Programmation Python,

les bases

Sébastien Jeudy

Sommaire

1 TECHNIQUES DE BASE........6

1.1 Le langage Python........7
1.2 L'interpréteur de commandes Python........8
1.2.1 Accès........8
1.2.2 Saisie de nombres........9
1.2.3 Calculs et opérateurs........10
1.2.4 Commentaires........10
1.3 Un programme Python........11
1.3.1 Sur Windows........11
1.3.2 Sur Unix / Linux / macOS........13
1.3.3 Encodage des caractères........15
1.4 Variables........16
1.4.1 Création........16
1.4.2 Types de données........18
1.4.3 Chaînes de caractères........19
1.4.4 Opérations complémentaires........21
1.4.5 Fonctions utiles........24
1.5 Structures conditionnelles........27
1.5.1 Structures if, else, elif........27
1.5.2 Opérateurs de comparaison........30
1.5.3 Expressions booléennes........30
1.5.4 Mots-clés or, and, not........32
1.5.5 Structure match, case........33
1.6 Structures itératives........34

1.6.1 Boucle while....34
1.6.2 Boucle for....35
1.6.3 Mots-clés break et continue....37
1.7 Fonctions....38
1.7.1 Définition....38
1.7.2 Sans paramètres....39
1.7.3 Avec paramètres....40
1.7.4 Valeur par défaut des paramètres....42
1.7.5 Appels avec paramètres nommés....43
1.7.6 Signature....45
1.7.7 Renvoi de valeurs avec return....46
1.7.8 Portée des variables....47
1.7.9 Docstrings....48
1.7.10 Fonctions lambda....49
1.8 Modules....50
1.8.1 Définition....50
1.8.2 Importation classique....50
1.8.3 Espaces de noms....52
1.8.4 Importation avec espace de noms personnalisé (alias)....52
1.8.5 Importations spécifiques....53
1.8.6 Création....54
1.9 Packages....57
1.9.1 Définition....57
1.9.2 Création et importations....58
1.10 Exceptions....61
1.10.1 Définition....61
1.10.2 Interceptions de base....62
1.10.3 Mots-clés else et finally....64
1.10.4 Mot-clé pass....65
1.10.5 Assertions....66

1.10.6 Lever une exception....67

2 OBJETS ESSENTIELS....68

2.1 Notion d'objet....69
2.2 Chaînes de caractères....69
2.2.1 Classe str....69
2.2.2 Méthodes de mise en forme....71
2.2.3 Formatages....73
2.2.4 Concaténation classique....76
2.2.5 Parcours de chaînes....77
2.2.6 Découpage de chaînes....79
2.2.7 Compter, rechercher, remplacer....80
2.3 Listes et tuples....81
2.3.1 Classe list....81
2.3.2 Accès et modification des objets....82
2.3.3 Insertion d'objets....83
2.3.4 Concaténation de listes....85
2.3.5 Suppression d'objets....86
2.3.6 Parcours de listes....88
2.3.7 Tuples....91
2.3.8 Passage d'une chaîne à une liste et inversement....94
2.3.9 Fonction avec un nombre indéterminé de paramètres....95
2.3.10 Transformer une liste ou un tuple en paramètres de fonction....98
2.3.11 Compréhensions de liste....99
2.4 Dictionnaires et ensembles....101
2.4.1 Classe dict....101
2.4.2 Insertion, accès et modification des objets....102
2.4.3 Suppression d'objets....104
2.4.4 Application : dictionnaire de fonctions....105

2.4.5 Parcours de dictionnaires....107
2.4.6 Fonction avec un nombre indéterminé de paramètres nommés....111
2.4.7 Transformer un dictionnaire en paramètres nommés de fonction....113
2.4.8 Ensembles (sets)....114
2.5 Référence des objets....116
2.6 Fichiers....119
2.6.1 Module os....119
2.6.2 Répertoire courant....119
2.6.3 Ouverture d'un fichier....120
2.6.4 Fermeture d'un fichier....121
2.6.5 Lecture d'un fichier....122
2.6.6 Ecriture dans un fichier....124
2.6.7 Position dans le fichier....125
2.6.8 Ouverture avec with, as....127
2.6.9 Ecriture et lecture d'objets....128

1 Techniques de base

1.1 Le langage Python

Python est un **langage de programmation interprété**.

Date de première version : 20 février 1991
Auteur : Guido van Rossum (Néerlandais)
Développeurs actuels : Python Software Foundation / PSF (depuis 2001)

Paradigmes : impératif, structuré, fonctionnel et orienté objet
Typage : dynamique fort

Systèmes d'exploitation : multiplateformes
Licence : libre

Site Web : https://www.python.org ⇒ *téléchargements, documentations,...*

Versions majeures :

1991 : Python 0.9
1994 : Python 1.0
2000 : Python 2.0
2008 : Python 3.0 (attention : rupture de compatibilité avec Python 2) ⇒ ***ici est traité Python 3***

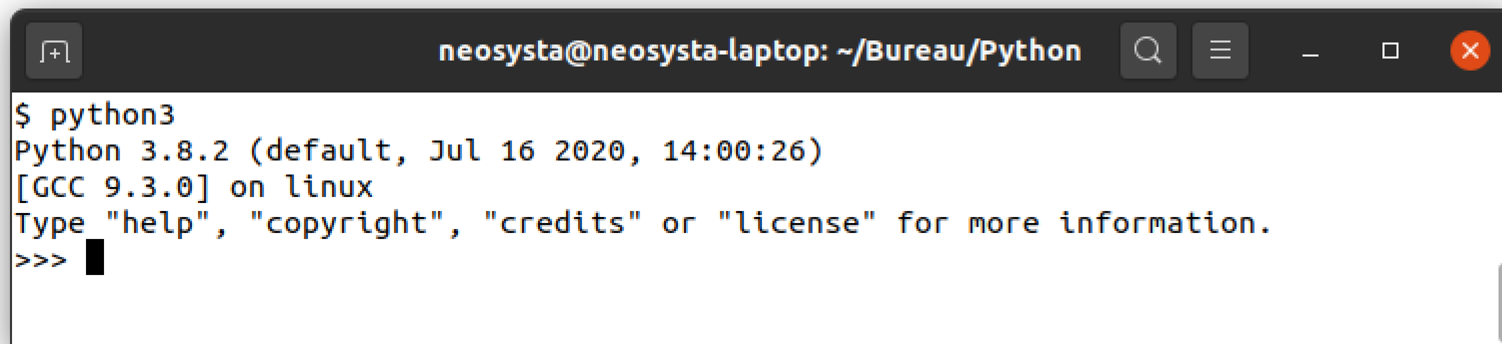
1.2 L'interpréteur de commandes Python

1.2.1 Accès

Les programmes en Python sont directement exécutés par l'**interpréteur Python** (abordé plus loin), mais ce dernier est également **utilisable en lignes de commandes :**

1) Ouvrir une **Console Windows** ou un **Terminal Unix / Linux / macOS**.

2) Saisir la **commande « python » sur Windows** ou **« python3 » sur Unix / Linux / macOS**.



⇒ *vérifier la version*

>>> correspond à l'invite de commande (prompt).

Dans l'interpréteur Python, **l'instruction « exit() » quitte l'interpréteur**. ⇒ *aussi les programmes*

Utiliser l'interpréteur Python en lignes de commandes est **pratique pour tester des instructions ou des séquences de codes** ("en live", sans sauvegarde).

1.2.2 Saisie de nombres

L'interpréteur Python en lignes de commandes **accepte la saisie directe de nombres :**

Interpréteur Python :

```
>>> 12
12
>>> -5.7
-5.7
```

⇒ *notation anglo-saxonne : le point correspond à la virgule*

1.2.3 Calculs et opérateurs

L'interpréteur Python en lignes de commandes **accepte aussi la saisie directe de calculs :**

Interpréteur Python :

```
>>> (5 + 4) / 3 + 2.3 * 10 - 6.5
19.5
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 7 % 3
1
```

⇒ *// correspond à la division entière et % au modulo (reste de la division entière)*

1.2.4 Commentaires

En Python, un commentaire **commence par un # (dièse)** et se termine par la fin de ligne.

1.3 Un programme Python

1.3.1 Sur Windows

1) **Accéder à un éditeur de texte brut** (Bloc-notes, Notepad++,…) **et écrire le programme :**

```
print("Hello, World!")
input()
```

⇒ *attention à la casse*

2) **Dans le dossier au choix, enregistrer le fichier sous le nom (par exemple) :** helloworld.py

⇒ *l'extension « .py » est obligatoire sur Windows pour que le fichier soit exécuté avec Python*

3) Avec l'Explorateur de fichiers, aller dans le dossier en question et **pour exécuter le programme double-cliquer sur l'icône du fichier** « helloworld.py ».

⇒ *le programme s'exécute dans une console et « input() » permet d'y rester jusqu'à [ENTRÉE]*

Autre solution d'exécution du programme :

1) **Ouvrir une Console Windows et à l'aide de la commande « cd » aller dans le dossier du fichier** « helloworld.py ».

2) **Pour exécuter le programme, saisir la commande :**

python helloworld.py

⇒ *en mode console, « input() » est inutile en fin de programme*

1.3.2 Sur Unix / Linux / macOS

1) **Ouvrir un Terminal Unix / Linux / macOS et à l'aide de la commande « cd » aller dans votre répertoire de travail**.

2) **Accéder à un éditeur de texte brut** (nano, vim, emacs,…) **pour éditer le fichier** « helloworld.py », par exemple :

```
nano helloworld.py
```

⇒ *l'extension « .py » n'est pas obligatoire sur Unix / Linux / macOS*

3) **Dans l'éditeur de texte, écrire le programme :**

```
#!/usr/bin/python3
print("Hello, World!")
```

⇒ *attention à la casse et à la version de Python (vérifier si nécessaire)*

4) **Enregistrer le fichier, retourner dans le Terminal et ajouter les permissions d'exécution au fichier** « helloworld.py » :

chmod ugo+x helloworld.py

5) **Pour exécuter le programme, saisir la commande :**

./helloworld.py

Autre solution d'exécution du programme (shebang et « chmod » inutiles mais hors normes) :

python3 helloworld.py

⇒ *attention à la version de Python (vérifier si nécessaire)*

1.3.3 Encodage des caractères

Par défaut, Python considère que ses fichiers sources sont encodés en UTF-8.

Idéalement, **pour éviter les problèmes d'encodage des caractères** des fichiers sources en Python, **ajouter la ligne suivante au début du code source :**

Par exemple sur Windows (à adapter si nécessaire) :

 # -*- coding: cp1252 -*-

⇒ *en 1ère ligne*

Normalement sur Unix / Linux / macOS :

 # -*- coding: utf-8 -*-

⇒ *sous le shebang, donc en 2ème ligne (mais logiquement inutile)*

1.4 Variables

1.4.1 Création

En Python, une variable existe à partir du moment où on lui affecte une valeur sous la forme :

nom_variable = valeur

C'est également au moment de cette affection qu'elle obtient son type (mais qui peut changer !).

Pour le nom des variables :

Lettres minuscules ou majuscules, chiffres et underscore « _ ».
Ne doit pas commencer par un chiffre.

⇒ *conseil de nommage : mon_age ou monAge*

Rappel : Python est sensible à la casse (différence entre minuscules et MAJUSCULES).

Python n'a pas besoin de point-virgule (;) pour terminer les instructions (en fin de ligne).

Les points-virgules sont uniquement utilisés pour séparer plusieurs instructions sur la même ligne.

Interpréteur Python :

```
>>> nombre1 = 100
>>> nombre1
100
```

Dans l'interpréteur de commandes, on peut afficher la valeur d'une variable en tapant seulement son nom (pas dans un programme Python !). **Autres opérations possibles (classiques) :**

Interpréteur Python :

```
>>> nombre1 = nombre1 + 50
>>> nombre1
150
>>> nombre2 = nombre1 / 3
>>> nombre2
50.0
```

<u>Attention</u> : ne pas utiliser les mots-clés de Python pour les noms créés (ils sont réservés).

1.4.2 Types de données

Python associe à chaque donnée un **type qui définit les opérations autorisées** sur celle-ci.

<u>Principaux types de données</u> : ⇒ *d'autres types existent*

int (entier), **float** (décimal), **str** (chaîne de caractères), **bool** (booléen, True ou False).

Une variable obtient un type à chaque affection, qui peut donc changer :

Interpréteur Python :

```
>>> variable = 10
>>> variable = "texte"
>>> variable
'texte'
```

1.4.3 Chaînes de caractères

Une chaîne de caractères **doit être entourée de délimiteurs :**

- guillemets : "exemple de chaîne de caractères"
- apostrophes : 'exemple de chaîne de caractères'
- triples guillemets : """exemple de chaîne de caractères"""
⇒ *il existe aussi les triples apostrophes (similaires aux triples guillemets)*

L'antislash **« \ » sert à échapper** notamment les apostrophes et les guillemets :

Interpréteur Python :

```
>>> texte = 'J'ai un problème'
  File "<stdin>", line 1
    texte = 'J'ai un problème'
                ^
SyntaxError: invalid syntax
>>> texte = 'Je n\'ai plus de problème'
>>> texte
"Je n'ai plus de problème"
```

<u>Précision</u> : pour écrire un antislash dans une chaîne, il faut l'échapper lui-même (« \\ »).

Les triples guillemets (ou les triples apostrophes) dispensent d'échapper les guillemets et apostrophes, et permettent d'écrire un texte sur plusieurs lignes :

Interpréteur Python :

```
>>> texte = """J'arrive
... à écrire
... des lignes"""
>>> texte
"J'arrive\nà écrire\ndes lignes"
```

⇒ *les retours à la ligne sont automatiquement remplacés par des « \n »*

1.4.4 Opérations complémentaires

Incréments :

Interpréteur Python :

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

⇒ *les opérateurs -=, *= et /= existent également (mais pas ++ et -- !)*

Python permet facilement d'**échanger les valeurs de deux variables :**

Interpréteur Python :

```
>>> i = 10
>>> j = 20
>>> i,j = j,i
>>> i
20
>>> j
10
```

Ou encore d'**affecter la même valeur à plusieurs variables :**

Interpréteur Python :

```
>>> i = j = k = 10
>>> i
10
>>> j
10
>>> k
10
```

On peut également **découper les longues instructions sur plusieurs lignes à l'aide de « \ » :**

Interpréteur Python :

```
>>> (5 + 30 - 6) \
... * (2 + 25 - 10) \
... / (13 - 15)
-246.5
```

⇒ *utile surtout dans un code source*

Enfin, **pour une amélioration de la rigueur de programmation :**

Python 3.6 a introduit l'annotation « nom: str = "DUPONT" ».

Mais ensuite tout de même possible « nom = 50 » !

⇒ *uniquement pour la lisibilité des programmes (aucuns contrôles)*

1.4.5 Fonctions utiles

Comme tous les langages, **Python dispose de fonctions intégrées et permet d'en définir** (abordé plus loin). Appel d'une fonction :

nom_fonction(parametre1, parametre2,…) ⇒ *sans paramètres : nom_fonction()*

La fonction « type » renvoie le type de ce qu'on lui passe en paramètre :

Interpréteur Python :

```
>>> nombre = 10
>>> type(nombre)
<class 'int'>
>>> type(1.23)
<class 'float'>
>>> type("texte")
<class 'str'>
>>> type(True)
<class 'bool'>
```

⇒ *en Python, les types de données sont définis sous forme de classes d'objets*

La fonction « print » affiche dans la console ce qu'on lui passe en paramètres (indispensable pour les affichages dans un programme) :

Interpréteur Python :

```
>>> nombre1 = 10
>>> nombre2 = 20
>>> print(nombre1)
10
>>> print("1er nombre =", nombre1, "et 2ème nombre =", nombre2)
1er nombre = 10 et 2ème nombre = 20
```

⇒ *Python affiche tous les paramètres transmis séparés par un espace*

Utilisation directe des **« \n » (retour à la ligne) et « \t » (tabulation) :**

Interpréteur Python :

```
>>> print("Bonjour\n\n\n\t\t\tà tous")
Bonjour



			à tous
```

La fonction « input » récupère une saisie au clavier :

input() ⇒ *attend la saisie de la touche [ENTRÉE]*

variable = input("Saisissez quelque chose : ") ⇒ *affiche le texte et la variable récupère la saisie*

Attention : **la saisie est récupérée sous forme de chaîne de caractères** (type « str »).

Si nécessaire, utiliser les **fonctions de conversion de type :** ⇒ *mêmes noms que les types*

Convertir en entier : **int(...)**
Convertir en décimal : **float(...)**
Convertir en booléen : **bool(...)**
Et convertir en texte : **str(...)**

Interpréteur Python :

```
>>> age = input("Saisissez votre âge : ")
Saisissez votre âge : 50
>>> type(age)
<class 'str'>
>>> age = int(age)
>>> type(age)
<class 'int'>
```

1.5 Structures conditionnelles

1.5.1 Structures if, else, elif

Code Python :

```
if age >= 18:
    print("Vous êtes majeur.")
```

Attention :

Les structures complexes en Python n'ont pas de début et de fin explicites, ni d'accolades qui pourraient marquer là où commence et se termine le bloc d'instructions.

Le seul délimiteur sont les deux points (:) et l'indentation du code lui-même (obligatoires).

La recommandation pour chaque niveau d'indentation est de 4 espaces.

Syntaxe également possible : if int(nombre) / 2 >= 50:

Code Python :

```
if age >= 18:
    print("Vous êtes majeur.")
else:
    print("Vous êtes mineur.")
```

Code Python :

```
if note >= 16 and note <= 20:
    print("Très Bien")
else:
    if note >= 14 and note < 16:
        print("Bien")
    else:
        if note >= 12 and note < 14:
            print("Assez Bien")
        else:
            if note >= 10 and note < 12:
                print("Passable")
            else:
                if note >= 0 and note < 10:
                    print("Ajourné")
                else:
                    print("La note doit être comprise entre 0 et 20 !")
```

Tous les « else: if » peuvent être remplacés par des « elif », simplifie la structure et les indentations :

Code Python :

```
if note >= 16 and note <= 20:
    print("Très Bien")
elif note >= 14 and note < 16:
    print("Bien")
elif note >= 12 and note < 14:
    print("Assez Bien")
elif note >= 10 and note < 12:
    print("Passable")
elif note >= 0 and note < 10:
    print("Ajourné")
else:
    print("La note doit être comprise entre 0 et 20 !")
```

⇒ *un seul « else: » à la fin (optionnel)*

Dans tous les cas, attention aux « : » et à l'indentation.

1.5.2 Opérateurs de comparaison

Pour les conditions de test :

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

⇒ *« == » et non « = » (affectation)*

1.5.3 Expressions booléennes

Les conditions de test sont des expressions booléennes également appelées prédicats.

Leur résultat est un **booléen (type « bool ») : True** (vrai) **ou False** (faux).

Interpréteur Python :

```
>>> age = 15
>>> age == 18
False
>>> age != 18
True
>>> age < 18
True
```

Les valeurs booléennes **« True » et « False » peuvent être affectées à des variables :**

majeur = False

Une condition de test sur une variable non définie génère une erreur.

1.5.4 Mots-clés or, and, not

Comme dans tous les langages, **entre deux conditions de test :**

« or » est utilisé **quand l'une OU l'autre condition** doit être vraie (True).
« and » est utilisé **quand l'une ET l'autre condition** doivent être vraies (True).

⇒ *le résultat de ces opérateurs logiques est également un booléen*

Exemple :

```
if note >= 16 and note <= 20:
```

« not » inverse un prédicat : « not age == 18 » équivaut donc à « age != 18 ».

Pour tester un booléen, préférer « if majeur: » / « if not majeur: ».

(plutôt que « majeur == True » / « majeur != True » ou « majeur is True » / « majeur is not True »)

Les parenthèses ( ) permettent également de **forcer les priorités** (rappel : ET prioritaire sur OU).

1.5.5 Structure match, case

Python 3.10 a introduit la structure « match, case » (différents types d'évaluations existent) :

Code Python :

```
match jour:     # variable testée (ici un numéro de jour), suivie de chaque cas à évaluer
    case 1:
        print("Lundi")
    case 2:
        print("Mardi")
    case 3:
        print("Mercredi")
    case 4:
        print("Jeudi")
    case 5:
        print("Vendredi")
    case 6:
        print("Samedi")
    case 7:
        print("Dimanche")
    case _:     # autres cas
        print("Jour inconnu")
```

1.6 Structures itératives

1.6.1 Boucle while

Similaire aux autres langages, **répète un bloc d'instructions tant qu'une condition est vraie :**

Code Python :

```
mot = "pasfin"
while mot != "fin":
    mot = input("Tapez 'fin' pour arrêter : ")
```

Code Python :

```
nombre = 1

while nombre <= 100:
    print(nombre)
    nombre += 1
```

1.6.2 Boucle for

Différente des autres langages, **sert à parcourir une séquence de données :** ⇒ *très utile !*

for element in sequence:

La variable « element » prend successivement chacune des valeurs de la séquence parcourue.

⇒ *« foreach » dans d'autres langages*

Une chaîne de caractères est une séquence de caractères :

Code Python :

```
texte = "Chaîne de caractères"
for lettre in texte:
    print(lettre)
```

⇒ *affiche chaque lettre ligne par ligne (chaque « print » faisant un retour à la ligne)*

<u>Fonctionne également</u> :

```
for lettre in "Chaîne de caractères":
    print(lettre)
```

Le mot-clé « in » peut s'utiliser dans d'autres conditions :

Code Python :

```
texte = "exemple"
phrase = "Ceci est un exemple de phrase."
if texte in phrase:
    print(texte, "est présent dans :", phrase)
```

Pour boucler sur des nombres entiers : ⇒ *range(début, fin+1)*

Code Python :

```
for nombre in range(1, 101):
    print(nombre)
```

<u>Et aussi</u> : for nombre in range(100, 48, -2): ⇒ *nombres de 100 à 50 de -2 en -2*

1.6.3 Mots-clés break et continue

Pour les boucles while / for :

« break » arrête la boucle.

« continue » continue la boucle à l'itération suivante (sans exécuter la suite du bloc).

Code Python :

```
n = 0
while True:     # par défaut : boucle infinie
    n += 1
    if n <= 3:
        continue
    print(n)
    if n == 6:
        break
```

⇒ *affiche uniquement 4, 5, 6*

Eviter d'utiliser « break » et « continue » (comme ici, en général on peut s'en passer).

1.7 Fonctions

1.7.1 Définition

Comme tous les langages, Python permet de définir des **fonctions réalisant un traitement et appelées au besoin** (éventuellement avec paramètre(s)).

Si nécessaire, celles-ci **peuvent également renvoyer / retourner un résultat** à l'appelant.

⇒ *modularité : boîtes noires encapsulant les traitements récurrents*

(comme les fonctions intégrées type, print, input,...)

Syntaxe générale de définition (création) d'une fonction :

def nom_fonction(parametre1, parametre2,…): ⇒ *sans paramètres : def nom_fonction():*
  # bloc d'instructions (avec indentations)

⇒ *mêmes règles de nommage que les variables*
⇒ *ne pas reprendre le nom d'une variable et vice-versa, l'une redéfinirait (écraserait) l'autre !*

1.7.2 Sans paramètres

Code Python :

```
# Définition :

def boucler():
    nombre = 1
    while nombre <= 100:
        print(nombre)
        nombre += 1

# Appel :

boucler()
```

Précision :

Les variables définies dans la fonction sont locales à cette fonction, elles sont inconnues du programme appelant.

1.7.3 Avec paramètres

Code Python :

```
# Définition :

def boucler(debut, fin, increment):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Appel :

boucler(10, 20, 2)
```

⇒ *affiche les nombres de 10 à 20 de 2 en 2*

Lors de l'appel, les valeurs des paramètres peuvent être transmises "en dur" ou via des variables.

Sans nommer les paramètres (abordé plus loin) : **attention à l'ordre, au type et au nombre des paramètres transmis** par rapport à leur définition dans la fonction.

En Python, **la définition des paramètres dans « def » doit uniquement préciser leur nom sans type** (mais attention aux types lors de l'appel).

Précision :

Python 3.5 a introduit le « typing » (annotation des types dans « def »), par exemple :

```
def nom_fonction(parametre1: str, parametre2: float) -> str:
```

⇒ *uniquement pour la lisibilité des programmes (aucuns contrôles)*

Dans certains langages, il y a également la notion de passage de paramètres par valeur ou par référence (adresse) :

Les paramètres sont toujours passés par référence en Python.

Par contre, il y a des paramètres « muables / mutables » (dont on peut modifier le contenu ou la valeur ; les listes et les dictionnaires par exemple) **et des paramètres « immuables / immutables »** (dont le contenu ou la valeur est fixe / statique ; les nombres, les booléens et les chaînes de caractères par exemple ⇒ *équivalent à un passage par valeur*).

1.7.4 Valeur par défaut des paramètres

Dans « def », on peut également définir une valeur par défaut pour les paramètres.
⇒ *idéalement : définir une valeur par défaut pour tous les paramètres (voir paramètres nommés)*

Si un paramètre n'est pas précisé lors de l'appel, il prend sa valeur par défaut :

Code Python :

```
# Définition :

def boucler(fin, debut=1, increment=1):     # paramètres par défaut en dernier !
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Appel normal :

boucler(20, 10, 2)     # de 10 à 20 de 2 en 2

# Appel avec paramètres par défaut (pour debut et increment) :

boucler(10)     # de 1 à 10 de 1 en 1
```

1.7.5 Appels avec paramètres nommés

Lors de l'appel d'une fonction, on peut nommer les paramètres (pratique quand il y a des paramètres avec valeur par défaut) :

Interpréteur Python :

```
>>> def test(p1=1, p2=2, p3=3):
...     print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test()
p1 = 1 p2 = 2 p3 = 3
>>> test(10)
p1 = 10 p2 = 2 p3 = 3
>>> test(p2=20)
p1 = 1 p2 = 20 p3 = 3
>>> test(p2=20, p3=30)
p1 = 1 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, p1=10)
p1 = 10 p2 = 20 p3 = 30
```

<u>Mais aussi (avec paramètres sans valeur par défaut)</u> :

Interpréteur Python :

```
>>> def test(p1, p2, p3):
...         print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test(10, p3=30, p2=20)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, p1=10)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p3=30, 10)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Interpréteur Python :

```
>>> def test(p1, p2, p3=3):
...         print("p1 =", p1, "p2 =", p2, "p3 =", p3)
...
>>> test(10, p3=30, p2=20)
p1 = 10 p2 = 20 p3 = 30
>>> test(p2=20, p1=10)
p1 = 10 p2 = 20 p3 = 3
```

Rappel : les valeurs transmises en paramètres peuvent être des variables.

1.7.6 Signature

La signature d'une fonction **c'est ce qui permet de l'identifier**.

Dans certains langages, c'est son nom et le type de ses paramètres.

En Python, les paramètres d'une fonction n'ayant pas de définition de type, **sa signature n'est que son nom**.

Dans le même programme, on peut redéfinir une fonction déjà définie auparavant, mais dans ce cas **c'est la dernière définition qui est prise en compte (la suivante écrase la précédente)**.

En Python, il n'y a donc pas de surcharges de fonctions (plusieurs fonctions du même nom avec paramètres différents).

⇒ *idéalement : définir les fonctions en début de programme (en évitant de les redéfinir ensuite)*

1.7.7 Renvoi de valeurs avec return

Une fonction peut également **renvoyer / retourner des valeurs à l'appelant grâce à l'instruction « return »** ; soit à une variable (stockage), soit à une autre fonction (utilisation).

En fonction des cas de traitement, **il peut y avoir plusieurs renvois / returns dans la même fonction**. Mais un renvoi / return provoque toujours la sortie (fin) de la fonction.

Interpréteur Python :

```
>>> def montant_ttc(montant_ht, taux_tva):
...         return montant_ht * (1 + taux_tva / 100)
...
>>> montant_facture = montant_ttc(10, 20)
>>> print(montant_facture)
12.0
>>> print(montant_ttc(10, 20))
12.0
```

On peut même retourner plusieurs valeurs en les séparant de virgules, qu'on peut récupérer dans des variables également séparées de virgules (abordé plus loin).

1.7.8 Portée des variables

Les variables extérieures qui ne sont que référencées (lues) à l'intérieur d'une fonction sont implicitement globales. Une variable qui se voit attribuer une valeur n'importe où dans le corps d'une fonction est par défaut locale, sauf si elle est déclarée auparavant avec « global » :

```
>>> variable = 10
>>> def test():
...         print(variable)     # uniquement lue : possède la valeur extérieure
>>> test()
10
>>> def test():
...         variable = 20     # définie en local : différente de la variable extérieure
>>> test()
>>> variable
10
>>> def test():
...         global variable     # définie en global : correspond à la variable extérieure
...         variable = 20
>>> test()
>>> variable
20
```

1.7.9 Docstrings

Une docstring est l'**explication (document d'aide) qu'on peut donner à une fonction** (ou un module, une classe, une méthode,…).

Elle **s’écrit entre triples guillemets** (ou triples apostrophes) **juste en-dessous de la ligne de définition**, tout en respectant l'indentation.

⇒ *avant la première ligne de code, après c'est un simple commentaire*

Elle **peut s'écrire sur une ou plusieurs lignes** (accepte les retours à la ligne).

On peut **ensuite l'afficher à l'aide de la fonction « help » :**

Interpréteur Python :

```
>>> def test():
...         """Fonction test"""
...         print("test")
...
>>> help(test)
```

La fonction « help » est **aussi utilisable pour tous les autres objets Python** (pensez-y).

1.7.10 Fonctions lambda

Une fonction lambda est une **fonction limitée à une seule instruction** (utile pour ce genre de cas simple).

Syntaxe générale de définition (création) d'une fonction lambda :

nom_fonction = lambda parametre1, parametre2,...: instruction de retour

Interpréteur Python :

```
>>> montant_ttc = lambda montant_ht, taux_tva: montant_ht * (1 + taux_tva / 100)
>>> montant_facture = montant_ttc(10, 20)
>>> print(montant_facture)
12.0
>>> print(montant_ttc(10, 20))
12.0
```

Attention aux « : ».

Les paramètres avec valeur par défaut et les paramètres nommés sont également possibles.

1.8 Modules

1.8.1 Définition

L'interpréteur Python intègre déjà un certain nombre de fonctions (et classes).

Mais **de nombreuses autres existent disponibles à travers des modules**, déjà fournis avec l'installation de Python ou téléchargeables en complément.

Un module est un **fichier contenant du code** (potentiellement n'importe quel code Python), généralement des **variables, fonctions ou classes liées par un sujet commun**.

Pour pouvoir utiliser les fonctionnalités d'un module, c'est-à-dire ses variables, fonctions, classes, **il suffit de l'importer auparavant** (dans l'interpréteur ou le programme concerné).

1.8.2 Importation classique

Pour importer un module (par exemple « math ») : **import math** ⇒ *aussi : import sys, os*

Puis, **pour utiliser une variable ou appeler une fonction du module (« math ») :**

Interpréteur Python :

```
>>> math.pi
3.141592653589793
>>> print(math.pi)
3.141592653589793
>>> math.sqrt(25)
5.0
>>> print(math.sqrt(25))
5.0
```

⇒ *on préfixe par le nom du module*

Pour connaître la liste des fonctionnalités du module (« math ») : **help(math)**

Pour lister uniquement le nom des variables et des fonctions du module (« math ») : **dir(math)**

Pour afficher l'aide d'une fonction du module (« math.sqrt ») : **help(math.sqrt)**

Attention : « math.sqrt » renvoie la référence de la fonction et « math.sqrt() » appelle et renvoie le résultat de la fonction.

1.8.3 Espaces de noms

Les variables et les fonctions (éventuellement les classes) **préfixées par le nom du module** (« math. ») **sont dans l'espace de noms du module** (« math »).

Ce qui évite d'avoir des conflits de noms entre ceux du module, ceux des autres modules et ceux du programme principal.

⇒ *chacun peut avoir une variable « pi » et une fonction « sqrt » sans conflits de noms*

1.8.4 Importation avec espace de noms personnalisé (alias)

On peut **personnaliser l'espace de noms d'un module (« math » en « mathematiques ») :**

Interpréteur Python :

```
>>> import math as mathematiques
>>> mathematiques.sqrt(25)
5.0
```

1.8.5 Importations spécifiques

On peut **importer une seule variable ou une seule fonction d'un module** (toujours « math ») :

Interpréteur Python :

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import sqrt
>>> sqrt(25)
5.0
```

⇒ *on ne préfixe pas par le nom du module (donc dans l'espace de noms du programme principal)*
⇒ *mais risques de conflits de noms (le dernier défini écrase le précédent) !*

De cette façon, on peut aussi **importer toutes les fonctionnalités du module (avec « * ») :**

Interpréteur Python :

```
>>> from math import *
>>> sqrt(25)
5.0
```

1.8.6 Création

Rappel :

Un module est un **fichier contenant du code** (potentiellement n'importe quel code Python), généralement des **variables, fonctions ou classes liées par un sujet commun**.

Exemple de création et d'utilisation d'un nouveau module :

1) **Créer le fichier du module « boucler.py » (contenant la fonction « fonc_boucler ») :**

Code Python :

```
"""Module boucler"""

def fonc_boucler(fin, debut=1, increment=1):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment
```

⇒ *module incluant une docstring (en 1ère ligne et entre triples guillemets) pour « help(boucler) »*

2) **Créer le fichier du programme « testboucler.py » (important le module « boucler ») :**

⇒ *dans le même répertoire que le fichier du module « boucler.py »*

Code Python :

```
from boucler import *

# Appel de la fonction fonc_boucler (du module boucler)
fonc_boucler(20, 10, 2)
```

⇒ *import du module « boucler » : nom de son fichier « boucler.py » sans l'extension « .py »*

<u>Précision</u> :

Au moment de l'import du module, Python lit (ou crée si inexistant / modifié) son **fichier "précompilé" avec extension « .pyc » dans le dossier « __pycache__ »** (générés automatiquement).

⇒ *pour des gains de performance à l'exécution*

Un module étant un fichier de code Python avec variables / fonctions / classes (importées depuis d'autres programmes), **on peut également prévoir leurs tests dans le module lui-même et exécuter ce dernier comme tout programme Python**.

Mais pour éviter l'exécution de ces tests lors de l'import du module, **il faut les conditionner dans le fichier du module par la ligne « if __name__ == "__main__": »**.

⇒ « __name__ » est une variable de l'interpréteur qui vaut « __main__ » quand le fichier du module est directement exécuté (i.e. hors import)

Code Python :

```
"""Module boucler"""

def fonc_boucler(fin, debut=1, increment=1):
    nombre = debut
    while nombre <= fin:
        print(nombre)
        nombre += increment

# Test de la fonction fonc_boucler
if __name__ == "__main__":
    fonc_boucler(10)
```

1.9 Packages

1.9.1 Définition

Un module est un fichier de code Python avec variables / fonctions / classes (importées depuis d'autres programmes).

Un package regroupe des modules ou même d'autres packages.

Concrètement, c'est un **dossier contenant des fichiers de modules éventuellement organisés en sous-dossiers** correspondant à des "sous-packages".

Au final, **un package est une bibliothèque de fonctionnalités distribuables**. ⇒ *modularité*

On peut ensuite **importer un package ou l'un de ses "sous-packages", ou encore seulement l'un de ses modules / variables / fonctions / classes**.

On accède aux packages / "sous-packages" / modules en précisant leur chemin avec séparateurs « . » dans la hiérarchie des dossiers / sous-dossiers / fichiers de modules.

⇒ *hiérarchie d'espaces de noms évitant les conflits de noms*

1.9.2 Création et importations

Exemple d'une hiérarchie de packages (issu de https://docs.python.org) :

```
sound/                      Top-level package
    __init__.py             Initialize the sound package
    effects/                Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/                Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Chaque dossier de package doit contenir un fichier « __init__.py » pouvant être vide ou contenir un code exécuté au moment de l'import du package (initialisations de variables, importations d'autres packages,…).

Dans un programme, les imports se font par défaut depuis le répertoire courant du programme (la liste « sys.path » du module « sys » permet de définir le "path" des chemins parcourus).

<u>Différentes syntaxes d'imports existent</u> :

Créer cet échantillon de packages (fichiers « __init__.py » vides) :

```
sound/
    __init__.py
    effects/
        __init__.py
        echo.py
```

Avec le module « echo.py » :

Code Python :

```
def echofilter():
    print("Fonction echofilter")
```

Puis tester ces **exemples d'imports et d'utilisations de fonctions :**

import sound.effects.echo ⇒ *import d'un module (« echo »)*
sound.effects.echo.echofilter()

from sound.effects import echo ⇒ *import d'un module (« echo »)*
echo.echofilter()

from sound.effects.echo import echofilter ⇒ *import d'une fonction (« echofilter »)*
echofilter()

Eviter d'utiliser « from package import * » : importe les noms de modules définis dans la liste « __all__ » initialisable dans « __init__.py » du package (mais lourd et ambigu).

from sound.effects import *
echo.echofilter()

⇒ *fonctionne uniquement avec « __all__ = ["echo"] » dans « __init__.py » du package « effects »*

1.10 Exceptions

1.10.1 Définition

Une exception est une **interruption du programme provoquée par une erreur** (mais pas que).

Quand une erreur se produit (syntaxe, opération,...), Python lève une exception :

Interpréteur Python :

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

En précisant :

- Les fichier / ligne / module concernés
- Le type d'exception (différents types existent) : « ValueError »
- Le message de l'erreur en question : « invalid literal for int() with base 10: 'texte' »

En Python, dès qu'une erreur / exception se produit le traitement s'arrête.

Mais on peut intercepter une exception : essayer un bloc de code et continuer s'il n'y a pas d'erreur ou exécuter un bloc particulier en cas d'erreur.

1.10.2 Interceptions de base

Syntaxe minimale de la structure « try, except » :

try:
 # bloc à essayer
except:
 # bloc exécuté en cas d'erreur

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except:
    print("Erreur de calcul.")
```

L'inconvénient de cette syntaxe minimale est que le traitement exécuté en cas d'erreur est le même quel que soit le type d'exception.

Mais on peut isoler et différencier les types d'exceptions possibles, avec la syntaxe :

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError:
    print("Montant HT ou taux de TVA non définis.")
except TypeError:
    print("Montant HT ou taux de TVA non numériques.")
```

Et aussi récupérer le message d'une exception dans une variable grâce au mot-clé « as » :

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError as message_exception:
    print(message_exception)
except TypeError as message_exception:
    print(message_exception)
```

Des exceptions ne sont pas des erreurs : CTRL+C (arrêt de programme) est une exception.

⇒ *donc éviter d'utiliser la syntaxe minimale (i.e. « except » sans préciser de types d'exceptions)*
⇒ *« except » sans type d'exception est surtout utile pour les autres cas (« except » en dernier)*
⇒ *l'interpréteur de commandes Python est aussi pratique pour tester les types d'exceptions*

1.10.3 Mots-clés else et finally

On peut également ajouter à la structure « try, except » les mots-clés « else » et « finally » :

« else » permet d'exécuter un traitement si aucune erreur ne survient dans le bloc « try ».

⇒ *par exemple pour mettre en forme le résultat*

« finally » permet d'exécuter un traitement après le bloc « try » quel que soit son résultat (avec ou sans erreur).

⇒ *« finally » est exécuté dans tous les cas, même avec un « return » dans un « except » !*

(ces deux mots-clés peuvent s'utiliser indépendamment)

Code Python :

```
try:
    mt_ttc = mt_ht * (1 + taux_tva / 100)     # erreur si variables inexistantes ou non numériques
except NameError:
    print("Montant HT ou taux de TVA non définis.")
except TypeError:
    print("Montant HT ou taux de TVA non numériques.")
else:
    print("Le montant TTC est :", mt_ttc)
finally:
    print("Fin de traitement du montant TTC.")
```

1.10.4 Mot-clé pass

On utilise « pass » **dans les blocs d'instructions sans code** (quels qu'ils soient) :

```
try:
    # bloc à tester
except type_exception:
    pass     # pas de traitement en cas d'erreur (rien ne se passe)
```

1.10.5 Assertions

Une assertion permet de **tester une condition avec le mot-clé « assert »**.

Si la condition est fausse (False), une exception de type « AssertionError » est levée :

Interpréteur Python :

```
>>> nombre = 10
>>> assert nombre == 10
>>> assert nombre == 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Utile dans les structures « try, except » :

Code Python :

```
try:
    assert mt_ht >= 0     # exception AssertionError si condition fausse
except AssertionError:
    print("Le montant HT est négatif !")
```

1.10.6 Lever une exception

On peut soi-même lever une exception **grâce au mot-clé « raise »**.

Syntaxe générale d'une levée d'exception :

raise type_exception("message affiché")

Interpréteur Python :

```
>>> raise ValueError("erreur de valeur")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: erreur de valeur
>>>
>>> raise TypeError("erreur de type")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: erreur de type
```

⇒ *peut être utile dans certains traitements*

2 Objets essentiels

2.1 Notion d'objet

Python est un **langage de programmation orienté objet**.

Un objet est avant tout une structure de données (en mémoire) :

La principale différence vient du fait que **l'objet regroupe les données** (variables sous forme d'attributs) **et les moyens de traiter ces données** (fonctions sous forme de méthodes).

Variables, fonctions,… **tout est objet en Python !**

Enfin, **un objet est créé (instancié) à partir d'une classe** qui est un "moule à objets" (modèle).

2.2 Chaînes de caractères

2.2.1 Classe str

Les chaînes de caractères sont des objets créés (instanciés) à partir de la classe « str ».

Quand on crée une variable contenant du texte c'est un objet de la classe « str » :

Interpréteur Python :

```
>>> texte = "Chaîne de caractères"
>>> type(texte)
<class 'str'>
```

Comme toute classe, on peut aussi instancier un objet "chaîne de caractères" avec la syntaxe :

Interpréteur Python :

```
>>> texte1 = str()     # créé vide, équivalent à : texte1 = ""
>>> type(texte1)
<class 'str'>
>>> print(texte1)

>>> texte2 = str("Chaîne de caractères")     # équivalent à : texte2 = "Chaîne de caractères"
>>> type(texte2)
<class 'str'>
>>> print(texte2)
Chaîne de caractères
```

La classe « str » dispose de nombreuses méthodes ("fonctions") accessibles par la syntaxe :

nom_objet.nom_methode(arguments) ⇒ *sans arguments : nom_objet.nom_methode()*

⇒ *fonctionnalités manipulant les chaînes de caractères*

2.2.2 Méthodes de mise en forme

Principales méthodes de mise en forme des chaînes de caractères :

lower : en minuscules
upper : en majuscules
capitalize : première lettre en majuscule, les autres en minuscules
strip : sans espaces ni « \n » au début et à la fin
center : centrée par rapport à une taille de caractères (passée entre parenthèses)
…

Liste complète des méthodes de la classe « str » : **help(str)** ⇒ *aussi : dir(str)*

Aide d'une méthode (exemple) : **help(str.lower)**

Interpréteur Python :

```
>>> texte = "EXEMPLE DE TEXTE"
>>> texte.lower()
'exemple de texte'
>>> texte = "exemple de texte"
>>> texte.upper()
'EXEMPLE DE TEXTE'
>>> texte.capitalize()
'Exemple de texte'
>>> texte = "    exemple de texte  "
>>> texte.strip()
'exemple de texte'
>>> texte.strip().center(30)
'       exemple de texte       '
```

Pour la dernière :

« texte.strip() » renvoie une chaîne de caractères, donc elle-même un objet de la classe « str » pour lequel on appelle à son tour la méthode « center(30) ».

Précision :

Ces méthodes renvoient une chaîne résultat mais **ne modifient pas la chaîne d'origine**.

2.2.3 Formatages

La méthode « format », également de la classe « str », **permet de formater une chaîne de caractères** à l'aide de plusieurs syntaxes.

Avec indices des paramètres à insérer : ⇒ *ordre indifférent*

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> print("{1} a {0} ans.".format(age, nom))    # directement affiché
Tom a 30 ans.
>>> chaine = "{1} a {0} ans.".format(age, nom)    # stocké dans une variable
>>> chaine
'Tom a 30 ans.'
```

⇒ *les indices commencent à 0 (zéro) : {0} correspond à « age » et {1} correspond à « nom »*

Sans indices des paramètres à insérer : ⇒ *ordre à respecter*

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> print("{} a {} ans.".format(nom.upper(), age))
TOM a 30 ans.
```

Avec noms et valeurs des paramètres à insérer : ⇒ *ordre indifférent et plus intuitif*

Interpréteur Python :

```
>>> print("{nom} a {age} ans.".format(age=30, nom="Tom"))
Tom a 30 ans.
>>> autrenom = "Bill"
>>> autreage = 40
>>> print("{nom} a {age} ans.".format(age=autreage, nom=autrenom.upper()))
BILL a 40 ans.
```

⇒ *les valeurs peuvent provenir d'autres variables ou de fonctions / méthodes*

On peut combiner indices et noms : print("{nom} a {0} ans.".format("30", nom="Tom"))

Il existe aussi de **nombreuses syntaxes de mise en forme des valeurs :**

Interpréteur Python :

```
>>> nb = 123.456789
>>> print("Le nombre vaut : {nombre:.3f}".format(nombre=nb))
Le nombre vaut : 123.457
>>> nb = 123
>>> print("Le nombre vaut : {nombre:10d}".format(nombre=nb))
Le nombre vaut :        123
>>> print("Le nombre vaut : {nombre:0>10d}".format(nombre=nb))
Le nombre vaut : 0000000123
>>> nom = "Tom"
>>> print("{0:10} complété par des espaces (sur 10 car.).".format(nom))
Tom        complété par des espaces (sur 10 car.).
```

Python 3.6 a également introduit les **chaînes littérales formatées ou « f-chaînes » (f-strings) :**

```
nom = "Tom" ; age = 30 ; chaine = f"{nom.upper()} a {age} ans." ; print(chaine)
nombre = 123.456789 ; print(f"Le nombre vaut : {nombre:.3f}")
nombre = 123 ; print(f"Le nombre vaut : {nombre:10d}")
nombre = 123 ; print(f"Le nombre vaut : {nombre:0>10d}")
nom = "Tom" ; print(f"{nom:10} complété par des espaces (sur 10 car.).")
```

2.2.4 Concaténation classique

Le signe **« + » sert également à concaténer des chaînes de caractères :**

Interpréteur Python :

```
>>> nom = "Tom"
>>> age = 30
>>> chaine = nom + " a " + age + " ans."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
>>> chaine = nom + " a " + str(age) + " ans."
>>> print(chaine)
Tom a 30 ans.
```

Les autres types doivent être convertis en chaîne de caractères avec la méthode « str(…) ».

⇒ *Python est un langage fortement typé*

2.2.5 Parcours de chaînes

Rappel : une chaîne de caractères est une séquence de caractères (donc aussi de chaînes).

On peut directement la parcourir **avec une boucle « for » :**

Code Python :

```
chaine = "Chaîne de caractères"
for caractere in chaine:
    print(caractere)
```

On peut également **accéder à chaque caractère d'une chaîne par indice (commençant à 0) :**

Interpréteur Python :

```
>>> chaine = "Chaîne de caractères"
>>> chaine[0]     # premier caractère
'C'
>>> chaine[3]     # quatrième caractère
'î'
>>> chaine[-1]     # dernier caractère (accès par la fin avec indice négatif)
's'
```

Pour obtenir le nombre de caractères d'une chaîne (longueur) : **len(chaine)**

⇒ *ce n'est pas une méthode de la classe « str », mais une fonction pour différents types d'objets*

On peut donc aussi parcourir une chaîne de caractères **avec une boucle « while » :**

Code Python :

```
chaine = "Chaîne de caractères"
indice = 0
while indice < len(chaine):
    print(chaine[indice])
    indice += 1
```

⇒ *accéder à un indice inexistant lève une exception de type « IndexError »*

Mais on ne peut pas modifier un caractère de chaîne par indice : ⇒ *objet « immutable »*

Interpréteur Python :

```
>>> chaine[5] = "s"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

2.2.6 Découpage de chaînes

Le découpage / slicing des chaînes de caractères **se fait par une autre syntaxe entre crochets :**

[début:fin] ou **[début:fin:pas]** ⇒ *début commence à 0 et fin correspond au dernier élément*
⇒ *valeurs négatives et variables / fonctions possibles*

Interpréteur Python :

```
>>> chaine = "Chaîne de caractères"
>>> chaine[:3]     # ou : chaine[0:3]
'Cha'
>>> chaine[3:]     # ou : chaine[3:len(chaine)]
'îne de caractères'
>>> chaine[12:16]
'ract'
>>> chaine[-8:-4]     # par la fin
'ract'
>>> chaine[::2]     # 1 sur 2
'Can ecrcèe'
>>> chaine[4:17:3]     # avec 1 sur 3
'ndcaè'
>>> chaine[::-1]
'serètcarac ed enîahC'
```

2.2.7 Compter, rechercher, remplacer

count : compte le nombre d'occurrences de la sous-chaîne dans la chaîne

find : recherche la première occurrence de la sous-chaîne dans la chaîne

replace : remplace les occurrences de la sous-chaîne dans la chaîne

Interpréteur Python :

```
>>> chaine = "Exemple de texte"
>>> chaine.count("te")
2
>>> chaine.find("te")     # commence à 0, -1 si non trouvé
11
>>> chaine.replace("te", "pa")
'Exemple de paxpa'
```

⇒ *ne modifie pas la chaîne d'origine*

2.3 Listes et tuples

2.3.1 Classe list

Les listes sont aussi **des séquences. Des objets capables de contenir d'autres objets** de n'importe quel type (même d'autres listes) et sans limite de taille.

Elles sont **créées (instanciées) à partir de la classe « list » :**

Interpréteur Python :

```
>>> liste = list()     # créée vide, équivalent à : liste = []
>>> type(liste)
<class 'list'>
>>> liste
[]
>>> liste = ["chaîne", 123, 9.99, []]     # créée avec des objets (de tous types)
>>> print(liste)
['chaîne', 123, 9.99, []]
```

⇒ *les objets d'une liste doivent être entre crochets « [ ] » et séparés de virgules « , »*

2.3.2 Accès et modification des objets

On accède aux objets d'une liste avec leur indice entre crochets (commençant à 0) :

Interpréteur Python :

```
>>> liste = ["chaîne", 123, 9.99, []]     # créée avec des objets (de tous types)
>>> print(liste)
['chaîne', 123, 9.99, []]
>>> liste = [321, "texte", 9.99, "a", "b"]     # réaffectée / écrasée (recréée)
>>> print(liste)
[321, 'texte', 9.99, 'a', 'b']
>>> liste[0]
321
>>> liste[3]
'a'
>>> liste[3] = "aaa"
>>> liste
[321, 'texte', 9.99, 'aaa', 'b']
```

Contrairement aux chaînes de caractères, **les listes sont des objets « muables / mutables »**.

2.3.3 Insertion d'objets

Ne fonctionne pas :

Interpréteur Python :

```
>>> liste = ["chaîne", 123]
>>> liste[1]
123
>>> liste[2] = 9.99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Pour ajouter un objet à la fin d'une liste, il faut passer par la méthode « append » :

Interpréteur Python :

```
>>> liste.append(9.99)
>>> liste
['chaîne', 123, 9.99]
```

⇒ *« append » est une méthode de la classe « list » (voir « help(list) »)*

« append » modifie l'objet mais ne renvoie rien. ⇒ *contraire des chaînes de caractères*

Attention au comportement des méthodes des classes :

Interpréteur Python :

```
>>> chaine1 = "TEST"
>>> chaine2 = chaine1.lower()     # renvoie un résultat (nouvel objet)
>>> chaine1
'TEST'
>>> chaine2
'test'
>>> liste1 = ["TEST"]
>>> liste2 = liste1.append(123)     # ne renvoie rien (modifie l'objet lui-même)
>>> liste1
['TEST', 123]
>>> liste2
>>> print(liste2)
None
```

« None » correspond à l'objet vide (rien) en Python.

Rappel :
Les chaînes de caractères sont des objets « immutables » et les listes des objets « mutables ».

Pour insérer un objet dans une liste, il faut passer par la méthode « insert » :

Interpréteur Python :

```
>>> liste = ["chaîne", 123, 9.99]
>>> liste.insert(2, "aaa")
>>> liste
['chaîne', 123, 'aaa', 9.99]
```

⇒ *en précisant l'indice concerné (décale la suite)*

2.3.4 Concaténation de listes

Interpréteur Python :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = ["a", "b", "c"]
>>> liste1 += liste2     # avec la concaténation classique "+" (liste1 = liste1 + liste2)
>>> print(liste1)
[1, 2, 3, 'a', 'b', 'c']
```

Interpréteur Python :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = ["a", "b", "c"]
>>> liste1.extend(liste2)     # avec la méthode "extend" de la classe "list"
>>> print(liste1)
[1, 2, 3, 'a', 'b', 'c']
```

2.3.5 Suppression d'objets

Le mot-clé « del » permet de supprimer un objet, variable ou autre, exemple : **del variable**

⇒ *ce n'est pas une méthode de la classe « list », mais un mot-clé général (interne à l'interpréteur)*

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> liste
['chaîne', 123, 'aaa', 9.99]
>>> del liste[2]     # on précise l'indice concerné
>>> liste
['chaîne', 123, 9.99]
```

La méthode « remove » de la classe « list » permet aussi de **supprimer un objet d'une liste mais en précisant sa valeur :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99, 123]
>>> liste
['chaîne', 123, 'aaa', 9.99, 123]
>>> liste.remove(123)
>>> liste
['chaîne', 'aaa', 9.99, 123]
>>> liste.remove(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

⇒ *supprime uniquement la première occurrence trouvée*

2.3.6 Parcours de listes

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> for element in liste:
...     print(element)
...
chaîne
123
aaa
9.99
>>> indice = 0
>>> while indice < len(liste):
...     print(liste[indice])
...     indice += 1
...
chaîne
123
aaa
9.99
```

⇒ *équivalent aux chaînes de caractères ; « len » (longueur) fonctionne également sur les listes*

Dans une boucle « for / in », **« enumerate »** (c'est une classe) **prend en paramètre la liste parcourue et retourne pour chaque élément un tuple sous la forme « (indice, objet) » :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> for element in enumerate(liste):
...         print(element)
...
(0, 'chaîne')
(1, 123)
(2, 'aaa')
(3, 9.99)
```

⇒ *les tuples sont détaillés plus loin*

Leurs valeurs peuvent être récupérées dans des variables : ⇒ *parenthèses inutiles*

Code Python :

```
for indice, objet in enumerate(liste):
    print("Pour l'indice {} : {}".format(indice, objet))
```

Autres exemples de parcours de listes (qui incluent d'autres listes) :

Interpréteur Python :

```
>>> panier = [
... ["Pommes", 6, 1.50],
... ["Oeufs", 12, 3.00],
... ["Tomates", 10, 3.50]
... ]
>>> for produit, quantite, prix in panier:
...         print(f"{quantite} {produit} pour {prix:.2f} euros.")
...
6 Pommes pour 1.50 euros.
12 Oeufs pour 3.00 euros.
10 Tomates pour 3.50 euros.
```

Interpréteur Python :

```
>>> for element in enumerate(panier):
...         print(element)
...
(0, ['Pommes', 6, 1.5])
(1, ['Oeufs', 12, 3.0])
(2, ['Tomates', 10, 3.5])
```

2.3.7 Tuples

Les tuples sont des **listes (séquences) « immuables / immutables », c'est-à-dire non modifiables**. Ils sont créés (instanciés) à partir de la **classe « tuple » :**

Interpréteur Python :

```
>>> liste_tuple = tuple()     # créé vide (en soi inutile...), équivalent à : liste_tuple = ()
>>> type(liste_tuple)
<class 'tuple'>
>>> liste_tuple
()
>>> liste_tuple = ("chaîne", 123, 9.99)     # (re)créé avec des objets
>>> liste_tuple = (321, "texte", 9.99, "a", "b")     # réaffecté (recréé), non modifié !
>>> print(liste_tuple)
(321, 'texte', 9.99, 'a', 'b')
>>> liste_tuple[0]     # accès avec indice entre crochets (comme une liste)
321
>>> liste_tuple[3] = "aaa"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

⇒ *les objets d'un tuple doivent être entre parenthèses « ( ) » et séparés de virgules « , »*

Une fois créé, **on ne peut plus y ajouter d'objets ou en retirer, ni en modifier** (uniquement le réaffecter / recréer intégralement).

Pour le reste, un tuple s'utilise comme une liste (accès, parcours,…).

Aide : **help(tuple)**
Nombre d'éléments : **len(liste_tuple)**
Concaténation de tuples : **tuple1 += tuple2** ⇒ *ou : tuple1 = tuple1 + tuple2*
Suppression d'un tuple : **del liste_tuple** ⇒ *mais pas d'un élément !*

Interpréteur Python :

```
>>> liste_tuple = (321, "texte", 9.99, "a", "b")
>>> liste_tuple
(321, 'texte', 9.99, 'a', 'b')
>>> liste_tuple = list(liste_tuple)     # conversion en liste
>>> liste_tuple
[321, 'texte', 9.99, 'a', 'b']
>>> liste_tuple = tuple(liste_tuple)     # conversion en tuple
>>> liste_tuple
(321, 'texte', 9.99, 'a', 'b')
```

Les affectations multiples passent par des tuples :

Interpréteur Python :

```
>>> a, b, c = 3, 2, 1
>>> print(c, b, a)
1 2 3
>>> (a, b, c) = (3, 2, 1)
>>> print(c, b, a)
1 2 3
```

⇒ *les parenthèses sont optionnelles (et ensuite les variables sont bien modifiables)*

Interpréteur Python :

```
>>> def fonction():
...         a = 10 ; b = 20 ; c = 30
...         return a, b, c
...
>>> var1, var2, var3 = fonction()
>>> print(var1, var2, var3)
10 20 30
>>> retour = fonction()
>>> print(retour)
(10, 20, 30)
```

2.3.8 Passage d'une chaîne à une liste et inversement

Passage d'une chaîne à une liste avec la méthode « split » de la classe « str » :

Interpréteur Python :

```
>>> chaine = "Sébastien;MEYER;10, Avenue des Vosges;67000;STRASBOURG"
>>> chaine.split(";")
['Sébastien', 'MEYER', '10, Avenue des Vosges', '67000', 'STRASBOURG']
```

⇒ *« split » prend en paramètre le séparateur (par défaut : espaces, tabulations et sauts de ligne)*

Passage d'une liste à une chaîne avec la méthode « join » de la classe « str » :

Interpréteur Python :

```
>>> liste = ['Sébastien', 'MEYER', '10, Avenue des Vosges', '67000', 'STRASBOURG']
>>> ";".join(liste)
'Sébastien;MEYER;10, Avenue des Vosges;67000;STRASBOURG'
```

⇒ *« join » est appliquée à la chaîne servant de séparateur (peut être une variable)*

Et aussi :

Interpréteur Python :

```
>>> chaine = "a---b---c"
>>> var1, var2, var3 = chaine.split("---")
>>> print(var1, var2, var3)
a b c
>>> chaine = "***".join([var1, var2, var3])
>>> print(chaine)
a***b***c
```

2.3.9 Fonction avec un nombre indéterminé de paramètres

Comme la fonction « print ».

Pour définir une fonction avec un nombre indéterminé de paramètres, il suffit de **prévoir un paramètre précédé d'une étoile (*) qui recevra dans un tuple la liste des paramètres transmis :**

def nom_fonction(*parametres):

Interpréteur Python :

```
>>> def fonction(*parametres):
...     print("Paramètres transmis :", parametres)
...
>>> fonction()
Paramètres transmis : ()
>>> fonction(123)
Paramètres transmis : (123,)
>>> fonction(123, "aaa", 9.99)
Paramètres transmis : (123, 'aaa', 9.99)
>>> variable = 456
>>> fonction(variable, variable+1000, [1, 2, "a", "b"])
Paramètres transmis : (456, 1456, [1, 2, 'a', 'b'])
```

Les appels avec paramètres nommés ne sont pas gérés (abordés plus loin avec les dictionnaires).

On peut définir une **fonction avec paramètres obligatoires (classiques) suivis de paramètres variables (<u>toujours après</u>) :**

def nom_fonction(parametre1, parametre2,… , *parametres):

Interpréteur Python :

```
>>> def fonction(parametre1, parametre2, *parametres):
...     print("Paramètres transmis : {}, {} et {}".format(parametre1, parametre2, parametres))
...
>>> fonction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fonction() missing 2 required positional arguments: 'parametre1' and 'parametre2'
>>> fonction(123, "aaa")
Paramètres transmis : 123, aaa et ()
>>> fonction(123, "aaa", 9.99, "etc")
Paramètres transmis : 123, aaa et (9.99, 'etc')
```

Et si besoin, **les paramètres avec valeur par défaut doivent être définis en dernier :**

Interpréteur Python :

```
>>> def fonction(parametre1, parametre2, *parametres, defaut1="valeur1", defaut2="valeur2"):
...     print("Paramètres transmis : {}, {} et {}".format(parametre1, parametre2, parametres))
...     print("Et paramètres par défaut :", defaut1, "et", defaut2)
...
>>> fonction(123, "aaa", 9.99, "etc")
Paramètres transmis : 123, aaa et (9.99, 'etc')
Et paramètres par défaut : valeur1 et valeur2
```

2.3.10 Transformer une liste ou un tuple en paramètres de fonction

Lors d'un appel de fonction (ou de méthode), on peut **transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre**, et non en tant qu'objet liste (ou tuple), **en précédant son nom également d'une étoile (*) :**

Interpréteur Python :

```
>>> liste = ["chaîne", 123, "aaa", 9.99]
>>> print(liste)     # appel classique : un seul paramètre transmis (l'objet liste)
['chaîne', 123, 'aaa', 9.99]
>>> print(*liste)     # appel en transmettant les éléments de l'objet liste paramètre par paramètre
chaîne 123 aaa 9.99
```

⇒ *utile dans certains cas de traitements*

Conclusion :

Dans la définition d'une fonction : l'étoile (*) permet de recevoir des paramètres variables dans un seul paramètre sous forme de tuple.

Lors de l'appel d'une fonction, c'est l'inverse : l'étoile (*) permet de transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre.

2.3.11 Compréhensions de liste

Une compréhension de liste (« list comprehension » en anglais) **permet de générer une liste à partir du parcours d'une autre liste tout en la modifiant et/ou la filtrant**. ⇒ *très puissant*

Sa syntaxe est donc **entre crochets « [ ] »**.

Parcours simple avec modification :

Interpréteur Python :

```
>>> nombres = [-9, -2, 0, 3, 5, 7]
>>> [n*10 for n in nombres]
[-90, -20, 0, 30, 50, 70]
```

⇒ *la boucle « for » parcourt la liste « nombres » et pour chaque élément (n) génère le résultat « n*10 » renvoyé en liste*

(peut également parcourir un tuple)

<u>Parcours avec filtrage et modification</u> :

Interpréteur Python :

```
>>> nombres = [-9, -2, 0, 3, 5, 7]
>>> [n*10 for n in nombres if n >= 0]
[0, 30, 50, 70]
```

⇒ *dans la boucle « for », le « if » permet de conditionner (filtrer) les valeurs souhaitées*

(plus simple que de produire le résultat avec des structures classiques imbriquées)

<u>Autre exemple (avec tri d'une liste de tuples)</u> : ⇒ *fonction « sorted » ou méthode liste « sort »*

Interpréteur Python :

```
>>> panier = [("Pommes", 6, 1.50), ("Oeufs", 12, 3.00), ("Tomates", 10, 3.50)]
>>> panier_prix = [(pri, pro, qte) for pro, qte, pri in panier]
>>> panier_prix     # prix en premier pour le tri
[(1.5, 'Pommes', 6), (3.0, 'Oeufs', 12), (3.5, 'Tomates', 10)]
>>> panier = [(pro, qte, pri) for pri, pro, qte in sorted(panier_prix, reverse=True)]
>>> panier     # nouvelle liste triée d'après prix décroissant
[('Tomates', 10, 3.5), ('Oeufs', 12, 3.0), ('Pommes', 6, 1.5)]
```

2.4 Dictionnaires et ensembles

2.4.1 Classe dict

Comme les listes, les dictionnaires sont **des conteneurs. Des objets capables de contenir d'autres objets** de n'importe quel type et sans limite de taille.

Mais contrairement aux listes, ils **ne sont pas ordonnés et pour accéder aux objets on utilise des clés** (chaînes ou autres, voire indices entiers). ⇒ *couples « clés / objets »*

Ils sont **créés (instanciés) à partir de la classe « dict » :**

Interpréteur Python :

```
>>> dictionnaire = dict()     # créé vide, équivalent à : dictionnaire = {}
>>> type(dictionnaire)
<class 'dict'>
>>> dictionnaire
{}
```

Les dictionnaires sont des objets « muables / mutables ».

2.4.2 Insertion, accès et modification des objets

Pour référencer un objet dans un dictionnaire (existant), on précise le nom du dictionnaire avec la clé entre crochets « [ ] » :

Interpréteur Python :

```
>>> dictionnaire["nom"] = "Tom"     # affecte la valeur (objet) "Tom" à la clé "nom"
>>> dictionnaire["age"] = 30
>>> print(dictionnaire["nom"], ",", dictionnaire["age"])
Tom , 30
>>> dictionnaire["age"] = 40     # modifie la clé "age"
>>> dictionnaire["taille"] = 1.75     # ajoute la clé "taille"
>>> dictionnaire
{'nom': 'Tom', 'age': 40, 'taille': 1.75}
>>> dictionnaire = {"nom": "Tom", "age": 30}     # entièrement réaffecté (recréé)
>>> dictionnaire
{'nom': 'Tom', 'age': 30}
```

⇒ *les couples « clés / objets » d'un dictionnaire doivent être entre accolades « { } »*
⇒ *séparés de virgules « , » et écrits sous la forme « clé: objet »*
⇒ *on peut utiliser quasiment tous les types comme clés et tous les types comme objets*

Les clés peuvent être des tuples, par exemple pour une matrice de points binaires :

Interpréteur Python :

```
>>> matrice = {}
>>> matrice[(1, 'a')] = 0
>>> matrice[(1, 'b')] = 1
>>> matrice[(1, 'c')] = 1
>>> matrice[(2, 'a')] = 1
>>> matrice[(2, 'b')] = 0
>>> matrice[(2, 'c')] = 1
>>> matrice[(3, 'a')] = 1
>>> matrice[(3, 'b')] = 0
>>> matrice[(3, 'c')] = 1
>>> print(matrice)
{(1, 'a'): 0, (1, 'b'): 1, (1, 'c'): 1, (2, 'a'): 1, (2, 'b'): 0, (2, 'c'): 1, (3, 'a'): 1, (3, 'b'): 0, (3, 'c'): 1}
```

⇒ *les parenthèses « ( ) » des tuples entre crochets « [ ] » sont optionnelles (mais plus explicites)*

2.4.3 Suppression d'objets

Avec le mot-clé « del » (déjà vu) ou la méthode « pop » de la classe « dict » :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 30, "taille": 1.75, "métier": "Informaticien"}
>>> dictionnaire
{'nom': 'Tom', 'age': 30, 'taille': 1.75, 'métier': 'Informaticien'}
>>> del dictionnaire["taille"]     # on précise la clé
>>> dictionnaire
{'nom': 'Tom', 'age': 30, 'métier': 'Informaticien'}
>>> dictionnaire.pop("métier")     # on précise également la clé
'Informaticien'
>>> dictionnaire
{'nom': 'Tom', 'age': 30}
```

⇒ *la méthode « pop » renvoie la valeur de la clé supprimée (peut être utile)*

2.4.4 Application : dictionnaire de fonctions

Rappel : « print » correspond à la référence de la fonction et « print() » à son appel.

Interpréteur Python :

```
>>> print
<built-in function print>
>>> print()

>>>
```

On peut donc **stocker la référence d'une fonction dans une variable :**

Interpréteur Python :

```
>>> afficher = print
>>> afficher("Un texte")
Un texte
```

⇒ *la variable « afficher » possède la même référence que la fonction « print »*
⇒ *quand on utilise la variable « afficher », elle appelle donc la fonction « print »*

On peut aussi **stocker les références de fonctions dans n'importe quel objet conteneur, listes, dictionnaires, etc :**

Interpréteur Python :

```
>>> def fonction1():
...     print("Appel fonction1")
...
>>> def fonction2():
...     print("Appel fonction2")
...
>>> fonctions = {}
>>> fonctions["f1"] = fonction1
>>> fonctions["f2"] = fonction2
>>> fonctions["f1"]    # fait référence à fonction1
<function fonction1 at 0x7f1f38a063a0>
>>> fonctions["f1"]()    # fait appel à fonction1
Appel fonction1
>>> fonctions["f2"]    # fait référence à fonction2
<function fonction2 at 0x7f1f38a06430>
>>> fonctions["f2"]()    # fait appel à fonction2
Appel fonction2
```

2.4.5 Parcours de dictionnaires

Le parcours classique ne retourne que les clés du dictionnaire :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>> for element in dictionnaire:
...         print(element)
...
nom
age
taille
```

Précision :

Les dictionnaires n'étant pas ordonnés, les clés retournées ne sont pas forcément dans l'ordre de saisie (ou alphanumérique).

⇒ *dépend de la gestion interne des dictionnaires*

La méthode « keys » de la classe « dict » retourne également les clés : ⇒ *à privilégier*

Interpréteur Python :

```
>>> dictionnaire.keys()
dict_keys(['nom', 'age', 'taille'])
>>> for cle in dictionnaire.keys():
...     print(cle)
...
nom
age
taille
```

Et la méthode « values » de la classe « dict » retourne les valeurs (objets) :

Interpréteur Python :

```
>>> dictionnaire.values()
dict_values(['Tom', 40, 1.75])
>>> for valeur in dictionnaire.values():
...     print(valeur)
...
Tom
40
1.75
```

Utilisables également **pour les recherches de clés ou de valeurs / objets :**

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>> cle = "age"
>>> if cle in dictionnaire.keys():
...         print(cle, "est présent.")
...
age est présent.
>>>
>>> valeur = 1.75
>>> if valeur in dictionnaire.values():
...         print(valeur, "est présent.")
...
1.75 est présent.
```

Enfin, **la méthode « items » de la classe « dict » retourne les couples « clés / objets » :**

Interpréteur Python :

```
>>> dictionnaire.items()
dict_items([('nom', 'Tom'), ('age', 40), ('taille', 1.75)])
>>>
>>> for element in dictionnaire.items():
...         print(element)
...
('nom', 'Tom')
('age', 40)
('taille', 1.75)
>>>
>>> for cle, valeur in dictionnaire.items():     # parenthèses inutiles autour de cle, valeur
...         print(cle, "=", valeur)
...
nom = Tom
age = 40
taille = 1.75
```

⇒ *retourne pour chaque élément un tuple sous la forme « (clé, objet) »*

2.4.6 Fonction avec un nombre indéterminé de paramètres nommés

Rappel : dans la définition d'une fonction, l'étoile (*) permet de recevoir des paramètres variables dans un seul paramètre sous forme de tuple, sauf les paramètres nommés.

Deux étoiles (**) permettent de recevoir des paramètres nommés variables dans un seul paramètre sous forme de dictionnaire (sauf les paramètres non nommés) :

Interpréteur Python :

```
>>> def fonction(**parametres_nommes):
...     print("Paramètres nommés transmis :", parametres_nommes)
...
>>> fonction()
Paramètres nommés transmis : {}
>>> fonction(entier=123, chaine="aaa", decimal=9.99)
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
>>> fonction(456, "zzz", entier=123, chaine="aaa", decimal=9.99)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fonction() takes 0 positional arguments but 2 were given
```

Et pour recevoir des paramètres variables non nommés et nommés (attention à l'ordre) :

Interpréteur Python :

```
>>> def fonction(*parametres, **parametres_nommes):     # obligatoirement dans cet ordre
...         print("Paramètres non nommés transmis :", parametres)
...         print("Paramètres nommés transmis :", parametres_nommes)
...
>>> fonction()
Paramètres non nommés transmis : ()
Paramètres nommés transmis : {}
>>> fonction(456, "zzz")
Paramètres non nommés transmis : (456, 'zzz')
Paramètres nommés transmis : {}
>>> fonction(entier=123, chaine="aaa", decimal=9.99)
Paramètres non nommés transmis : ()
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
>>> fonction(456, "zzz", entier=123, chaine="aaa", decimal=9.99)
Paramètres non nommés transmis : (456, 'zzz')
Paramètres nommés transmis : {'entier': 123, 'chaine': 'aaa', 'decimal': 9.99}
```

2.4.7 Transformer un dictionnaire en paramètres nommés de fonction

Rappel : lors de l'appel d'une fonction, l'étoile (*) permet de transmettre les éléments d'une liste (ou d'un tuple) paramètre par paramètre.

Deux étoiles (**) permettent de transmettre les éléments d'un dictionnaire paramètre nommé par paramètre nommé :

Interpréteur Python :

```
>>> dictionnaire = {"sep": " - ", "end": ".\n"}
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", dictionnaire)
dictionnaire en paramètres de fonction {'sep': ' - ', 'end': '.\n'}
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", **dictionnaire)
dictionnaire - en - paramètres - de - fonction.
>>>
>>> # Equivalent à :
>>> print("dictionnaire", "en", "paramètres", "de", "fonction", sep=" - ", end=".\n")
dictionnaire - en - paramètres - de - fonction.
```

⇒ *la fonction « print » dispose des paramètres nommés :*
⇒ *« sep » (séparateur des paramètres affichés) et « end » (fin d'affichage)*

2.4.8 Ensembles (sets)

Cas particulier : ⇒ *voir « help(set) »*

Un ensemble est une **collection d'éléments non ordonnée, sans index (indices ou clés) et qui ne peut pas posséder d'éléments dupliqués** (il supprime automatiquement les doublons).

C'est également un **objet conteneur, de la classe « set », dont les objets sont entre accolades « { } » et séparés de virgules « , » :**

Interpréteur Python :

```
>>> ensemble = {"Tom", "Bill", "Bob", "Tom", "Tom", "Bill"}
>>> ensemble
{'Bob', 'Tom', 'Bill'}
>>> type(ensemble)
<class 'set'>
>>> for objet in ensemble:
...     print(objet)
...
Bob
Tom
Bill
```

Il n'est **pas possible d'accéder aux objets d'un ensemble avec l'opérateur « [ ] »**.
⇒ *mais on peut convertir un ensemble en liste avec « liste = list(ensemble) »*

Attention, **pour créer un ensemble vide, il faut utiliser la fonction « set() » :**

Interpréteur Python :

```
>>> ensemble = {}
>>> type(ensemble)
<class 'dict'>
>>> ensemble = set()
>>> type(ensemble)
<class 'set'>
```

Pour ajouter et supprimer des objets dans un ensemble (existant) : ⇒ *objet « mutable »*

Interpréteur Python :

```
>>> ensemble = {"Bob", "Tom", "Bill"}
>>> ensemble.add("Jim")
>>> ensemble
{'Bill', 'Tom', 'Jim', 'Bob'}
>>> ensemble.remove("Jim")
>>> ensemble
{'Bill', 'Tom', 'Bob'}
```

2.5 Référence des objets

Rappels et principes :

Les nombres, les chaînes de caractères,… sont des **objets « immuables / immutables » : leurs méthodes ne modifient pas l'objet** (mais renvoient un nouvel objet modifié).

Les listes, les dictionnaires,… sont des **objets « muables / mutables » : leurs méthodes modifient directement l'objet** (en fonction de l'objectif de la méthode).

Une variable est en fait un nom identifiant, pointant vers la référence de l'objet (sa position en mémoire Python ⇒ *"adresse"*).

La fonction intégrée **« id » retourne la référence de l'objet**.

L'opérateur **« == » compare les valeurs / contenus**.

Le mot-clé **« is » compare les références**.

Interpréteur Python :

```
>>> var1 = "essai"
>>> var2 = var1     # ont la même référence d'objet
>>> var1 ; id(var1)
'essai'
140227330927152
>>> var2 ; id(var2)
'essai'
140227330927152
>>> var1 == var2
True
>>> var1 is var2
True
>>> var1 += ", essai2"     # ne modifie pas l'objet, nouvel objet créé !
>>> var1 ; id(var1)
'essai, essai2'
140227330928048
>>> var2 ; id(var2)
'essai'
140227330927152
>>> var1 == var2
False
>>> var1 is var2
False
```

Interpréteur Python :

```
>>> liste1 = ["essai"]
>>> liste2 = liste1     # ont la même référence d'objet (mais pas : liste2 = list(liste1) )
>>> liste1 ; id(liste1)
['essai']
140586784186944
>>> liste2 ; id(liste2)
['essai']
140586784186944
>>> liste1 == liste2
True
>>> liste1 is liste2
True
>>> liste1.append("essai2")     # modifie directement l'objet
>>> liste1 ; id(liste1)
['essai', 'essai2']
140586784186944
>>> liste2 ; id(liste2)
['essai', 'essai2']
140586784186944
>>> liste1 == liste2
True
>>> liste1 is liste2
True
```

2.6 Fichiers

2.6.1 Module os

Le module « os » **permet d'effectuer des opérations courantes liées au système d'exploitation** (très riche en fonctionnalités, notamment en gestion de fichiers).

Il est **indépendant du système d'exploitation** et peut fonctionner sur n'importe quel OS.

Importer le module « os » : **import os**
Aide et fonctionnalités : **help(os)** ⇒ *liste : dir(os)*

2.6.2 Répertoire courant

Changer de répertoire courant : **os.chdir("*chemin*")** ⇒ *avec chemin absolu ou relatif*
⇒ *et des « / » (même sur Windows)*

Obtenir le répertoire courant : **os.getcwd()** ⇒ *depuis la racine (chemin absolu)*

2.6.3 Ouverture d'un fichier

La fonction intégrée **« open » (sans import) ouvre un fichier et le renvoie en tant qu'objet fichier de la classe « _io.TextIOWrapper »** pour lecture, écriture ou modification (à l'aide de méthodes dédiées).

Syntaxe générale :

open("*chemin/fichier*", mode) ⇒ *avec chemin absolu ou relatif*

Avec pour modes :

r (read) : ouvre le fichier en lecture (mode par défaut), erreur si inexistant
w (write) : ouvre le fichier en écriture, le crée si inexistant sinon l'écrase
x (create) : crée le fichier et l'ouvre en écriture, erreur si existant
a (append) : ouvre le fichier en ajout (à la fin), le crée si inexistant
b (binary) : après r/w/x/a ouvre le fichier en mode binaire
t (text) : après r/w/x/a ouvre le fichier en mode texte (mode par défaut)
+ : après r/w/x/a ouvre le fichier en mise à jour (lecture et écriture)

Ouvrir en lecture le fichier texte « fichier.txt » (existant dans le répertoire courant) :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> fichier
<_io.TextIOWrapper name='fichier.txt' mode='r' encoding='UTF-8'>
>>> type(fichier)
<class '_io.TextIOWrapper'>
```

2.6.4 Fermeture d'un fichier

Une fois utilisé, **un fichier doit être fermé à l'aide de la méthode « close » :**

Interpréteur Python :

```
>>> fichier.close()
```

⇒ *libère les ressources liées au fichier*
⇒ *en écriture, les modifications sont réellement écrites sur disque au moment du « close »*

2.6.5 Lecture d'un fichier

Par défaut, la méthode « read » de la classe « TextIOWrapper » lit l'intégralité d'un fichier :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> contenu = fichier.read()
>>> contenu
'Une ligne\nUne autre ligne\nEncore une ligne\n'
>>> print(contenu)
Une ligne
Une autre ligne
Encore une ligne

>>> fichier.close()
```

On peut tout faire avec la variable récupérant le contenu (ici du texte), par exemple la "splitter" en liste avec la méthode « split » afin de traiter ses lignes individuellement.

⇒ *la taille maximum est limitée par la mémoire*
⇒ *« read » peut prendre en paramètre la taille maximum lue (nombre de caractères / octets)*
⇒ *une boucle avec « fichier.read(1) » peut lire le fichier caractère par caractère*

La méthode « splitlines » permet de lire l'intégralité d'un fichier directement dans une liste :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> liste = fichier.read().splitlines()
>>> liste
['Une ligne', 'Une autre ligne', 'Encore une ligne']
>>> fichier.close()
```

Pour lire un fichier ligne par ligne : ⇒ *allège la mémoire*

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> for ligne in fichier:
...     ligne = ligne.strip()    # retire le retour à la ligne (\n)
...     print(ligne)
...
Une ligne
Une autre ligne
Encore une ligne
>>> fichier.close()
```

La méthode « readline » lit uniquement la ligne suivante du fichier :

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r")
>>> fichier.readline()
'Une ligne\n'
>>> fichier.readline().strip()     # retire le retour à la ligne (\n)
'Une autre ligne'
>>> fichier.readline().strip()
'Encore une ligne'
>>> fichier.readline().strip()
''
>>> fichier.close()
```

2.6.6 Ecriture dans un fichier

La méthode « write » de la classe « TextIOWrapper » permet d'écrire dans un fichier.

Elle **n'accepte en paramètre que des chaînes de caractères**. ⇒ *convertir les autres types*
(ou du binaire avec le mode « b »)

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "w")
>>> fichier.write("Ligne une\n")
10
>>> fichier.write("Ligne deux\n")
11
>>> fichier.write("Ligne trois\n")
12
>>> fichier.close()
```

⇒ *« write » retourne le nombre de caractères écrits*
⇒ *le mode « w » écrase le fichier, le mode « a » ajoute à la fin du fichier*

2.6.7 Position dans le fichier

La méthode « seek » permet de changer la position dans le fichier.

La méthode « tell » permet de connaître la position dans le fichier.

⇒ *à partir de 0 (zéro)*

Interpréteur Python :

```
>>> fichier = open("fichier.txt", "r+")     # lecture avec mise à jour
>>> fichier.read()
'Ligne une\nLigne deux\nLigne trois\n'
>>> fichier.seek(16)
16
>>> fichier.tell()
16
>>> fichier.read(4)
'deux'
>>> fichier.tell()
20
>>> fichier.seek(16)
16
>>> fichier.write("XXXX")
4
>>> fichier.tell()
20
>>> fichier.seek(0)
0
>>> fichier.read()
'Ligne une\nLigne XXXX\nLigne trois\n'
>>> fichier.close()
```

2.6.8 Ouverture avec with, as

Ouvrir un fichier avec **« with, as » évite de le fermer avec la méthode « close »**.

Si une erreur se produit dans le bloc de traitement du fichier, l'exception se lève et le fichier est automatiquement fermé. ⇒ *fiabilise les traitements et plus court qu'un « try, finally »*

Interpréteur Python :

```
>>> with open("fichier.txt", "r") as fichier:
...         print(fichier.read())
...
Ligne une
Ligne XXXX
Ligne trois

>>> fichier.closed
True
>>> print(fichier.read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

2.6.9 Ecriture et lecture d'objets

Le module « pickle » dispose de fonctionnalités permettant d'écrire et de lire des objets dans des fichiers, à importer au préalable :

import pickle

Il possède les **classes « Pickler » pour écrire un objet et « Unpickler » pour lire un objet**.

Un objet de la classe « Pickler » permet d'écrire un objet dans un fichier à l'aide de la méthode « dump » :

Interpréteur Python :

```
>>> dictionnaire = {"nom": "Tom", "age": 40, "taille": 1.75}
>>>
>>> with open("objet-dictionnaire", "wb") as fichier:     # écriture obligatoirement en binaire
...         mon_pickler = pickle.Pickler(fichier)     # crée l'objet pickler pour le fichier
...         mon_pickler.dump(dictionnaire)     # écrit l'objet dans le fichier grâce à l'objet pickler
```

⇒ *on peut écrire avec des « dump » plusieurs objets à la suite dans le fichier*

Un objet de la classe « Unpickler » permet de lire un objet d'un fichier à l'aide de la méthode « load » :

Interpréteur Python :

```
>>> with open("objet-dictionnaire", "rb") as fichier:     # lecture obligatoirement en binaire
...         mon_unpickler = pickle.Unpickler(fichier)     # crée l'objet unpickler pour le fichier
...         dictionnaire_lu = mon_unpickler.load()     # lit l'objet du fichier grâce à l'objet unpickler
...
>>> dictionnaire_lu
{'nom': 'Tom', 'age': 40, 'taille': 1.75}
```

⇒ *on peut lire avec des « load » plusieurs objets à la suite dans le fichier*

Le module « pickle » **implémente donc des protocoles binaires de sérialisation et dé-sérialisation d'objets Python**.