots-clés « `nonlocal` » ou « `global` ».

On modifie maintenant le premier exemple ci-dessus ainsi :

```
nom = "titou"

print("1 " + nom)

def fonction1():
    print("2 " + nom)

def fonction2():
    global nom
    nom = "polka"
    print("3 " + nom)

fonction1()
fonction2()

print("4 " + nom)
```

L'affichage produit est alors le suivant :

```
1 Titou
2 Titou
3 Polka
4 Polka
```

En effet, suite à la liaison « global nom », la variable « nom » modifiée par la fonction fonction2 est celle du module, et non plus celle de la fonction.

2. Syntaxe de base

Les commentaires sont des fins de ligne commençant par le caractère « # »

Dans Python, il n'y a pas de séparateur d'instruction (pas de « ; »). Par ailleurs, il n'y a pas d'accolade pour indiquer l'imbrication de blocs ; c'est l'indentation qui sert à matérialiser l'imbrication de blocs. Le retour au niveau d'indentation précédent marque la fin d'un bloc. À noter : le nombre d'espaces/tabulations spécifiant un niveau d'indentation est libre ; c'est le fait d'avoir plus d'espaces que le niveau précédent qui indique un nouveau bloc imbriqué. Le retour au niveau d'indentation précédent marque la fin d'un bloc.

Voici un exemple de code Python affichant un damier :

```
for i in [1, 2, 3, 4, 5, 6, 7, 8]:
    print(17 * "-") # permet d'afficher 17 tirets
    for j in [1, 2, 3, 4, 5, 6, 7, 8]:
        # le end='' est là pour éviter un retour à la ligne
        print("|", end='')
```

```
        if (i+j)%2 == 0:
            print(' ', end='')
        else:
            print('■', end='')
    print("|")
print(17 * "-")
```

Dans ce code, on retrouve la structure de blocs suivante :

```
for i in [1, 2, 3, 4, 5, 6, 7, 8]:
    print(17 * "-")
    for j in [1, 2, 3, 4, 5, 6, 7, 8]:
        print("|", end='')
        if (i+j)%2 == 0:
            print(' ', end='')
        else:
            print('■', end='')
    print("|")
print(17 * "-")
```

III. Syntaxe générale

A. Structures de contrôle

1. test

```
if condition :
    bloc d'instructions
```

```
if condition :
    bloc d'instructions
else :
    bloc d'instructions
```

```
if condition :
    bloc d'instructions
elif condition2 :
    bloc d'instructions
elif condition3 : # autant de elif qu'on veut
    bloc d'instructions
…
else :            # le else est facultatif
    bloc d'instructions
```

2. boucles

On peut utiliser 2 types de boucles

```
while condition :
    bloc

for variable in sequence :
    bloc
```

La syntaxe du while est la même qu'en java (sans les accolades).

La syntaxe du for est spécifique à python. Une *séquence* est une donnée itérable (avec une méthode `__iter()__`, renvoyant un itérateur à chaque appel – voir plus bas).

À noter : la fonction `range()` est pratique pour itérer sur des entiers, puisqu'elle renvoie une séquence d'entiers (et non une liste ; le stockage en mémoire n'est donc pas démesuré) :

```
for var in range(5) : # séquence des entiers de 0 à 4
    print(var)
```

produit l'affichage :

```
0
1
2
3
4
```

```
for var in range(10, 15) : # entiers de 10 à 14
    print(var)
```

produit l'affichage :

```
10
11
12
13
14
```

```
for var in range(10,20,2) : # entiers de 10 à 20, pas = 2
    print(var)
```

produit l'affichage :

```
10
12
14
16
18
```

B. Types de données en Python

En Python, toutes les données sont des objets. Les types de base sont les suivants :

entiers (int)

le « / » effectue une division de réels. Il faut utiliser « // » pour la division euclidienne

l'élévation à la puissance est possible grâce à l'opérateur « ** »

les entiers ne sont pas bornés :

```
from math import factorial
factorial(50)
```

```
30414093201713378043612608166064768844377641568960512000000000000
```

réels (float)

complexes (complex)

utiliser j ou J pour la partie imaginaire. Exemple : 3+5j

booléens :

2 valeurs True et False

sauf pour des cas précis ou si une méthode __bool__() est définie, tout objet est interprété comme « True » dans un contexte booléen

chaînes de caractères (str)

Les chaînes sont des *séquences immuables* de caractères. Les constantes peuvent être spécifiées entre guillemets (doubles) ou entre apostrophes (simples). Ainsi, toutes les opérations classiques sur les séquences peuvent être appliquées sur les chaînes de caractères. L'utilisation de 3 guillemets à suivre permet de représenter des chaînes de caractères de plusieurs lignes sans l'utilisation du '\n'.

listes

Les listes sont des séquences modifiables (ou *mutables*), représentées par des crochets. Python étant faiblement typé, mêmes si elles servent en général plutôt à contenir des données d'un même type, elles peuvent en réalité contenir des données de différents types. Elles peuvent être définies :

- En extension : [1, 2, 3, 4], []
- En compréhension : [(2*x) for x in range(5)] construit la séquence [0, 2, 4, 6, 8]

- En utilisant le constructeur list (itérable)

Pour l'accès à une valeur :

- liste[indice] (les indices commencent à 0)
- liste[-indice] (recherche à partir de la fin ; dernière position = -1

tuples

Les tuples sont des séquences immuables. Ils peuvent être définis :

- En extension : (), (1, 'a', True), (2,) (ne pas oublier la virgule dans le cas d'un singleton pour des problèmes de parsing)
- En utilisant le constructeur tuple (itérable)

Pour l'accès à une valeur :

- liste[indice] (les indices commencent à 0)
- liste[-indice] (recherche à partir de la fin ; dernière position = -1)

intervalles (range)

Construits grâce au constructeur range (), ils permettent de représenter tous les éléments d'un intervalle sans pour autant le construire. On gagne ainsi de la place. Ils peuvent être construits ainsi :

- range(fin)
- range(début, fin)
- range(début, fin, pas)

N.B. : le début est inclus, la fin est exclue.

ensembles (set – mutables et frozenset – immuables)

Les ensembles mutables peuvent être construits :

- en extension : {1, 2, 3}
- en utilisant le constructeur set()

Les ensembles non mutables ne peuvent être construits que par le constructeur frozenset()

tables associatives (dict)

Les tables associatives doivent avoir des clés d'un type « hashable », les valeurs sont des données quelconques. Elles peuvent être construites :

- En extension: {cle1 : valeur1, cle2 : valeur2, cle3 : valeur3}
- Via le constructeur de la class dict

NB : on peut connaître le type de n'importe quelle variable ou élément d'un type avec la fonction **`type(var)`**. Cette fonction prend en paramètre la variable ou l'élément dont on cherche le type et renvoie en résultat une description du type de la variable ou de l'élément.

C. Définition de fonctions

1. Syntaxe de base

Une fonction se définit principalement ainsi :

```
def nomFonction(paramètres) :
    """documentation"""
    bloc
```

La documentation est bien évidemment optionnelle, mais fortement recommandée ! Dans l'idéal, elle est constituée d'une première ligne, courte, de résumé, puis d'une ligne vide, et enfin d'un description plus détaillée sur plusieurs lignes.

Exemple :

```
def somme(a : int, b : int) → int:
    """Fonction renvoyant la somme de ses 2 arguments."""
    return a+b
```

Remarque : Dans l'exemple précédent, on a utilisé des **annotations de types** qui permettent d'indiquer que les deux paramètres a et b doivent être des entiers et que la fonction renvoie un entier.

NB : le type de retour d'une fonction qui ne renvoie rien est `NoneType` ou encore `None`.

L'appel se fait alors très classiquement :

```
    somme(2,3)
```

NB : les annotations de types sont indispensables pour déboguer efficacement votre programme (notamment avec Pycharm). **Elles seront toujours demandées en TP**.

2. Assertions et fonctions

Le langage python permet de définir des assertions permettant notamment de décrire des tests pour les fonctions. Elles se définissent à partir de la primitive `assert` qui prend en paramètres une expression booléenne et, de manière facultative (donc obligatoire !), une chaîne de caractères. Lorsque l'assertion est évaluée par l'interpréteur, son comportement est le suivant :

- si l'expression booléenne est évaluée à vrai, il passe à la suite
- si l'expression booléenne est fausse, il arrête l'exécution du programme à cette assertion en déclenchant une erreur. Si une chaîne de caractères a été fournie, il affichera cette chaîne comme message.

Exemples pour le cas de la fonction `somme` :

```
def somme(a : int, b : int) → int:
    """Fonction renvoyant la somme de ses 2 arguments."""
    return a+b
```

```
assert somme(2,3) == 5, "erreur sur le calcul de 2+3"
assert somme(2,0) == 2, "erreur sur un calcul avec 0"
assert somme(0,0)
```

A partir de la partie sur les fonctions, il vous sera régulièrement fourni des ensembles d'assertions que vos programmes devront satisfaire. Cela vous permettra de connaître facilement les différents cas de figure que votre programme doit être capable de traiter.

3. Valeurs par défaut des arguments

Les paramètres d'une fonction peuvent avoir une valeur par défaut, ce qui évite de les passer une valeur lors de l'appel de la fonction. On peut par exemple définir une fonction ainsi :

def sommeBis(a : int = 2, b : int = 1) → int :

    return a+b

On pourra alors effectuer les 2 appels suivants :

```
sommeBis(2,3)
```

```
5
```

```
sommeBis(6)
```

```
6
```

```
sommeBis()
```

```
3
```

4. Arguments nommés

On peut aussi appeler une fonction en nommant les arguments, ce qui permet de ne pas tenir compte de l'ordre de leur déclaration. On peut ainsi appeler également la fonction `sommeBis` ainsi :

```
sommeBis(b = 5, a = 4)
```

```
9
```

```
sommeBis(b = 2)
```

```
4
```

Le mélange des arguments non nommés et nommés est possible mais les arguments nommés doivent être en dernier, et ne peuvent correspondre à des arguments dont la valeur a déjà été fixé par des arguments non nommés. Ainsi, les appels `sommeBis(b=4, 5)` et `sommeBis(2, a=3)` sont illégaux.

5. Utilisation de listes ou de tables associatives en guise d'arguments

Il est possible d'utiliser des tables associatives pour passer des arguments à un fonction, moyennant que celle-ci ait pour ensemble de clés l'ensemble des noms des arguments de la fonction. Il faudra alors faire précéder la référence à la table associative de « ** ».

Exemple :

```
params= {'a': 6, 'b': 7}
sommeBis(**params)
```

On peut aussi utiliser une liste ou un tuple pour passer les arguments à une fonction. Il faut alors faire précéder la référence à la liste ou au tuple de « * ».

Exemples :

```
liste = [1,2]
sommeBis(*liste)
tuple = (5,6)
sommeBis(*liste)
```

6. Fonctions à nombres variable de paramètres

On peut rajouter à la fin de la liste des paramètres d'une fonction, dans sa définition, 2 pseudo-paramètres :

- un paramètre dont le nom est précédé par « * » : il aura pour valeur la liste des valeurs des paramètres non nommés surnuméraires
- un paramètre dont le nom est précédé par « ** » : il aura pour valeur la table associative (nomParam, valeurParam) pour les paramètres nommés passés qui ne correspondent à aucun des paramètres formels.

Ces 2 paramètres (combinables) permettent de définir des fonctions à nombre variable de paramètres.

Exemples :

```
def fonctionVar1(a: int, b: int, *reste) → None:
    print("a = " + str(a))
    print("b = " + str(b))
    print("reste = "+ str(reste))
fonctionVar1(1,2,3,4)
```

```
a = 1
```

```
b = 2

reste = (3,4)
```

```
def fonctionVar2(a: int, b: int, **reste) → None:
    print("a = " + str(a))
    print("b = " + str(b))
    print("reste = "+ str(reste))

fonctionVar2(d=4,c=3,b=2,a=1)
```

```
a = 1
b = 2
reste = {'d': 4, 'c': 3}
```

7. Lambda-fonctions

Il est également possible de définir des fonctions anonymes :

```
lambda listeParametres :
    bloc
```

Exemples :

```
>>> f = lambda x : x+1
>>> f(5)
6
>>> def g(fonc,x):
...     return fonc(x)*2
...
>>> g(f,3)
8
```

IV. Manipulation de quelques types importants

A. Les Séquences

Les listes, les tuples, les intervalles et les chaînes de caractères sont des séquences. Aussi, ces types partagent un grand nombre de choses en commun.

1. Accès à un élément

`sequence[indicePositif]` : renvoie l'élément à la position `indicePositif` à partir du début de la séquence, le premier élément étant à l'indice zéro.

`sequence[indiceNegatif]` : pour exprimer les indices à partir de la fin. Le dernier élément est à l'indice -1, le précédent à l'indice -2, etc.

Exemples :

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

```
2
```

```
(1,2)[0]
```

```
1
```

```
"Hello"[3]
```

```
'l'
```

```
"world"[-1]
```

```
'd'
```

2. Accès à une tranche d'élément (renvoie une copie)

sequence[debutInclus:finExclue]

sequence[:finExclue] : depuis le début

sequence[debutInclus:] : jusqu'à la fin

sequence[:] : tout ; pratique pour copier une séquence

Exemples :

```
"Tout le monde"[5:7]
```

```
'le'
```

```
"Tout le monde"[:4]
```

```
'Tout'
```

```
(35, 45, 75)[1:]
```

```
(45,75)
```

3. Concaténation

sequence1 + sequence2

Exemples :

```
[1,2]+[3,4,5]
```

```
[1, 2, 3, 4, 5]
```

```
(1,2)+(3,)
```

```
(1,2,3)
```

```
'Hello' + 'World'
```

```
'HelloWorld'
```

Remarque : les chaînes de caractères littérales peuvent être concaténées par simple juxtaposition : `'Hello' + ' ' + 'World'` peut être également écrit `'Hello' ' ' 'World'`. Ceci n'est plus valable dès l'instant où la chaîne de caractères est contenue dans une variable par exemple.

4. Comparaison

Les séquences sont comparables (via les opérateurs classiques de comparaison) en utilisant un ordre lexicographique.

```
    [2,1,3] < [2,2]
```

5. Longueur d'une séquence

```
len('HelloWorld')
len([1, 2, 3])
len((3,4))
```

6. Modification d'un séquence mutable (donc ni tuple, ni chaînes de caractères)

a. Les bases

```
a = [1, 2, 3, 4]
a[1] = 0
print(a)
```

```
[1, 0, 3, 4]
```

```
a[1:3] = [7, 8, 9]
print(a)
```

```
[1, 7, 8, 9, 4]
```

```
a.append(5)
print(a)
```

```
[1, 7, 8, 9, 4, 5]
```

```
b = a.pop()
print(b)
```

```
5
```

```
print(a)
```

```
[1, 7, 8, 9, 4]
```

```
c = a.pop(0)
print(c)
```

```
1
```

```
print(a)
```

```
[7, 8, 9, 4]
```

```
del a[1:3]
```

```
[7,4]
```

b. Et encore…

- Liste.append(…)
- Liste.remove(…)
- Liste.count(…)
- Liste.sort(…)
- Liste.reverse()
- Liste.insert(…)
- Liste.clear()
- Liste.index(…)
- Liste.extend(…)

B. Définition de listes en compréhension

Syntaxe :

`[expression for variable1 in for variable2 in .. if condition]`

Exemples :

```
[x*100 for x in range(1,4)]
```

```
[100, 200, 300]
```

```
[(x, y) for x in range(2) for y in range(2)]
```

```
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

```
[(x, y) for x in range(3) for y in range(3) if x <= y]
```

```
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
```

```
[[x for x in range(y)] for y in range(4)]
```

```
[[], [0], [0, 1], [0, 1, 2]]
```

C. Traitement de listes

1. Map

La fonction `map(fonction, itérable)` permet d'appliquer une fonction à tous les éléments d'une liste. Elle ne renvoie pas directement un objet liste, mais un objet map itérable, à partir duquel on peut donc facilement recréer un objet liste.

Exemple :

```
def double(x: float) → float:
    return 2*x

liste = [3, 7, 10]
doubles = list(map(double, liste))
print(doubles)
```

```
[6, 14, 20]
```

On peut bien sûr passer une lambda-expression comme fonction :

```
liste = [3, 7, 10]
doubles = list(map(lambda x: 2*x, liste))
print(doubles)
```

```
[6, 14, 20]
```

2. Filter

La fonction `filter(fonction, itérable)` prend en premier paramètre une fonction à valeur booléenne. Seuls les éléments de l'itérable pour lesquels cette fonction renvoie Vrai seront gardés dans le résultat, qui est un objet itérable de type *filter*.

Exemple :

```
pairs = list(filter(lambda x: x%2 == 0, range(0,20)))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. Réduction

La réduction permet d'appliquer une fonction f à 2 arguments sur tous les éléments d'un itérable pour renvoyer une valeur finale : le premier appel est effectué en passant à f les 2 premiers éléments de l'itérable. Puis pour chaque nouvelle valeur de l'itérable, la fonction est appelée avec en premier paramètre le résultat du calcul précédent et en deuxième paramètre la nouvelle valeur de la liste. Un cas classique d'utilisation consiste à calculer la somme des éléments d'une liste.

La réduction est possible grâce à la méthode `reduce` du module *functools*.

Exemple :

```
from functools import reduce
reduce(lambda x,y: x+y, range(11))
```

```
55
```

Remarque : il est possible de passer en troisième paramètre une valeur d'initialisation, utilisée également en cas de liste vide.

D. Les ensembles

1. appartenance

```
ensemble = {1, 2, 3, 4,1, 2, 3, 1}
print(ensemble)
```

```
{1, 2, 3, 4}
```

```
print(1 in ensemble)
```

```
True
```

```
print(5 in ensemble)
```

```
False
```

2. Union

```
ens1 = {1, 2, 3}
ens2 = {3,4, 5}
ens = ens1 | ens2
print(ens)
```

```
{1, 2, 3, 4, 5}
```

3. Intersection

```
ens1 = {1, 2, 3}
ens2 = {3,4, 5}
ens = ens1 & ens2
print(ens)
```

```
{3}
```

4. Différence

```
ens1 = {1, 2, 3}
ens2 = {3,4, 5}
ens = ens1 - ens2
print(ens)
```

```
{1, 2}
```

5. Union disjointe

```
ens1 = {1, 2, 3}
ens2 = {3,4, 5}
ens = ens1 ^ ens2
print(ens)
```

```
{1, 2, 4, 5}
```

E. Tables associatives (dict)

1. Utilisation de base

```
dict1 = {'bonjour': 'hello', 'aurevoir': 'bye', 'monsieur':
'sir'}
print(dict1['bonjour'])
```

```
'hello'
```

```
dict1['aurevoir'] = "goodbye"
print(dict1)
```

```
{'bonjour': 'hello', 'aurevoir': 'goodbye', 'monsieur': 'sir'}
```

```
del dict1['monsieur']
print(dict1)
```

```
{'bonjour': 'hello', 'aurevoir': 'goodbye'}
```

```
print('bonjour' in dict1)
```

```
True
```

```
print('monsieur' in dict1)
```

```
False
```

```
print('monsieur' not in dict1)
```

```
True
```

2. Autres utilisations

a. Création à partir d'une liste de couples

```
trad = dict([('hello', 'bonjour'),('goodbye','aurevoir')])
print(trad)
```

```
{'hello': 'bonjour', 'goodbye': 'aurevoir'}
```

b. Création à partir de paramètres nommés

```
trad = dict(hello = 'bonjour', goodbye = 'aurevoir')
print(trad)
```

```
{'hello': 'bonjour', 'goodbye': 'aurevoir'}
```

c. Création en compréhension

```
doubles = {x:2*x for x in range(0,5)}
print(doubles)
```

```
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
```

d. Parcours d'un dictionnaire

```
doubles = {x:2*x for x in range(0,5)}
for cle, valeur in doubles.items():
    print('2 * ' + str(cle) + ' = ' + str(valeur))
```

```
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
```

```
2 * 4 = 8
```

V. Entrées-Sorties

A. Affichage à l'écran

1. La fonction print

La principale façon d'afficher des données à l'écran consiste à passer par la fonction print(). Le profil précis de cette fonction est le suivant :

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Cela signifie que par défaut :

- la fonction *print* peut prendre plusieurs valeurs (séparées par des virgules)
- les différentes valeurs sont par défaut séparées par un espace à l'affichage
- à la fin d'un *print*, un retour à la ligne est automatiquement ajouté
- Par ailleurs, il n'y a pas de conversion implicite type → string quand l'opérateur de concaténation est utilisé. On ne peut donc pas écrire `print('a = ' + 4)`.

Exemples :

```
print('Hello')
print(2*3)
print('2 * 3 = ',2*3)
liste = [1, 2, 3]
print(liste)
print(4, 5, 6, sep=' - ')
print('liste = ', end='')
print(liste)

Hello
6
2 * 3 = 6
[1,2,3]
```

2. Conversion d'une donnée en chaîne de caractères

Il existe 2 fonctions permettant de convertir une donnée quelconque en chaîne :

`str()` : génère une représentation « humaine »

`repr()` : génère une représentation utilisable par python.

La différence est subtile. Elle peut cependant être envisagée sur le cas suivant :

```
bienvenue = 'Bonjour\n'
print(bienvenue)
```

```
Bonjour

```

```
print(repr(bienvenue))
```

```
'bonjour\n'
```

Noter :

- dans le premier cas, le retour chariot rajouté (il y a un saut de ligne après le « Bonjour »)
- dans le deuxième cas, les apostrophes et le « \n »

On peut donc écrire :

```
print('2*3 = ' + str(2*3))
```

Mais pour faire la même chose, il est malgré tout plus simple d'écrire :

```
print('2*3 =', 2*3)
```

3. Chaînes de caractères formatées

En faisant précéder une chaîne de caractères du caractère 'f', il est possible d'utiliser des noms de variables dans une chaîne de caractères, et de préciser le format d'affichage de ces variables.

Exemple sans instruction de formatage :

```
personne = {'nom': 'martin', 'prenom': 'jacques'}
age = 30
chaine = f"Je me prénomme {personne['prenom']} et j'ai {age} ans"
print(chaine)
```

```
Je me prénomme jacques et j'ai 30 ans
```

Quelques exemples avec formatage :

```
legumes = ['tomates', 'pommes de terre', 'carottes', 'navets',
'poivrons', 'haricots', 'petits pois', 'choux', 'céleris',
'fèves', 'courges']
for indice in range(len(legumes)):
    print(f"J'ai {indice:2} {legumes[indice]:.>20} dans mon
panier")
```

```
J'ai  0 ............tomates dans mon panier
J'ai  1 .....pommes de terre dans mon panier
J'ai  2 ............carottes dans mon panier
```

```
J'ai  3 ..............navets dans mon panier
J'ai  4 ............poivrons dans mon panier
J'ai  5 ............haricots dans mon panier
J'ai  6 .........petits pois dans mon panier
J'ai  7 ...............choux dans mon panier
J'ai  8 ............céleris dans mon panier
J'ai  9 ...............fèves dans mon panier
J'ai 10 ............courges dans mon panier
```

Explications :

le «`:2` » pour l'affichage de l'indice précise qu'une place de 2 caractères est prévue. Les nombres sont par défaut alignés à droite.

Le «`:.>20` » pour l'affichage du nom du légume précise que 20 caractères sont réservés, et que les noms de légumes seront alignés à droite (par défaut, les chaînes de caractères sont alignées à gauche). Par ailleurs, les espaces insérés seront remplacés par des « . ».

4. Utilisation de la méthode format()

La méthode format de la classe string permet de passer à une chaîne formatée des valeurs pour chacun des champs figurant dans la chaîne. Les champs peuvent être spécifiés soit par un numéro, soit par un nom. Pour chaque champ, un format, similaire à celui présenté ci-dessus, peut être précisé.

Exemple 1 :

```
texte1 = "J'ai {1:4} {0} dans mon {2}"
texte2 = texte1.format("carottes", 2, "panier")
print(texte2)
```

```
J'ai    2 carottes dans mon panier
```

Exemple 2 :

```
texte3 = "J'ai {quantite} {legume} dans mon {contenant}"
texte4 = texte3.format(legume='carottes', quantite=2,
contenant="panier")
print(texte4)
```

```
J'ai 2 carottes dans mon panier
```

B. Lecture au clavier

On utilise tout simplement la fonction `input()` qui prend en paramètre un phrase d'invite et qui renvoie une chaîne de caractères. Attention : la fonction input renvoie ce qui a été saisi sous la forme d'une chaîne de caractères. Si l'on veut obtenir un nombre alors il faut utiliser une fonction de conversion vers les entiers/les réels :`int()` ou `float()`.

Exemple :

```
saisie = input("Entrez un nombre : ")
resultat = 2 * saisie
print(resultat)
```

```
Entrez un nombre : 12
1212
```

```
nombre = int(saisie)
resultat = 2*nombre
print("Le double de",nombre,"est",resultat)
```

```
Entrez un nombre : 12
Le double de 12 est 24
```

C. Lecture/Écriture dans des fichiers

1. Ouverture du fichier

Il faut commencer par ouvrir le fichier :

```
fichier = open("nomFichier", mode)
```

où mode peut valoir :

- "r " : read
- "w" : write (on écrase le contenu précédent)
- "a" : append (on ajoute au contenu précédent)

2. Lecture depuis un fichier

- `fichier.read()` : lit tout le fichier
- `fichier.read(taille)` : lit taille octets maximum
- `fichier.readline()` : lit la ligne suivante (y compris le retour-chariot)
- À noter : lorsque la fin du fichier est atteinte, la lecture renvoie la chaîne vide.

3. Écriture dans un fichier

```
fichier.write(chaine)
```

4. Fermeture du fichier

```
fichier.close()
```

VI. Modules

Tout fichier toto.py définit un module toto. Lorsqu'on est dans l'interpréteur de Python, on est dans un module par défaut appelé `__main__`.

Lorsqu'on veut utiliser un module *nomModule*, il faut écrire :

```
import nomModule
```

Les modules sont recherchés dans les répertoires contenus dans la variable sys.path, initialisée avec :

- le répertoire du module courant
- le contenu de la variable d'environnement PYTHONPATH
- l'initialisation propre à l'installation.

Une fois un module importé, ce qu'il définit (variables, fonctions) peut être utilisée en utilisant la notation `nomModule.nomVariable`.

Exemple :le module math

Le module math permet d'accéder à des fonctions mathématiques sur les réels.

```
import math
math.factorial(20)
```

On peut également lier une variable de l'espace des noms courants à une variable du module afin de se dispenser de la notation pointée :

```
from math import sin, cos
sin(1)
cos(0)
```

On peut également affecter de nouveaux noms locaux :

```
from math import sin as sinus
sinus(3.14/2)
```

Exemple 2 : le module random

Le module random permet d'accéde à des fonctions permettant de faire des tirages aléatoires. Voir le fichier *random.py* pour des exemples.

Exemple 3 : le module matplotlib

Il existe également de nombreuses librairies externes qui peuvent être utilisée en Python via des modules. On trouve notamment des librairies mathématiques, pour la gestion de bases de données, ou encore pour faire des statistiques. En particulier, la librairie *matplotlib* permet de facilement tracer différentes sortes de courbes et graphiques.

Pour avoir un descriptif plus détaillé des modules évoqués ou découvrir d'autres modules, voir ce lien : https://docs.python.org/fr/3/tutorial/stdlib.html ou encore https://docs.python.org/fr/3/tutorial/stdlib2.html.

VII. Exceptions

Les exceptions fonctionnent de la même façon que dans de nombreux autres langages, comme Java par exemple. Une exception est un objet sous-type de la classe Exception.

Introduire un bloc d'exceptions :

```
    try :
```

Attraper des exceptions :

```
    except TypeException :
    except TypeException as objetDescripteur :
```

Déclencher une exception

```
    raise NomException
    raise NomException(paramètres)
```

Propager une exception attrapée

```
    raise
```

Exemple (d'après docs.python.org) :

```
while True:
    try:
         x = int(input("Entrez un nombre : "))
         break
    except ValueError:
         print("Ce n'est pas un nombre… ")
```

VIII. Classes et objets

A. Introduction

En Python, les classes sont définies grâce au mot-clé *class*. La création d'une classe implique la création d'un espace de nom associé à la classe. Au sein d'une classe, on peut définir :

- une documentation (chaîne """… """ juste après la déclaration)
    - sera affichée si on utilise la commande *help* suivie du nom de la classe
- des variables de classe ie des variables rattachées à la classe et donc uniques

- des fonctions
- un constructeur permettant de créer des objets instances de la classe. Leur type sera le nom de la classe.

Les fonctions pourront être utilisées, suivant leur appel, comme des méthodes de classe (ne sont pas appliquées à un objet) ou comme des méthodes d'instance (ie. s'appliquant à un objet instance de la classe).

Attention ! Il n'y a pas de notion de droit d'accès aux fonctions et variables. Cependant, par convention, une variable ou une fonction dont le nom commence par un tiret de soulignement pourra être considérée comme privée ie. utilisables uniquement à l'intérieur de la classe.

B. Variables de classe

Toute variable initialisée directement dans une classe sera considérée comme une variable de classe.

Exemple :

```
class Personne:
    nbPersonnes = 0

print(Personne.nbPersonnes)
```

```
0
```

```
Personne.nbPersonnes = Personne.nbPersonnes+1
print(Personne.nbPersonnes)
```

```
1
```

C. Constructeur et variables d'instance

Le constructeur d'une classe est une fonction particulière, appelée `__init__()`, dont le premier paramètre, appelé *self* par convention, est une référence sur l'objet créé. Pour créer un objet, on écrit juste le nom de la classe suivi de parenthèses.

Exemple 1 :

```
class Personne:
    def __init__(self: 'Personne') → None:
        print("Appel du constructeur")

# pers va contenir une référence à un objet de type Personne
pers = Personne()
```

Les variables d'instance (ie. les variables propres à l'objet créé) sont créées dans le constructeur de la classe en ajoutant des attributs à l'objet `self`. Elles n'apparaissent pas clairement dans la structure de la classe comme

en Java par exemple. D'ailleurs, il est toujours possible de rajouter des variables à un objet, là encore contrairement à Java.

Exemple 2 :

```
class Personne:
    """représentation d'une personne.

    Une personne est décrite par son prénom et son nom."""
    def __init__(self: 'Personne') → None:
        self.nom = "inconnu"
        self.prenom = "inconnu"

pers = Personne()
```

Il n'est pas possible de définir plusieurs constructeurs pour une classe. Pour la classe Personne, si on veut avoir des façons différentes de créer des instances, il faut utiliser les valeurs par défaut des paramètres du constructeur, le nombre variable de paramètres, un contrôle sur les types des paramètres, etc.

Voici une nouvelle version de la classe Personne :

Exemple 3 :

```
class Personne:
    """représentation d'une personne.

    Une personne est décrite par son prénom et son nom."""
    def __init__(self, nom: str = "inconnu", prenom: str = ""):
        self.nom = nom
        self.prenom = prenom

pers1 = Personne()            # personne de nom "Inconnu"
pers2 = Personne("Martin")    # personne sans prénom
pers3 = Personne("Martin", "Jacques")
pers4 = Personne(prenom="Jacques", nom="Martin")
```

D. Méthodes d'instance

Les fonctions définies dans une classe peuvent a priori être utilisées comme des méthodes de classe. Cependant, si on appelle cette fonction en l'appliquant à un objet instance de la classe avec un « `.` », elle sera considérée comme une méthode d'instance. Elle sera donc dans ce cas appliqué à l'objet. Son premier paramètre sera alors initialisé avec une référence à l'objet auquel la méthode est appliquée, et sera appelé `self` par convention.

Exemple :

```
class Point:
    def __init__(self : 'Point', x : int = 0, y : int = 0):
        self.x = x
        self.y = y
```

```
    def translater(self, dx: int = 0, dy: int = 0):
        self.x += dx
        self.y += dy

p1 = Point(3,4)
p1.translater(3,2)
print(p1.x,p1.y)
```

```
6 6
```

Attention : contrairement à la plupart des langages à base de classes, mais comme dans un langage à base de prototypes comme javascript, en Python, on peut :

- ajouter des attributs à un objet : à la fin du programme précédent, on peut écrire `p1.z = 10` ce qui aura pour effet d'ajouter une variable d'instance z à l'objet référencé par p1.
- supprimer des attributs à un objet : à la fin du programme précédent, on peut écrire `del p1.x` ce qui aura pour effet de supprimer la variable d'instance z de l'objet référencé par p1.

E. Affichage automatique sous forme de chaîne de caractères

Pour avoir une chaîne de caractères correspondant à l'état d'un objet, on peut ajouter à la classe dont il est instance une fonction __str__(). On peut donc réécrire la classe Personne ainsi :

```
class Personne:
    """représentation d'une personne.

    Une personne est décrite par son prénom et son nom."""
    def __init__(self :'Personne', nom: str = "inconnu",
                 prenom: str = ""):
        self.nom = nom
        self.prenom = prenom

    # la fonction __str__ d'une classe permet lorsqu'elle
    # est appliquée à un objet instance de la classe de
    # renvoyer une chaine de caractères représentant l'état
    # de l'objet.
    def __str__(self) → str:
        return self.prenom + " " + self.nom
```

F. Héritage

Pour spécifier qu'une classe hérite d'une (ou plusieurs) autres – l'héritage multiple est en effet autorisé dans Python – , on met le(s) nom(s) de la (des) classe(s) mère(s) entre parenthèse après le nom de la classe. Le chaînage des constructeurs doit être mis en place explicitement en appelant la méthode `__init__` de la classe mère explicitement, avec 2 syntaxes différentes :

Exemple (version 1) :

```
class Etudiant(Personne):
    def __init__(self, nom: str = "inconnu", prenom: str = "",
                 filiere  str = "université") → None:
        Personne.__init__(self, nom, prenom)
        self.filiere = filiere

    def __str__(self) → str:
        return Personne.__str__(self) + f" ({self.filiere})"

etu1 = Etudiant("Martin", "Jacques", "DUT Informatique")
print(etu1)
```

```
Jacques Martin (DUT Informatique)
```

Comme on peut le voir, en mentionnant le nom de la super-classe, il faut repasser l'objet « self »

Exemple (version 2) :

```
class Etudiant(Personne):
    def __init__(self, nom: str = "inconnu", prenom: str = "",
                 filiere: str = "université") → None:
        super().__init__(nom, prenom)
        self.filiere = filiere

    def __str__(self) → str:
        return super().__str__() + f" ({self.filiere})"

etu1 = Etudiant("Martin", "Jacques", "DUT Informatique")
print(etu1)
```

```
Jacques Martin (DUT Informatique)
```

En utilisant la fonction `super()`, il ne faut pas repasser l'objet « self ».

IX. Structure itérable, itérateur et générateur

A. Intérêt des notions d'itérable et d'itérateur

Une boucle for permet de parcours n'importe quelle structure de donnée « *itérable* ». Pour qu'une structure de données soit *itérable*, il faut qu'elle définisse une méthode `__iter__()`. Une telle méthode doit renvoyer un *itérateur*, c'est-à-dire un objet avec une méthode `__next__()` qui, à chaque appel, renvoie la donnée suivante de la structure de donnée itérable. Un procédé classique en python consiste à décrire la méthode `__next__()` dans la classe définissant la stucture de donnée itérable elle-même, ce qui fait que la méthode `__iter__()` peut se contenter de renvoyer l'objet « self » (mais cela empêche d'avoir plusieurs itérateurs simultanés sur la même donnée).

B. Exemple de définition d'une structure itérable et d'un itérateur

```
class Pile:
    def __init__(self):
        self.donnees = []

    def empiler(self, donnee : any) → None:
        self.donnees.append(donnee)

    def depiler(self) → any:
        return self.donnees.pop()

    def sommet(self) → any:
        return self.donnees[-1]

    def __str__(self) → str:
        retour = ""
        for i in range(1, len(self.donnees)+1):
            retour += str(self.donnees[-i]) + "\n"
        return retour

    def __iter__(self):
        return Pile.IterPile(self)

    class IterPile:
        def __init__(self, source) → None:
            self.source = source
            self.indice = 0
            self.fin = -len(source.donnees)
        def __next__(self) → any:
            if self.indice <= self.fin:
                raise StopIteration
            self.indice = self.indice - 1
            return self.source.donnees[self.indice]
```

On peut alors parcourir une pile avec un « for » ainsi :

```
pile = Pile()
pile.empiler(2)
pile.empiler(3)
for val in pile:
    print(val)
```

C. Générateur

Un générateur est une fonction qui, grâce à l'instruction yield, permet d'implanter facilement la notion d'itérateur. L'instruction yield permet, comme un « return », de sortir d'une méthode en renvoyant une valeur, mais elle mémorise également le contexte d'exécution pour que l'exécution reprenne à l'instruction suivante au prochain appel.

On peut donc également parcours la pile ci-dessus en définissant un générateur plutôt qu'un itérateur ainsi :

```
class Pile:
    def __init__(self):
        self.donnees = []

    def empiler(self, donnee):
        self.donnees.append(donnee)

    def depiler(self):
        return self.donnees.pop()

    def sommet(self):
        return self.donnees[-1]

    def __str__(self):
        retour = ""
        for i in range(1, len(self.donnees)+1):
            retour += str(self.donnees[-i]) + "\n"
        return retour
    def __iter__(self):
        return Pile.IterPile(self)

    def parcours(self):
        for i in range(len(self.donnees)):
            indice = -1-i
            yield self.donnees[indice]

pile = Pile()
pile.empiler(2)
pile.empiler(3)
for val in pile.parcours():
    print(val)
```