Boîte à outils - Premiers pas avec Python 3

Christophe BAL : projetmbc@gmail.com

Version du 2015-10-18

Mentions « légales »

Le code source LaTeX de ce document est disponible sur github. Voir le dossier `program[mation]/python3/toolboxes/beginner/fr` à l’adresse `https://github.com/projetmbc/books`.

Table des matières

Ce document sans prétention propose de réunir un ensemble de techniques de base sous une forme facile d'utilisation. Le but ici n'est pas d'explorer en détail les différents thèmes proposés mais au contraire de proposer un ensemble d'outils suffisants et utiles quand on débute. Ceci évitera aux débutants de se noyer dans des concepts un peu trop avancés. Toute remarque constructive est bienvenue pour faire évoluer **ce document non définitif**.

1 Généralités

1.1 Programmer, c'est quoi ?

La programmation sert à résoudre des problèmes utiles ou non, concrets ou asbtraits, en s'appuyant sur un langage plus ou moins simple qui fera appel aux capacités de calcul d'un ordinateur tel qu'un ordinateur portable, un smartphone ou aussi par exemple aux fonctionnalités d'un Raspberry. Suivant les problèmes à résoudre, on pourra s'orienter vers différents langages. Voici des exemples possibles d'utilisation.

1. Vous souhaitez gérer au mieux la mémoire pour de l'électronique embarquée, un langage comme le C devrait vous rendre de grand service.
2. Votre problème à résoudre n'utilise pas de grosses données volumineuses, ou bien vous n'avez pas de restriction quant à l'utilisation de bibliothèques tierces programmées par d'autres, Python pourrait vous rendre très productif.
3. Envie de mettre en page des rapports de façon automatisée, intéressez vous au langage HTML, ou bien à LaTeX.
4. Vous devez partir à l'assaut de données structurées ou non, dans ce cas, les langages de type SQL ou ceux de la famille des NoSQL seront là pour vous servir.
5. ... etc.

Dans les exemples précédents, les langages HTML, SQL et NoSQL ont ceci de notables qu'ils ne permettent pas de faire des calculs comme par exemple le calcul du PGCD de deux naturels. Tous les autres le peuvent !

1.2 Le choix de Python

Voici des avantages qui peuvent justifier le choix de Python comme langage de programmation.

1. Le nombre de mots clés de base du langage est limité et la syntaxe très économe. Les instructions se rédigent dans un langage proche du nôtre.
2. La structuration visuelle du code fait partie des régles de rédaction d'un code *(ceci améliore la lisibilité)*.
3. On peut utiliser des codes Python sous Linux, Mac OS et Windows sans avoir à utiliser un processus *"complexe"* de compilation. Par contre, ceci demande d'installer initialement un interpréteur, mais cette installation reste simple à faire.
4. Les variables utilisées n'ont pas besoin d'être typées par le programmeur. C'est un peu comme en mathématique quand dans un même devoir la lettre F peut dans un exercice indiquer un point, puis dans un autre une primitive.
5. Python se charge des problèmes de gestion de la mémoire en libérant tout ce qui n'est plus utile *(ceci implique que l'on peut faire des choses très lentes si l'on utilise mal Python mais ceci ne veut pas dire qu'au contraire on ne puisse pas faire des choses très rapidement)*.
6. Un nombre important de bibliothèques permettent de palier aux manques intrinsèques de rapidité du langage Python *(l'usage de ces bibiothèques n'empêchent pas d'avoir à réfléchir pour élaborer des codes)*.
7. On a le choix de la technique de programmation. Si l'on veut faire de la programmation orientée objet, pas de souci. Si l'on veut n'utiliser que des fonctions pour faire de la programmation itérative à la sauce C, c'est aussi possible. On peut même mélanger les deux types d'approche. Il est aussi possible d'adapter à Python certaines techniques de programmation propres aux langages dits fonctionnels comme OCaml.

1.3 Installer Python 3

1.3.1 Pyzo

Les contraintes matérielles liées aux réseaux dans les lycées français font que l'usage de Pyzo, voir ce lien `http://www.pyzo.org/`, semble le choix le plus simple *(Anaconda aurait été un choix intéressant pour des ordinateurs Linux en réseau)*. Pyzo propose un environnement capable d'interpréter les programmes écrits en Python en utilisant éventuellement un ensemble de bibliothèques intéressantes proposées par défaut. Pyzo s'accompagne de IEP un éditeur de codes très correct, et aussi de IPython qui permet de rédiger efficacement des documents *"interactifs"* contenant des codes Python.

1.3.2 IPython

Dans sa version IPython via un navigateur, le document que vous lisez permet de proposer des documents contenant des codes modifiables comme dans l'exemple suivant.

```
# Quand IPython est lancé en tant qu'application web locale dans
# un navigateur web pour ouvrir ce document, il est alors possible
# de changer ci-dessous les lignes de codes.
#
# Par exemple, il est alors possible de changer le texte entre les
# guillemets, puis en cliquant sur le triangle "Play" en haut de
# la page, ou bien en utilisant le raccourci clavier CTRL + ENTREE,
# on peut constater l'effet des modifications apportées.
mon_nom = "Personne"

print("Mon nom :", mon_nom)
```

Sortie Python

```
Mon nom : Personne
```

1.4 Utilisez des algorithmes !

Même si ceci peut se faire uniquement mentalement, programmer demande une grande organisation. Un outil très utile est la notion d'algorithme. Par exemple, je souhaite afficher trois nombres réels connus `a`, `b` et `c` du plus petit au plus grand. Je peux opter pour la tactique suivante écrite sous forme d'un algorithme utilisant un langage naturel où les lignes commençant par le symbole sharp `#` sont des commentaires *(la tactique proposée n'est pas optimale mais peu importe)*.

```
# On se débrouille pour avoir b <= c.
Si c < b alors échanger les valeurs de b et c.

# On est certain que b <= c à ce stade. Il reste
# juste à mettre le réel a au bon endroit.

# Cas 1: a < b . On n'a rien à faire.

# Cas 2: b <= a <= c
Si b <= a <= c alors échanger les valeurs de a et b.
```

```
# Cas 3: a > c
Si a > c, autrement dit b <= c < a, alors procéder
comme suit.
    + a prend l'ancienne valeur de b.
    + b prend l'ancienne valeur de c.
    + c prend l'ancienne valeur de a.

# Fin du travail
Afficher a, b et c dans cet ordre.
```

Ceci se traduit assez directement en Python comme suit à condition de disposer d'un *"dictionnaire"* permettant une traduction dans le langage Python. Les sections suivantes vous aideront à faire vos propres traductions.

```
# Donnons directement des valeurs dans le code.
a = 2
b = 1
c = 3

# On se débrouille pour avoir b <= c.
if c < b:
    b, c = c, b

# On est certain que b <= c à ce stade. Il reste
# juste à mettre le réel a au bon endroit.

# Cas 1: a <= b . On n'a rien à faire.

# Cas 2: b <= a <= c
if b <= a <= c:
    a, b = b, a

# Cas 3: a > c
if a > c:
    a, b, c = b, c, a

# Fin du travail
print(a, "<=", b, "<=", c)
```

Sortie Python

```
1 <= 2 <= 3
```

Important ! Il ne faut jamais négliger la phase de réflexion avant de programmer. Ce ne sera jamais une perte de temps que d'organiser ses pensées avant de taper des lignes de code.

2 Python 3, comment fait-on ?

2.1 Comment taper et utiliser un programme écrit en Python

2.1.1 IEP, un outil deux en un

L'intérêt de IEP est qu'il est *"rassurant"* quand l'on débute car il permet à la fois de taper son code, puis ensuite de le lancer directement au sein d'une unique interface graphique.

2.1.2 IPython, un outil deux en un mais plus spécifique

On peut aussi utiliser IPython si l'on souhaite faire des bouts de code, ou bien si l'on veut écrire un code que l'on souhaite commenter de façon très visuelle.
Remarque : on peut exporter une feuille IPython sous forme d'un fichier Python. Dans ce cas, toutes les explications deviendront des commentaires dans votre code.
Attention ! Dans une feuille IPython, on peut utiliser différentes cellules pour taper différentes séquences de code. Il faut savoir que IPython garde en mémoire tout ce qui a été fait avant. Voici un exemple de code.

```
x = 3
```

Tapons maintenant du texte toujours dans le cadre de notre exemple. Dans la cellule suivante, la variable `x` n'est pas définie mais nous pouvons l'utiliser car elle a été définie dans une cellule précédente.

```
# ‘print‘ sert à afficher un résultat.
print(x*100)
```

Sortie Python

```
300
```

Il faut garder ceci en tête car quelque fois on peut être étonné du résultat renvoyé par IPython qui utilise la valeur d'une variable définie précédemment.

2.1.3 Faire comme les pros

Avec l'habitude, certaines personnes trouvent plus simple d'utiliser des éditeurs de programmes légers et robustes comme Atom, Notepad++, Geany, ... etc couplé avec une invite de commande ou un terminal bien configuré, c'est à dire avec un alias `python` qui lance directement Python.

2.2 Commenter, c'est important

Python ne propose qu'un seul type de commentaires : il faut utiliser la caractère hash `#` en début de ligne avec éventuellement des espaces et/ou des tabulations le précédant. Le code suivant ne contenant aucune commande Python, il ne produira rien si on le lance.

```
# Je suis un commentaire
         # Moi aussi
    # Moi de même...
```

Les commentaires sont importants dans un code pour lever des ambiguïtés ou expliquer des petits points techniques. Faites attention à ne pas les utiliser de façon excessive. Si vous commencez à commenter beaucoup votre code, il faudra vous poser deux questions.

1. Mon commentaire est à but pédagogique. Dans ce cas, pas de souci.
2. Mon commentaire est indispensable pour comprendre ce que je fais. Dans ce cas, il faudra peut-être s'interroger sur la façon dont le code a été tapé, ou bien sur l'algorithme employé.

2.3 Communiquer basiquement avec l'utilisateur

Une façon évoluée de communiquer est d'utiliser une interface graphique comme celle de IPython par exemple. Ceci demande pas mal de travail et n'est donc pas faisable quand on débute. On peut aussi simplement passer par l'invite de commande, ou terminal. C'est ce qui est présenté ici.

2.3.1 Afficher du texte

Pour afficher par exemple le texte *"Nous ne dirons pas bonjour au monde car il n'est pas là dans son ensemble."* sans les guillemets, il suffit de faire comme suit où la fonction `print` proposée par Python signifie *"imprimer" (il faut se souvenir que les premiers ordinateurs disposaient juste d'une imprimante pour retourner des résultats).*

```
print("Nous ne dirons pas bonjour au monde car il n'est pas complètement là aujourd'hui.")
```

Sortie Python

```
Nous ne dirons pas bonjour au monde car il n'est pas complètement là aujourd'hui.
```

2.3.2 Récupérer du texte demandé à l'utilisateur dans une invite de commandes ou un terminal

La fonction `input` sert à poser une question à l'utilisateur pour récupérer une réponse de type texte.

```
reponse = input("Taper un texte puis utilisez la touche [ENTREE]: ")

print("Votre texte :", reponse)
```

Sortie Python

```
Taper un texte puis utilisez la touche [ENTREE]: Je ne sais que écrire...
Votre texte : Je ne sais que écrire...
```

2.4 Textes

2.4.1 Créer une variable contenant du texte

Vous disposez deux types de guillemets simples `'...'` et `"..."` pour taper directement dans le code du texte, sans retour à la ligne. Voici deux exemples.

```
une_ligne = 'Que dire, je ne sais pas !'
une_autre_ligne = "Toujours en manque d'inspiration..."

print(une_ligne)
print(une_autre_ligne)
```

Sortie Python

```
Que dire, je ne sais pas !
Toujours en manque d'inspiration...
```

Le code suivant montre que les guillemets simples ne permettent pas de taper directement un texte sur plusieurs lignes *(dans le message d'erreur ci-dessous, \textttEOL est l'acronyme de \textit"End Of Line" soit \textit"fin de ligne" en français).*

```
interdit = "Ce qui est tapé
ici n'est pas autorisé !"
```

Sortie Python

```
File "<ipython-input-11-21d1c80e2827>", line 1
    interdit = "Ce qui est tapé
                               ^
SyntaxError: EOL while scanning string literal
```

2.4.2 Définir du texte avec des retours à la ligne

Les triples guillemets '''...''' et """..""" permettent d'écrire du texte contenant des retours à la ligne.

```
retours_a_ligne = """Ce que j'ai tapé
est valide
maintenant."""

print(retours_a_ligne)
```

Sortie Python

```
Ce que j'ai tapé
est valide
maintenant.
```

2.4.3 C'est quoi une chaîne pour Python ?

Les textes sont des instances de la classe `str` pour *"string"* soit *"chaîne"* de caractères.

```
print(type("Je suis une chaîne de caractères."))
```

Sortie Python

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

2.4.4 Accoler différents textes les uns après les autres

Pour faire ceci du point de vue de Python, on parle plus techniquement de concaténation de chaînes de caractères, il suffit d'additionner des chaînes.

```
sujet = "Je"
verbe = "suis"
cod   = "lui"

phrase = sujet + " " + verbe + " " + cod + "."

print(phrase)
```

Sortie Python

```
Je suis lui.
```

Il existe un racourci pour additionner plusieurs fois de suite la même chaîne de caractères.

```
drole      = "ah"
tres_drole = drole*3   # 3 fois de suite.
hilarant   = drole*12  # 12 fois de suite.

print("drole      :", drole)
print("tres_drole :", tres_drole)
print("hilarant   :", hilarant)
```

Sortie Python

```
drole      : ah
tres_drole : ahahah
hilarant   : ahahahahahahahahahahahah
```

2.4.5 Extraire une lettre d'un texte

Tout se fait assez naturellement dès lors que l'on sait qu'un premier caractère aura la position 0 et non 1.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

# Les crochets permettent d'indiquer une position.
premiere_lettre  = alphabet[0]
troisieme_lettre = alphabet[2]

# Une valeur négative peut-être utilisée pour partir
# de la fin, la dernière lettre ayant le numéro (-1).
derniere_lettre       = alphabet[-1]
antepenultieme_lettre = alphabet[-3]

# Vérifications
print("premiere_lettre       :", premiere_lettre)
print("troisieme_lettre      :", troisieme_lettre)

print("derniere_lettre       :", derniere_lettre)
print("antepenultieme_lettre :", antepenultieme_lettre)
```

Sortie Python

```
premiere_lettre       : a
troisieme_lettre      : c
derniere_lettre       : z
antepenultieme_lettre : x
```

2.4.6 Extraire un groupe de lettres d'un texte

De nouveau, on va utiliser des crochets.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

# Ci-dessous, dans `[4:7]` on peut être étonné de voir `7`
# et non `6`. Ce qui va suivre montrera que ceci n'est pas
# qu'une simple convention.
les_trois_lettres_apres_le_D = alphabet[4:7]

# Ci-après, l'absence d'un second entier dans `[7:]` indique
# d'aller jusqu'à la fin de la chaîne de caractères.
les_lettres_apres_le_G = alphabet[7:]

# On peut utiliser une convention analogue à la précédente en
# omettant le premier entier pour partir du tout début de la
# chaîne de caractères.
les_lettres_avant_le_E = alphabet[:4]

# Vérifications
print("les_lettres_vant_le_E        :", les_lettres_avant_le_E)
print("les_trois_lettres_apres_le_D :", les_trois_lettres_apres_le_D)
print("les_lettres_apres_le_G       :", les_lettres_apres_le_G)
```

Sortie Python

```
les_lettres_vant_le_E        : abcd
les_trois_lettres_apres_le_D : efg
les_lettres_apres_le_G       : hijklmnopqrstuvwxyz
```

2.4.7 Modifier un texte caractère par caractère

Il est impossible de modifier directement un caractère d'une chaine de caractères comme le montre l'exemple ci-dessous.

```
texte = "Testons l'impossible..."

texte[0] = "Z"
print(texte)
```

Sortie Python

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-a24373db672e> in <module>()
      1 texte = "Testons l'impossible..."
      2
----> 3 texte[0] = "Z"
      4 print(texte)
TypeError: 'str' object does not support item assignment
```

Pour changer la première lettre de notre texte, il va falloir procéder d'une façon analogue à la suivante.

```
texte = "Testons l'impossible..."

texte = "Z" + texte[1:]
print(texte)
```

Sortie Python

```
Zestons l'impossible...
```

2.4.8 Remplacer du texte

Voici deux exemples de remplacements effectués partout dans un texte. Notez que l'on peut changer des morceaux de texte en d'autres sans le faire lettre par lettre, et aussi que le texte initial n'a pas été modifié.

```
texte = "zozo et tata sont sympas"

texte_a_en_i   = texte.replace("a", "i")
texte_ta_en_ri = texte.replace("ta", "ri")

print("texte          =", texte)
print("texte_a_en_i   =", texte_a_en_i)
print("texte_ta_en_ri =", texte_ta_en_ri)
```

Sortie Python

```
texte          = zozo et tata sont sympas
texte_a_en_i   = zozo et titi sont sympis
texte_ta_en_ri = zozo et riri sont sympas
```

2.4.9 Nettoyer ces espaces que je ne saurais voir

Les textes sont des instances de la classe `str` qui possède une méthode `strip` qui permet de retirer les espaces et les tabulations qui sont éventuellement au tout début et tout à la fin d'un texte. Le code suivant montre un exemple concret d'utilisation. Notez au passage que le texte initial n'a pas été modifié.

```
reponse = "  Prénom NOM    "

reponse_nettoye = reponse.strip()

# Nous modifions le paramètre ‘sep‘ de la fonction
# ‘print‘ qui par défaut vaut un espace.
print("reponse         : >>", reponse, "<<", sep = "")
print("reponse_nettoye : >>", reponse_nettoye, "<<", sep = "")
```

Sortie Python

```
reponse         : >>  Prénom NOM    <<
reponse_nettoye : >>Prénom NOM<<
```

2.4.10 Un texte *(en)* capitale*(s)*

Les textes sont des instances de la classe `str` qui possède deux méthodes `upper` et `lower` pour mettre respectivement un texte tout en majuscule ou tout en minuscule. Le code ci-dessous explique comment les utiliser. Notez au passage que le texte initial n'a pas été modifié.

```
texte = "Un texte capitale ou minuscule ? Bien testé ?"

texte_majuscules = texte.upper()
texte_minuscules = texte.lower()

print("texte            :", texte)
print("texte_majuscules :", texte_majuscules)
print("texte_minuscules :", texte_minuscules)
```

Sortie Python

```
texte            : Un texte capitale ou minuscule ? Bien testé ?
texte_majuscules : UN TEXTE CAPITALE OU MINUSCULE ? BIEN TESTÉ ?
texte_minuscules : un texte capitale ou minuscule ? bien testé ?
```

2.4.11 Longueur d'un texte

Le mot *"longueur"* se traduit en *"len"* en anglais. Une fois ceci connu, le code suivant peut se *"deviner"*.

```
texte = "1234567890"

print(len(texte))
```

Sortie Python

```
10
```

2.5 Les nombres - Différents types

2.5.1 Entiers relatifs

Sous Python, les entiers relatifs ne sont pas limités en taille. Ils s'écrivent très naturellement...

```
un                  = 1
moins_dix           = -10
grand_naturel       = 2**200
oppose_grand_naturel = -2**200

print("un                   :", un)
print("moins_dix            :", moins_dix)
print("grand_naturel        :", grand_naturel)
print("oppose_grand_naturel :", oppose_grand_naturel)

print("")
print("Type de un                   :", type(un))
print("Type de moins_dix            :", type(moins_dix))
print("Type de grand_naturel        :", type(grand_naturel))
print("Type de oppose_grand_naturel :", type(oppose_grand_naturel))
```

Sortie Python

```
un                  : 1
moins_dix           : -10
grand_naturel       : 1606938044258990275541962092341162602522202993782792835301376
oppose_grand_naturel : -1606938044258990275541962092341162602522202993782792835301376

Type de un                   : <class 'int'>
Type de moins_dix            : <class 'int'>
Type de grand_naturel        : <class 'int'>
Type de oppose_grand_naturel : <class 'int'>
```

2.5.2 Nombres réels approchés

La notion de nombres réels n'existe pas en informatique *(même si l'on peut faire appel à des bibliothèques de calcul formel qui tente d'effectuer un maximum de calculs de façon exacte)*. A la place, on dispose des nombres à virgule flottante qui permettent de faire des calculs approchés. Sans entrer dans les détails techniques, savoir que pour Python un nombre à virgule est en fait un nombre à virgule **flottante**, et non un nombre décimal, permet de ne pas trop être surpris par ce qu'affiche le code suivant *(à cette adresse* ***http://www.afpy.org/doc/python/2.7/tutorial/floatingpoint.html*** *se trouve un document expliquant très bien ce qu'il se passe en coulisse)*.

```
resultat_bizarre = 0.1 + 0.2

print("resultat_bizarre :", resultat_bizarre)
print("Type de resultat_bizarre :", type(resultat_bizarre))
```

Sortie Python

```
resultat_bizarre : 0.30000000000000004
Type de resultat_bizarre : <class 'float'>
```

2.5.3 Décimaux

Il est possible de travailler de façon exacte avec des nombres décimaux via un module Python spécialisé. Il suffit de procéder comme suit.

```
from decimal import Decimal

resultat_normal = Decimal("0.1") + Decimal("0.2")

print("resultat_normal :", resultat_normal)
print("Type de resultat_normal :", type(resultat_normal))
```

Sortie Python

```
resultat_normal : 0.3
Type de resultat_normal : <class 'decimal.Decimal'>
```

2.5.4 Fractions

On peut définir des fractions rationnelles en gardant leur valeur exacte grâce à un module Python spécialisé.

```
from fractions import Fraction

fraction_approchee = 1/3
fraction_exacte    = Fraction("1/3")

print("fraction_approchee :", fraction_approchee)
print("fraction_exacte    :", fraction_exacte)

print("")
print("Type de fraction_approchee :", type(fraction_approchee))
print("Type de fraction_exacte    :", type(fraction_exacte))
```

Sortie Python

```
fraction_approchee : 0.3333333333333333
fraction_exacte    : 1/3

Type de fraction_approchee : <class 'float'>
Type de fraction_exacte    : <class 'fractions.Fraction'>
```

2.5.5 Nombres complexes approchés

Python propose par défaut un type pour calculer avec des nombres complexes approchés, plus précisément ces nombres seront des complexes ayant pour parties imaginaire et réelle des nombres à virgule flottante.

```
# Attention ! j tout seul ne fonctionnera pas.
nbre_i_en_math    = 1j
un_reel_complexe  = 2 + 0j

un_complexe       = 2 - 4j
partie_imaginaire = un_complexe.imag
partie_reelle     = un_complexe.real

print("nbre_i_en_math    :", nbre_i_en_math)
print("un_reel_complexe  :", un_reel_complexe)

print("un_complexe       :", un_complexe)
print("partie_imaginaire :", partie_imaginaire)
print("partie_reelle     :", partie_reelle)

print("")
print("Type de nbre_i_en_math   :", type(nbre_i_en_math))
print("Type de un_reel_complexe :", type(un_reel_complexe))

print("")
print("Type de un_complexe       :", type(un_complexe))
print("Type de partie_imaginaire :", type(partie_imaginaire))
print("Type de partie_reelle     :", type(partie_reelle))
```

Sortie Python

```
nbre_i_en_math    : 1j
un_reel_complexe  : (2+0j)
un_complexe       : (2-4j)
partie_imaginaire : -4.0
partie_reelle     : 2.0

Type de nbre_i_en_math   : <class 'complex'>
Type de un_reel_complexe : <class 'complex'>

Type de un_complexe       : <class 'complex'>
Type de partie_imaginaire : <class 'float'>
Type de partie_reelle     : <class 'float'>
```

2.5.6 Récupérer un nombre d'un type particulier demandé à l'utilisateur

Pour chacun des types rencontrés précédemment, autre que les cas particuliers des décimaux et des fractions, il existe une fonction éponyme qui transforme une chaîne de caractères en un nombre du type souhaité, dans la mesure du possible. Ceci permet de récupérer un nombre d'un type particulier demandé à l'utilisateur dans une invite de commandes ou un terminal.

```
reponse_texte = input("Entrer un nombre : ")

version_entiere   = int(reponse_texte)
version_flottante = float(reponse_texte)
version_complexe  = complex(reponse_texte)

print("")
print("Réponse via ‘input‘ :", reponse_texte)
print("Type : ", type(reponse_texte))

print("")
print("Version ‘int‘ :", version_entiere)
print("Type : ", type(version_entiere))

print("")
print("Version ‘float‘ :", version_flottante)
print("Type ; ", type(version_flottante))

print("")
print("Version ‘complex‘ :", version_complexe)
print("Type : ", type(version_complexe))
```

Sortie Python

```
Entrer un nombre : 1418

Réponse via ‘input‘ : 1418
Type :  <class 'str'>

Version ‘int‘ : 1418
Type :  <class 'int'>

Version ‘float‘ : 1418.0
Type ;  <class 'float'>

Version ‘complex‘ : (1418+0j)
Type :  <class 'complex'>
```

2.6 Les nombres - Comment les manipuler

2.6.1 Addition et soustraction

Ce qui suit paraît évident mais nous allons voir dans la section qui suit qu'il y a des pièges cachés à éviter.

```
a = 15
b = 7

somme      = a + b
difference = b - a

print("somme      :", somme)
print("difference :", difference)

print("Type de somme      :", type(somme))
print("Type de difference :", type(difference))
```

Sortie Python

```
somme      : 22
difference : -8
Type de somme      : <class 'int'>
Type de difference : <class 'int'>
```

2.6.2 Attention aux changements de types !

Nous venons de voir comment additionner et soustraire deux nombres. Rien de magique. Ceci étant dit, il faut faire attention aux changements de type. Voici une illustration.

```
un_naturel       = 2
un_reel_approche = 7.0
un_complexe      = -4 + 0j

naturel_plus_reel     = un_naturel + un_reel_approche
naturel_plus_complexe = un_naturel + un_complexe
reel_plus_complexe    = un_reel_approche + un_complexe

print("Type de naturel_plus_reel     :", type(naturel_plus_reel))
print("Type de naturel_plus_complexe :", type(naturel_plus_complexe))
print("Type de reel_plus_complexe    :", type(reel_plus_complexe))
```

Sortie Python

```
Type de naturel_plus_reel     : <class 'float'>
Type de naturel_plus_complexe : <class 'complex'>
Type de reel_plus_complexe    : <class 'complex'>
```

Ce qui saute le plus aux yeux, c'est `2 + 7.0` qui est traité comme un nombre flottant et non comme un entier. Ceci vient de ce que `7.0` est un nombre flottant, et non le décimal `7,0` et encore moins le naturel `7`.

2.6.3 Multiplication et division

Dans ce qui suit, bien que $\frac{15}{3} = 5$, Python utilise le type `float` pour la division $15/3$. Ceci implique au passage qu'il va falloir faire autrement si l'on souhaite obtenir le quotient de la division euclidenne de deux entiers. Ceci est expliqué un peu plus bas.

```
a = 15
b = 7
c = 3

produit    = a*b
division_1 = a/b
division_2 = a/c

print("produit    :", produit)
print("division_1 :", division_1)
print("division_2 :", division_2)

print("Type de produit    :", type(produit))
print("Type de division_1 :", type(division_1))
print("Type de division_2 :", type(division_2))
```

Sortie Python

```
produit    : 105
division_1 : 2.142857142857143
division_2 : 5.0
Type de produit    : <class 'int'>
Type de division_1 : <class 'float'>
Type de division_2 : <class 'float'>
```

2.6.4 Puissance

Comme $2^{-4} = \dfrac{1}{2^4}$, il n'est pas étonnant de voir apparaître le type `float` dans l'exemple qui suit.

```
a = 2
b = 7
c = -4

puissance_1 = a**b
puissance_2 = a**c

print("puissance_1 :", puissance_1)
print("puissance_2 :", puissance_2)

print("Type de puissance_1 :", type(puissance_1))
print("Type de puissance_2 :", type(puissance_2))
```

Sortie Python

```
puissance_1 : 128
puissance_2 : 0.0625
Type de puissance_1 : <class 'int'>
Type de puissance_2 : <class 'float'>
```

2.6.5 Division euclidienne

La division euclidienne de 128 par 5, celle que l'on apprend à l'école primaire, peut s'écrire $128 = 5 \times 25 + 3$ où 3 est appelé reste, ou plus précisément reste modulo 5, tandis que 25 est appelé quotient. Voyons comment faire cela avec Python.

```
n = 128

# Calcul modulo 5 pour obtenir le reste.
reste = n%5

# Quotient d'une division euclidienne.
quotient = n//5

print("reste    :", reste)
print("quotient :", quotient)

print("Type de reste    :", type(reste))
print("Type de quotient :", type(quotient))
```

Sortie Python

```
reste    : 3
quotient : 25
Type de reste    : <class 'int'>
Type de quotient : <class 'int'>
```

2.7 Les booléens `VRAI` et `FAUX`

2.7.1 Un peu de vocabulaire booléen

George Boole a vécu du 2 novembre 1815 au 8 décembre 1864. Il a inventé ce que l'on appelle aujourd'hui l'algèbre de Boole car il cherchait traduire les raisonnements logiques en calculs algébriques. Indiquons au passage qu'il existe plusieurs types de logique. Celle modélisée par l'agèbre de Boole s'appuie sur deux valeurs de vérité `VRAI` et `FAUX` utilisables avec des opérateurs logiques `ET` , `OU` et `NON`. Commençons par voir comment distinguer le `VRAI` du `FAUX`.

```
variable_symbolisant_vrai = True
variable_symbolisant_faux = False

print("Type de variable_symbolisant_vrai :", type(variable_symbolisant_vrai))
print("Type de variable_symbolisant_faux :", type(variable_symbolisant_faux))
```

Sortie Python

```
Type de variable_symbolisant_vrai : <class 'bool'>
Type de variable_symbolisant_faux : <class 'bool'>
```

2.7.2 Les opérateurs booléens

Il suffit de savoir que *"et"*, *"ou"* et *"non"* se disent respectivement *"and"*, *"or"* et *"not"* en anglais pour comprendre les lignes de code suivantes.

```
# L'opérateur booléen ET renvoie VRAI uniquement si
# les deux valeurs boléennes valent VRAI, sinon il
# renvoie FAUX.
print("‘and‘ en action...")
print("    True and True   =", True and True)
print("    True and False  =", True and False)
print("    False and True  =", False and True)
print("    False and False =", False and False)

# L'opérateur booléen OU renvoie VRAI uniquement si
# l'une au moins des deux valeurs boléennes est VRAI,
# sinon il renvoie FAUX.
print("")
print("‘or‘ en action...")
print("    True or True   =", True or True)
print("    True or False  =", True or False)
print("    False or True  =", False or True)
print("    False or False =", False or False)

# L'opérateur booléen NON renvoie une valeur booléenne
# "contraire".
print("")
print("‘not‘ en action...")
print("    not True  =", not True)
print("    not False =", not False)
```

Sortie Python

```
‘and‘ en action...
True and True   = True
True and False  = False
False and True  = False
False and False = False

‘or‘ en action...
True or True   = True
True or False  = True
False or True  = True
False or False = False

‘not‘ en action...
not True  = False
not False = True
```

2.7.3 Attention danger ! Calculer avec des booléens, c'est possible.

Le code ci-dessous montre une bizarrerie. Ceci peut se comprendre si l'on sait qu'à l'origine Python avait été construit *"au-dessus"* du langage C qui lui ne dispose pas de type booléen. Du coup dans le langage C on utilise

les entiers `1` et `0` pour symboliser les valeurs de vérité `VRAI` et `FAUX` respectivement. Avec ceci en tête, on peut comprendre l'apparition du `3` ci-dessous.

```
bizzar = 2 + True

print(bizzar)
```

Sortie Python

```
3
```

2.8 Comparer c'est tester

2.8.1 A-t-on égalité ?

Une grande partie des langages de programmation utilisent `=` pour affecter une valeur à une variable, et `==` pour tester une égalité. Python fait partie de ces langages. Notez au passage le manque de clarté en mathématiques où l'on peut écrire *"Soit $x = 4$"* et *"A-t-on $x = 4$?"* dans deux contextes différents.

```
x = 2

print("A-t-on x = 2 ?", x == 2)
print("A-t-on x = 5 ?", x == 5)
```

Sortie Python

```
A-t-on x = 2 ? True
A-t-on x = 5 ? False
```

On note que les tests d'égalité renvoient un résultat de type booléen. Ce sera aussi le cas des tests de comparaison de type *"ordre"* présentés un peu plus bas.

2.8.2 Est-ce différent ?

L'opérateur `!=` teste si deux variables sont différentes. Voici un exemple.

```
x = 2

print("A-t-on x différent 2 ?", x != 2)
print("A-t-on x différent 5 ?", x != 5)
```

Sortie Python

```
A-t-on x différent 2 ? False
A-t-on x différent 5 ? True
```

2.8.3 Plus petit, plus grand et compagnie

Dans le code suivant, remarquez qu'il est possible d'utiliser directement `1 <= x < 3` sans avoir à faire `(1 <= x) and (x < 3)`.

```
x = 2

print("A-t-on x > 0 ?     ", x > 0)
print("A-t-on x > 2 ?     ", x > 2)
print("A-t-on x <= 2 ?    ", x <= 2)
print("A-t-on 1 <= x < 3 ?", 1 <= x < 3)
```

Sortie Python

```
A-t-on x > 0 ?      True
A-t-on x > 2 ?      False
A-t-on x <= 2 ?     True
A-t-on 1 <= x < 3 ? True
```

2.8.4 Attention danger ! `1` et `True` sont égaux mais pas identiques pour Python

Les opérateurs de comparaison peuvent s'appliquer à des types différents de variables. Voici un premier exemple. Le résultat est prévisible car `2` désigne le naturel 2, tandis que `"2"` désigne le texte réduit à l'unique caractère 2.

```
x = 2

print('x == "2" vaut', x == "2")
```

Sortie Python

```
x == "2" vaut False
```

Regardons un code un peu plus problématique.

```
print(1 == True)
```

Sortie Python

```
True
```

La section sur les booléens explique que dans l'implémentation de Python les booléens sont d'une certaine façon des naturels. Ceci permet de comprendre le résultat étrange précédent. Si vous souhaitez vraiment comparer strictement et rigoureusement une variable à une valeur booléenne, il faudra utiliser l'opérateur `is` comme dans le code suivant.

```
print(1 is True)
```

Sortie Python

```
False
```

2.9 Agir si quelque chose est vérifiée

2.9.1 Si ... sinon si ... sinon ...

Programmer c'est prendre des décisions en fonction d'un contexte donné. Dans l'exemple basique suivant, notez l'utilisation de l'indentation, ou du décalage, de quatre espaces qui sert à indiquer le bloc d'instructions à faire en cas de test validé.

```
# Nous souhaitons une réponse de type naturel.
choix = int(input("Votre choix entre 1, 2, et 3 ? "))

if choix == 1:
    print("Premier choix")

elif choix == 2:
    print("Deuxième choix")

elif choix == 3:
    print("Troisième choix")

else:
    print("Choix inconnu")
```

Sortie Python

```
Votre choix entre 1, 2, et 3 ? 7
Choix inconnu
```

Bien entendu, on peut utiliser juste les formes *"réduites"* `if ... else ...` , et aussi juste `if ...` si le contexte le demande *(voir la section suivante)*.

2.9.2 Indentation, attention danger !

L'indentation est importante comme le montre le code suivant.

```
x = 5

# Tout au même niveau.
print("TOUT AU MÊME NIVEAU")
print("==================")


if x == 2:
    print("x == 2")

if x > 2:
    print("x > 2")

if x < 0:
    print("x < 0")

if x <= 5:
    print("x <= 5")
```

```
# Des indentations différentes.
print("")
print("DES INDENTATIONS DIFFÉRENTES")
print("============================")

if x == 2:
    print("x == 2")

    if x > 2:
        print("x > 2")

if x < 0:
    print("x < 0")

    if x <= 5:
        print("x <= 5")
```

Sortie Python

```
TOUT AU MÊME NIVEAU
===================
x > 2
x <= 5

DES INDENTATIONS DIFFÉRENTES
============================
```

2.10 Répéter des actions

2.10.1 Répéter un nombre de fois connu

Dans le code qui suit, la fonction `range` renvoie une plage de valeurs. Voici un premier exemple d'utilisation.

```
for i in range(4):
    print(i)
```

Sortie Python

```
0
1
2
3
```

La convention utilisée pour `range(n)` est de renvoyer les naturels de `0` à `(n - 1)`, et non de `1` à `n`. Mais comment obtenir ce dernier résultat ? Il suffit de faire comme suit où `range(d, f)` renvoie les naturels de `d` à `(f - 1)`.

```
for i in range(1, 5):
    print(i)
```

Sortie Python

```
1
2
3
4
```

Un dernier usage possible consiste à indiquer un pas d'avancement. Le code suivant affiche tous les multiples de 5 entre 0 compris et 35 non compris.

```
for i in range(0, 35, 5):
    print(i)
```

Sortie Python

```
0
5
10
15
20
25
30
```

2.10.2 Répéter un nombre de fois inconnu

Lorsque l'on ne connait pas le nombre de répétitions à effectuer, Python propose la boucle `while`. Voici un exemple typique.

```
reponse = ""

while(reponse != "oui"):
    reponse = input('Taper "oui" sans les guillemets pour partir. ')
```

Sortie Python

```
Taper "oui" sans les guillemets pour partir. o
Taper "oui" sans les guillemets pour partir. NON
Taper "oui" sans les guillemets pour partir. Infernal
Taper "oui" sans les guillemets pour partir. Je vais devoir céder !
Taper "oui" sans les guillemets pour partir. oui
```

2.10.3 Indentation, attention danger !

L'indentation est importante comme nous allons le voir. Considérons le code suivant où l'utilisation maladroite d'une boucle `while` au lieu d'une boucle `for` sert juste à illustrer notre propos.

```
colonne = 0

while(colonne < 3):
    colonne = colonne + 1

    for ligne in range(1, 5):
        print(ligne, ";", colonne)
```

Sortie Python

```
1 ; 1
2 ; 1
3 ; 1
4 ; 1
1 ; 2
2 ; 2
3 ; 2
4 ; 2
1 ; 3
2 ; 3
3 ; 3
4 ; 3
```

Bien que le code suivant change juste quelques indentations du code précédent, les résultats affichés sont très différents.

```
colonne = 0

while(colonne < 3):
    colonne = colonne + 1

for ligne in range(1, 5):
    print(ligne, ";", colonne)
```

Sortie Python

```
1 ; 3
2 ; 3
3 ; 3
4 ; 3
```

2.11 Listes

2.11.1 Créer une liste directement

Les éléments d'une liste sont à mettre entre des crochets séparés par des virgules. Une liste peut contenir différents types de variables comme le montre l'exemple suivant.

```
liste_heterogene = [0, "un", 8/4] # 8/4 = 2.0 est un flottant !

print("liste_heterogene =", liste_heterogene)
print("Type de liste_heterogene :", type(liste_heterogene))
```

Sortie Python

```
liste_heterogene = [0, 'un', 2.0]
Type de liste_heterogene : <class 'list'>
```

Notons que l'on peut faire une liste de listes pour par exemple représenter une liste de coordonnées *(on peut bien entendu aller plus loin en faisant une liste de listes de listes de listes...etc)*.

```
coord_1d = [1]
coord_2d = [0, 10]
coord_3d = [33, -4, 17]

liste_de_coord = [coord_1d, coord_2d, coord_3d]

print(liste_de_coord)
```

Sortie Python

```
[[1], [0, 10], [33, -4, 17]]
```

2.11.2 Ajouter un nouvel élément à une liste

Une liste est une instance de la classe `list` qui possède une méthode `append` permettant d'ajouter un nouvel élément. Voici comment utiliser cette dernière.

```
notre_liste = []

print("Une liste vide au départ :", notre_liste)

notre_liste.append("nouvel élément")
notre_liste.append("autre ajout")

print("Liste remplie maintenant :", notre_liste)
```

Sortie Python

```
Une liste vide au départ : []
Liste remplie maintenant : ['nouvel élément', 'autre ajout']
```

2.11.3 Créer une liste à l'aide de boucles

La méthode `append` présentée dans la section précédente permet de fabriquer facilement certaines listes ayant une structure régulière.

```
naturels = []

for i in range(14):
    naturels.append(i)

print(naturels)
```

Sortie Python

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Il existe un raccourci très pratique basé sur le concept de liste en compréhension. Dans ce qui suit, les crochets nous indiquent que nous fabriquons une liste, puis ensuite cette liste a pour élément les `i` pour `i` parcourant la plage de valeurs renvoyées par `range(14)`.

```
naturels = [i for i in range(14)]

print(naturels)
```

Sortie Python

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

2.11.4 Accoler des listes

L'opérateur + permet d'accoler des listes les unes apès les autres.

```
liste_1 = [1, 2, 3]
liste_2 = [100, 1000]
liste_3 = [9, 99, 999, 9999]

listes_accolees = liste_1 + liste_2 + liste_3

print(listes_accolees)
```

Sortie Python

```
[1, 2, 3, 100, 1000, 9, 99, 999, 9999]
```

2.11.5 Accéder à un élément d'une liste

L'accès à un élément d'une liste se fait comme suit où le premier élément d'une liste a toujours pour position 0. On peut aussi partir de la fin en utilisant un entier relatif négatif, le dernier élément correspondant à la position *(-1)*.

```
naturels_pairs = [0, 2, 4, 6, 8, 10, 12]

print("Premier naturel pair de notre liste   :", naturels_pairs[0])
print("Denier naturel pair de notre liste    :", naturels_pairs[-1])
print("Troisième naturel pair de notre liste :", naturels_pairs[2])
```

Sortie Python

```
Premier naturel pair de notre liste   : 0
Denier naturel pair de notre liste    : 12
Troisième naturel pair de notre liste : 4
```

2.11.6 Extraire une sous-liste

Le code ci-après montre que `[d:f]` va extraire les éléments des positions `d` à *(f - 1)*, et que `[:f]` et `[d:]` sont des raccourcis pour `[0:f]` et `[d:-1]` respectivement. Le fonctionnement est similaire à ce qu'il se fait avec les chaînes de caractères.

```
naturels = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

naturels_avant_4             = naturels[:4]
naturels_entre_4_et_7_inclus = naturels[4:8]
naturels_apres_7             = naturels[8:]

print(naturels_avant_4)
print(naturels_entre_4_et_7_inclus)
print(naturels_apres_7)
```

Sortie Python

```
[0, 1, 2, 3]
[4, 5, 6, 7]
[8, 9, 10, 11, 12, 13, 14]
```

2.11.7 Modifier une liste élément par élément

Il est possible de modifier un élément d'une liste comme suit *(par contre ce type de manipulations ne marche pas avec les chaînes de caractères)*.

```
liste_heterogene = [0, "un", 2.0]

print("liste_heterogene avant :", liste_heterogene)

liste_heterogene[0] = "zéro"
liste_heterogene[2] = "deux"

print("liste_heterogene après :", liste_heterogene)
```

Sortie Python

```
liste_heterogene avant : [0, 'un', 2.0]
liste_heterogene après : ['zéro', 'un', 'deux']
```

2.11.8 Copier une liste, attention danger !

Commençons par quelque chose de déroutant et parfois dur à repérer dans de grands codes.

```
liste_ancienne = [1, 2, 3]
liste_nouvelle = liste_ancienne

liste_nouvelle.append(4)
liste_nouvelle.append(5)

print("liste_ancienne =", liste_ancienne)
print("liste_nouvelle =", liste_nouvelle)
```

Sortie Python

```
liste_ancienne = [1, 2, 3, 4, 5]
liste_nouvelle = [1, 2, 3, 4, 5]
```

Regardez bien ce qu'il vient de se passer. Les **deux** listes ont été modifiées. Si cela ne vous choque pas, comparez avec le code suivant.

```
texte_ancien  = "Texte au départ."
texte_nouveau = texte_ancien

texte_nouveau = "Des changements ont eu lieu."

print("texte_ancien  =", texte_ancien)
print("texte_nouveau =", texte_nouveau)
```

Sortie Python

```
texte_ancien  = Texte au départ.
texte_nouveau = Des changements ont eu lieu.
```

Que s'est il passé avec nos listes, ou peut-être devrait-on dire avec notre liste ? Une liste est un objet informatique qui consomme beaucoup de mémoires, par conséquent les concepteurs de Python ont choisi de limiter la copie en mémoire des listes. Dans notre cas, ceci signifie que les variables `texte_ancien` et `texte_nouveau` pointent vers le même emplacement mémoire d'une unique liste *(ce phénomène se rencontre aussi avec les variables de type ensemble et dictionnaire qui sont présentées dans les sections suivantes)*.
Nous devons donc nous débrouiller pour imposer la création d'une nouvelle copie en mémoire. Voici une méthode propre qui fait appel à un module dédié à la copie d'objets *"mémophages"*. Nous proposons d'utiliser `deepcopy` qui va gérer les cas de listes contenants des listes contenant des listes... etc.

```
from copy import deepcopy

liste_ancienne = [1, 2, 3]
liste_nouvelle = deepcopy(liste_ancienne)

liste_nouvelle.append(4)
liste_nouvelle.append(5)

print("liste_ancienne =", liste_ancienne)
print("liste_nouvelle =", liste_nouvelle)
```

Sortie Python

```
liste_ancienne = [1, 2, 3]
liste_nouvelle = [1, 2, 3, 4, 5]
```

2.11.9 Taille d'une liste

Le mot *"len"* signifiant *"longueur"* en anglais, le code suivant peut se *"deviner"*.

```
une_liste = [1, 2, 3, 4, 5, 6]

print(len(une_liste))
```

Sortie Python

```
6
```

2.12 Ensembles

2.12.1 Des ensembles presque comme en mathématiques

Un ensemble à la sauce Python se comporte pratiquement comme un ensemble fini. Voyons pour commencer comment définir directement de tels ensembles. Notez au passage que les répétitions dans la liste disparaissent dans l'ensemble car un ensemble ne contient que des éléments *"uniques"*. Sachez que les éléments d'un ensemble peuvent être de n'importe quel type.

```
ensemble_vide = set()
print("ensemble_vide =", ensemble_vide)
print("Type de ensemble_vide :", type(ensemble_vide))

ensemble_via_une_liste  = set(['p', 'a', 'p', 'a', 'b', 'é', 'b', 'é', 'g', 'a', 'g', 'a'])
print("ensemble_via_une_liste =", ensemble_via_une_liste)
print("Type de ensemble_via_une_liste :", type(ensemble_via_une_liste))
```

Sortie Python

```
ensemble_vide = set()
Type de ensemble_vide : <class 'set'>
ensemble_via_une_liste = {'g', 'p', 'b', 'a', 'é'}
Type de ensemble_via_une_liste : <class 'set'>
```

2.12.2 Ajouter un nouvel élément à un ensemble

Un ensemble est une instance de la classe `set` qui possède une méthode `add` permettant d'ajouter un nouvel élément. Voici comment utiliser cette dernière.

```
notre_ensemble = set()

print("Un ensemble vide au départ :", notre_ensemble)

notre_ensemble.add("nouvel élément")
notre_ensemble.add("autre ajout")

# Ce qui suit ne doit rien modifier.
notre_ensemble.add("nouvel élément")
notre_ensemble.add("nouvel élément")
notre_ensemble.add("nouvel élément")
notre_ensemble.add("autre ajout")
notre_ensemble.add("autre ajout")

print("Ensemble rempli maintenant :", notre_ensemble)
```

Sortie Python

```
Un ensemble vide au départ : set()
Ensemble rempli maintenant : {'autre ajout', 'nouvel élément'}
```

2.12.3 Créer un ensemble à l'aide de boucles

La méthode `add` vu juste avant peut être utilisée dans une boucle `for` pour construire un ensemble. Voici un exemple un peu mathématique mais pas trop. On construit l'ensemble des naturels obtenus en ne gardant que les deux derniers chiffres du carré d'un naturel. Par exemple, $11^2 = 121$ et $10^2 = 100$ donneront respectivement 21 et 0.

```
# Code non optimal !
ens_deux_derniers_chiffres = set()

for i in range(10**6):
    carre = i**2
    deux_derniers_chiffres = carre%100

    ens_deux_derniers_chiffres.add(deux_derniers_chiffres)

print(ens_deux_derniers_chiffres)
```

Sortie Python

```
{0, 1, 4, 9, 16, 21, 24, 25, 29, 36, 41, 44, 49, 56, 61, 64, 69, 76, 81, 84, 89, 96}
```

2.12.4 Réunion et intersection

L'intérêt du type `set` n'est pas juste d'éliminer les répétitions. Il permet notamment de faire des opérations ensemblistes présentées ci-après.

```
ens_1 = set(["a", "b", "c", 1, 2, 3])
ens_2 = set(["a", "b", 1, 10, 100])

reunion = ens_1.union(ens_2)
intersection = ens_1.intersection(ens_2)

print("reunion =", reunion)
print("intersection =", intersection)
```

Sortie Python

```
reunion = {1, 2, 3, 100, 'b', 'a', 10, 'c'}
intersection = {1, 'b', 'a'}
```

A la place des méthodes `union` et `intersection`, vous pouvez utiliser les opérateurs spéciaux `|` et `&` comme suit.

```
ens_1 = set(["a", "b", "c", 1, 2, 3])
ens_2 = set(["a", "b", 1, 10, 100])

reunion = ens_1 | ens_2
intersection = ens_1 & ens_2

print("reunion =", reunion)
print("intersection =", intersection)
```

Sortie Python

```
reunion = {1, 2, 3, 100, 'b', 'a', 10, 'c'}
intersection = {1, 'b', 'a'}
```

2.12.5 Soustraire d'un ensemble les éléments d'un autre

La méthode à employer est donnée dans le titre comme le montre le code suivant.

```
ens_1 = set(["a", "b", "c", 1, 2, 3])
ens_2 = set(["a", "b", 1, 10, 100])

diff_1_moins_2 = ens_1 - ens_2
diff_2_moins_1 = ens_2 - ens_1

print("diff_1_moins_2 =", diff_1_moins_2)
print("diff_2_moins_1 =", diff_2_moins_1)
```

Sortie Python

```
diff_1_moins_2 = {2, 3, 'c'}
diff_2_moins_1 = {10, 100}
```

2.12.6 Copier un ensemble, attention danger !

Tout comme avec les listes, Python évite autant que possible de copier des ensembles. Ceci permet de comprendre le résulat ci-dessous où les variables `ensemble_ancien` et `ensemble_nouveau` pointent vers le même emplacement mémoire.

```
ensemble_ancien  = set([1, 2, 3])
ensemble_nouveau = ensemble_ancien

ensemble_nouveau.add(4)
ensemble_nouveau.add(5)

print("ensemble_ancien  =", ensemble_ancien)
print("ensemble_nouveau =", ensemble_nouveau)
```

Sortie Python

```
ensemble_ancien  = {1, 2, 3, 4, 5}
ensemble_nouveau = {1, 2, 3, 4, 5}
```

Pour copier un ensemble pour avoir deux versions différentes en mémoire, il suffit de faire appel au module spécialisé `copy`.

```
from copy import deepcopy

ensemble_ancien  = set([1, 2, 3])
ensemble_nouveau = deepcopy(ensemble_ancien)

ensemble_nouveau.add(4)
ensemble_nouveau.add(5)

print("ensemble_ancien  =", ensemble_ancien)
print("ensemble_nouveau =", ensemble_nouveau)
```

Sortie Python

```
ensemble_ancien  = {1, 2, 3}
ensemble_nouveau = {1, 2, 3, 4, 5}
```

2.13 Dictionnaires - Associer des clés à des valeurs

2.13.1 Un dictionnaire Python, c'est quoi !

Un dictionnaire Python sert à associer une clé à une valeur. Voici un exemple d'utilisation.

```
numero_tel = {"James Bond": '007', "M. Arignan": '13.14.09.1515'}

print("numero_tel =", numero_tel)
print("Type de numero_tel :", type(numero_tel))

print("Le numéro de James Bond :", numero_tel["James Bond"])
```

Sortie Python

```
numero_tel = {'James Bond': '007', 'M. Arignan': '13.14.09.1515'}
Type de numero_tel : <class 'dict'>
Le numéro de James Bond : 007
```

Attention ! Les clés doivent avoir des types *"simples"* comme le montre l'exemple suivant. Les chaînes de caractères, les nombres et les booléens ont des types *"simples"*.

```
probleme = {"ok": 'pas ok ensuite', [0, 1]: 4}
```

Sortie Python

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-4012efa7f409> in <module>()
----> 1 probleme = {"ok": 'pas ok ensuite', [0, 1]: 4}
TypeError: unhashable type: 'list'
```

2.13.2 Modifier la valeur d'une clé

Tout se fait très facilement grâce aux crochets. Voici un exemple.

```
numero_tel = {"James Bond": '007', "M. Arignan": '13.14.09.1515'}

print("Avant : numero_tel =", numero_tel)

numero_tel["James Bond"] = "***"

print("Après : numero_tel =", numero_tel)
```

Sortie Python

```
Avant : numero_tel = {'James Bond': '007', 'M. Arignan': '13.14.09.1515'}
Après : numero_tel = {'James Bond': '***', 'M. Arignan': '13.14.09.1515'}
```

2.13.3 Ajouter de nouvelles clés

Tout se fait de nouveau via l'utilisation de crochets comme le montre l'exemple suivant

```
notre_dico = {}

print("Un dictionnaire vide au départ :", notre_dico)

notre_dico["James Bond"] = '007'
notre_dico["M. Arignan"] = '13.14.09.1515'

print("Dictionnaire rempli maintenant :", notre_dico)
```

Sortie Python

```
Un dictionnaire vide au départ : {}
Dictionnaire rempli maintenant : {'James Bond': '007', 'M. Arignan': '13.14.09.1515'}
```

2.13.4 Copier un dictionnaire, attention danger !

Ci-dessous les variables `dico_ancien` et `dico_nouveau` pointent vers le même emplacement mémoire car Python cherche à limiter les copies des objets *"mémophages"*. Ceci permet de comprendre ce qu'il se passe.

```
dico_ancien  = {"un": 1, "deux": 2, "trois": 3}
dico_nouveau = dico_ancien

dico_nouveau["quatre"] = 4
dico_nouveau["cinq"]   = 5

print("dico_ancien  =", dico_ancien)
print("dico_nouveau =", dico_nouveau)
```

Sortie Python

```
dico_ancien  = {'un': 1, 'trois': 3, 'cinq': 5, 'deux': 2, 'quatre': 4}
dico_nouveau = {'un': 1, 'trois': 3, 'cinq': 5, 'deux': 2, 'quatre': 4}
```

Pour copier un dictionnaire afin d'avoir deux versions différentes en mémoire, il suffit de faire appel au module spécialisé `copy`.

```
from copy import deepcopy

dico_ancien  = {"un": 1, "deux": 2, "trois": 3}
dico_nouveau = deepcopy(dico_ancien)

dico_nouveau["quatre"] = 4
dico_nouveau["cinq"]   = 5

print("dico_ancien  =", dico_ancien)
print("dico_nouveau =", dico_nouveau)
```

Sortie Python

```
dico_ancien  = {'un': 1, 'trois': 3, 'deux': 2}
dico_nouveau = {'un': 1, 'trois': 3, 'cinq': 5, 'deux': 2, 'quatre': 4}
```

2.14 Parcourir les textes, les listes, les ensembles et les dictionnaires

2.14.1 Parcourir des textes, des listes et des ensembles

Ces trois types de variable se parcourent en utilisant le même type de syntaxe. Voici un premier exemple d'utilisation via `for ... in ...` . On fera attention au cas d'un ensemble Python : pour ce type de variables, il n'y a pas un ordre prévisible de renvoi des éléments d'un ensemble.

```
# TEXTE
print("Cas d'un texte")
print("--------------")

un_texte = "Un bon départ"

for caractere in un_texte:
    print(caractere)


# LISTE
print("")
print("Cas d'une liste")
print("---------------")

une_liste = [1, 2, 3, 4, 5]

for elt in une_liste:
    print(elt)


# ENSEMBLE
print("")
print("Cas d'un ensemble")
print("-----------------")

un_ensemble = set([3, 2, 1, 3, 2, 1, 3, 2, 1])

for elt in un_ensemble:
    print(elt)
```

Sortie Python

```
Cas d'un texte
--------------
U
n

b
o
n

d
é
p
```

```
a
r
t

Cas d'une liste
---------------
1
2
3
4
5

Cas d'un ensemble
-----------------
1
2
3
```

Ceci donne un moyen simple pour transformer un texte en une liste de lettres respectant l'ordre d'écriture du texte. Pas convaincu*(e)* ? Voici comment faire.

```
texte = "Transformez moi !"

liste_caracteres = [caractere for caractere in texte]

print(liste_caracteres)
```

Sortie Python

```
['T', 'r', 'a', 'n', 's', 'f', 'o', 'r', 'm', 'e', 'z', ' ', 'm', 'o', 'i', ' ', '!']
```

Utile : le code suivant fonctionne car les textes, comme les listes, peuvent être parcourus via `for ... in ...` .

```
texte = "caractères utilisées"

caracteres_utilises = set(texte)

print(caracteres_utilises)
```

Sortie Python

```
{' ', 'r', 'e', 's', 'é', 'è', 't', 'c', 'u', 'a', 'i', 'l'}
```

2.14.2 Parcourir un texte, une liste ou un ensemble en *"récupérant la position"*

Il peut être utile de récupérer à la fois une lettre ou un élément avec sa position. Dans ce cas, on utilisera `for ... in enumerate(...)` comme le montre le code qui suit.

```python
# TEXTE
print("Cas d'un texte")
print("--------------")

un_texte = "Un bon départ"

for position, lettre in enumerate(un_texte):
    print(position, ":", lettre)


# LISTE
print("")
print("Cas d'une liste")
print("---------------")

une_liste = [1, 2, 3, 4, 5]

for position, elt in enumerate(une_liste):
    print(position, ":", elt)


# ENSEMBLE
print("")
print("Cas d'un ensemble")
print("-----------------")

un_ensemble = set([3, 2, 1, 3, 2, 1, 3, 2, 1])

for position, elt in enumerate(un_ensemble):
    print(position, ":", elt)
```

Sortie Python

```
Cas d'un texte
--------------
0 : U
1 : n
2 :
3 : b
4 : o
5 : n
6 :
7 : d
8 : é
9 : p
10 : a
11 : r
12 : t

Cas d'une liste
---------------
0 : 1
```

```
1 : 2
2 : 3
3 : 4
4 : 5

Cas d'un ensemble
-----------------
0 : 1
1 : 2
2 : 3
```

2.14.3 Parcourir un dictionnaire

Le parcours des dictionnaires peut se faire de deux façons possibles présentées dans l'exemple ci-après. Tout comme pour les ensembles, il n'y a pas d'ordre prévisible de renvoi des clés.

```
un_dico = {0: "zéro", 1: "un", 2: "deux"}

print("Méthode 1")
print("---------")

for cle in un_dico:
    print(cle, "s'écrit", un_dico[cle],".")


print("")
print("Méthode 2")
print("---------")

for cle, valeur in un_dico.items():
    print(cle, "s'écrit", valeur,".")
```

Sortie Python

```
Méthode 1
---------
0 s'écrit zéro .
1 s'écrit un .
2 s'écrit deux .

Méthode 2
---------
0 s'écrit zéro .
1 s'écrit un .
2 s'écrit deux .
```

2.15 Tests liés aux textes, aux listes, aux ensembles et aux dictionnaires

2.15.1 Cas des textes

Pour tester la présence d'un texte dans un autre, on procède comme suit où le mot *"in"* signifie *"dans"* en anglais. Dans ce type de test, les majuscules et les minuscules sont concisédérées comme différentes !

```
var_texte = "Testons un peu le cas d'un texte."

print('var_texte a pour contenu :')
print(var_texte)

a_dans_texte = "a" in var_texte   # a_dans_texte est de type booléen.
z_dans_texte = "z" in var_texte

print("")
print('Le texte "a" apparaît-il dans le contenu de var_texte ?', a_dans_texte)
print('Le texte "z" apparaît-il dans le contenu de var_texte ?', z_dans_texte)

cas_dans_texte  = "cas" in var_texte
test_dans_texte = "test" in var_texte
Test_dans_texte = "Test" in var_texte

print("")
print('Le texte "cas" apparaît-il dans le contenu de var_texte ? ', cas_dans_texte)
print('Le texte "test" apparaît-il dans le contenu de var_texte ?', test_dans_texte)
print('Le texte "Test" apparaît-il dans le contenu de var_texte ?', Test_dans_texte)
```

Sortie Python

```
var_texte a pour contenu :
Testons un peu le cas d'un texte.

Le texte "a" apparaît-il dans le contenu de var_texte ? True
Le texte "z" apparaît-il dans le contenu de var_texte ? False

Le texte "cas" apparaît-il dans le contenu de var_texte ?  True
Le texte "test" apparaît-il dans le contenu de var_texte ? False
Le texte "Test" apparaît-il dans le contenu de var_texte ? True
```

2.15.2 Cas des listes

Ce qui suit est simple à comprendre car très similaire à ce qui a été proposé pour les chaînes de caractères.

```
premiers = [2, 3, 5, 7, 11, 13]

texte_deux_dans_premiers      = "2" in premiers
entier_deux_dans_premiers     = 2 in premiers
entier_dix_sept_dans_premiers = 17 in premiers

print('Le texte "2" est-il dans la liste ?', texte_deux_dans_premiers)
print("L'entier 2 est-il dans la liste ?  ", entier_deux_dans_premiers)
print("L'entier 17 est-il dans la liste ? ", entier_dix_sept_dans_premiers)
```

Sortie Python

```
Le texte "2" est-il dans la liste ? False
L'entier 2 est-il dans la liste ?   True
L'entier 17 est-il dans la liste ?  False
```

2.15.3 Cas des ensembles

Pour commencer, rien de nouveau par rapport aux sections précédentes.

```
premiers = set([2, 3, 5, 7, 11, 13])

texte_deux_dans_premiers      = "2" in premiers
entier_deux_dans_premiers     = 2 in premiers
entier_dix_sept_dans_premiers = 17 in premiers

# Norez au passage l'utilisation de \' à l'intérieur de
# '...' pour pouvoir utiliser le caractère '.
print('Le texte "2" est-il dans l\'ensemble ?', texte_deux_dans_premiers)
print("L'entier 2 est-il dans l'ensemble ?  ", entier_deux_dans_premiers)
print("L'entier 17 est-il dans l'ensemble ? ", entier_dix_sept_dans_premiers)
```

Sortie Python

```
Le texte "2" est-il dans l'ensemble ? False
L'entier 2 est-il dans l'ensemble ?   True
L'entier 17 est-il dans l'ensemble ?  False
```

On peut aussi comparer des ensembles entre eux. Ci-dessous, nous avons juste montré l'emploi de `... <= ...` pour *"est inclus dans ou égal à"*. Il est bien entendu aussi possible d'utiliser `... < ...` pour tester une inclusion stricte, ainsi que les écritures symétriques `... >= ...` et `... > ...` .

```
ens_1 = set([1, 2, 3, 4])
ens_2 = set([2, 4])
ens_3 = set([1, 2, 7])

ens_1_inclus_dans_ens_1 = ens_1 <= ens_1
ens_2_inclus_dans_ens_1 = ens_2 <= ens_1
ens_3_inclus_dans_ens_1 = ens_3 <= ens_1

print("L'ensemble ens_1 est-il inclus dans l'ensemble ens_1 ?", ens_1_inclus_dans_ens_1)
print("L'ensemble ens_2 est-il inclus dans l'ensemble ens_1 ?", ens_2_inclus_dans_ens_1)
print("L'ensemble ens_3 est-il inclus dans l'ensemble ens_1 ?", ens_3_inclus_dans_ens_1)
```

Sortie Python

```
L'ensemble ens_1 est-il inclus dans l'ensemble ens_1 ? True
L'ensemble ens_2 est-il inclus dans l'ensemble ens_1 ? True
L'ensemble ens_3 est-il inclus dans l'ensemble ens_1 ? False
```

2.15.4 Cas des dictionnaires

Avec les dictionnaires, on peut tester s'ils *"contiennent"* une clé donnée.

```
dico = {"inspiration": 0, "automatisme": 10**80}

cle_inspiration_dans_dico = "inspiration" in dico
cle_erreur_dans_dico      = "erreur" in dico

print('dico contient-il la clé "inspiration" ?', cle_inspiration_dans_dico)
print('dico contient-il la clé "erreur" ?     ', cle_erreur_dans_dico)
```

Sortie Python

```
dico contient-il la clé "inspiration" ? True
dico contient-il la clé "erreur" ?      False
```

2.16 Organiser son code - Les fonctions

2.16.1 Scinder son code en petits bouts

Organiser un code est une tache importante, et pas toujours simple. L'une des techniques consistent à définir des fonctions qui agissent de façon précise sur des arguments donnés. Une fonction peut se définir comme suit *(nous ne faisons qu'effleurer les choses ici)*.

```
def puissance(x, n):
    return x**n

def carre(x):
    return puissance(x, 2)

def cube(x):
    return puissance(x, 3)

print("Carré de 2 :", carre(2))
print("Cube de 5  :", cube(5))
```

Sortie Python

```
Carré de 2 : 4
Cube de 5  : 125
```

2.16.2 Indentation, attention danger !

L'indentation est importante comme le montre le code suivant où la fonction `cube_test` n'existe qu'au sein de la fonction `carre_test`. Notez au passage la possibilité de définir des fonctions dans le code d'une fonction, une technique qui a des applications très concrètes via un outil très puissant que sont les décorateurs Python *(ceci ne sera pas du tout abordé dans ce document)*.

```
def carre_test(x):
    def cube_test(x):
        return x**3

    return x**2
```

```
# Fonction carre_test utilisable.
print("Carré de 2 :", carre_test(2))

# Fonction cube_test inconnue en dehors de carre_test.
print("Cube de 5  :", cube_test(5))
```

Sortie Python

```
Carré de 2 : 4
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-39-7ef84e534a61> in <module>()
      9
     10 # Fonction cube_test inconnue en dehors de carre_test.
---> 11 print("Cube de 5  :", cube_test(5))
NameError: name 'cube_test' is not defined
```

2.16.3 Portée des variables

Il peut arriver que l'on souhaite modifier une variable au sein d'une fonction en gardant cette nouvelle valeur dans la suite de l'exécution du code *(dans ce type de situation, la programmation orienté objet devient une technique très efficace, mais ceci ne sera pas abordé dans ce document)*. Commençons par un code ne fonctionnant pas.

```
x = 5

def modifie_x():
    x = x + 10

print("Avant : x =", x)
modifie_x()
print("Après : x =", x)
```

Sortie Python

```
Avant : x = 5
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-40-87436836dfd5> in <module>()
      5
      6 print("Avant : x =", x)
----> 7 modifie_x()
      8 print("Après : x =", x)
<ipython-input-40-87436836dfd5> in modifie_x()
      2
      3 def modifie_x():
----> 4     x = x + 10
      5
      6 print("Avant : x =", x)
UnboundLocalError: local variable 'x' referenced before assignment
```

Le message indique que la variable `x` est inconue du point de vue de la fonction `modifie_x`. Le mot clé `global` permet d'indiquer à une fonction une variable existant globalement. On obtient ainsi le code suivant qui fait ce qui est attendu.

```
x = 5

def modifie_x_avec_global():
    global x
    x = x + 10

print("Avant : x =", x)
modifie_x_avec_global()
print("Après : x =", x)
```

Sortie Python

```
Avant : x = 5
Après : x = 15
```

Indiquons que pour des variables *"mémophages"* comme les listes, les ensembles et les dictionnaires, il peut y avoir des comportements bizarres comme le montre le code suivant *(voir ce qui a été dit à propos de la copie des listes, des dictionnaires ou des ensembles)*.

```
# Une méthode qui marche.
liste = ["et un", "et deux"]

def modifie_liste_accepte():
    liste.append("et trois zéro")

print("modifie_liste_accepte - Avant")
print("============================")
print("liste =", liste)

modifie_liste_accepte()
print("")

print("modifie_liste_accepte - Après")
print("============================")
print("liste =", liste)


# Une méthode qui pose problème.
liste = ["et un", "et deux"]

def modifie_liste_problematique():
    liste = liste + ["et trois zéro"]

print("")
print("modifie_liste_problematique - Avant")
print("==================================")
print("liste =", liste)

modifie_liste_problematique()
print("")
print("modifie_liste_problematique - Après")
```

```
print("==================================")
print("liste =", liste)
```

Sortie Python

```
modifie_liste_accepte - Avant
=============================
liste = ['et un', 'et deux']

modifie_liste_accepte - Après
=============================
liste = ['et un', 'et deux', 'et trois zéro']

modifie_liste_problematique - Avant
===================================
liste = ['et un', 'et deux']
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-42-0d911c15b216> in <module>()
     28 print("liste =", liste)
     29
---> 30 modifie_liste_problematique()
     31 print("")
     32 print("modifie_liste_problematique - Après")
<ipython-input-42-0d911c15b216> in modifie_liste_problematique()
     21
     22 def modifie_liste_problematique():
---> 23     liste = liste + ["et trois zéro"]
     24
     25 print("")
UnboundLocalError: local variable 'liste' referenced before assignment
```

Expliquons rapidement le problème rencontré ici. Lorsque Python exécute le code, il rencontre `liste = ...` or il n'existe pas de variable `liste` du point de vue de la fonction. Ceci aboutit alors à l'erreur *"local variable 'liste' referenced before assignment"* soit *"variable locale 'liste' referencé avant affectation"*. Le code suivant montre que c'est bien l'exécution du code qui renvoie une erreur et non son analyse syntaxique.

```
liste = ["et un", "et deux"]

def modifie_liste_problematique():
    print("Je suis affiché si le code est exécuté.")
    liste = liste + ["et trois zéro"]

modifie_liste_problematique()
```

Sortie Python

```
Je suis affiché si le code est exécuté.
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-43-982a4a6ce25f> in <module>()
      5     liste = liste + ["et trois zéro"]
      6
```

```
----> 7 modifie_liste_problematique()
<ipython-input-43-982a4a6ce25f> in modifie_liste_problematique()
      3 def modifie_liste_problematique():
      4     print("Je suis affiché si le code est exécuté.")
----> 5     liste = liste + ["et trois zéro"]
      6
      7 modifie_liste_problematique()
UnboundLocalError: local variable 'liste' referenced before assignment
```

Conseil : utilisez `global` dès que vous souhaitez agir globalement, et sinon utilisez un nom *"non global"* pour une variable utilisée uniquement dans le code d'une fonction.