Kit de démarrage en Python

12 septembre 2021

I Introduction

1) Les quatre piliers de l'informatique

L'informatique repose sur quatre piliers :

- la machine
- l'information
- l'algorithmique
- le langage

Voici quelques questions soulevées :

- Qu'est-ce qu'un ordinateur ? Un réseau ? Un robot ?
- Comment représente-t-on une image, un son, un nombre, un texte dans un ordinateur ?

  L'exemple du codage d'un entier en base 2.
- Comment trier une liste de manière efficace ?

  Comment calculer le pgcd de deux nombres ? Décomposition primaire Vs algorithme d'Euclide
- Les langages reposent sur les instructions fondamentales suivantes : affectation, séquence, tests conditionnels, itération

2) Un peu d'histoire et de culture

Premières automatisations : machine à calculer (Pascaline 1645), machine à tisser (Jacquard 1801 avec des cartes perforées), premiers ordinateurs ENIAC (1943)

Frise chronologique des langages :

Assembleur, Fortran (1954), Lisp (1958), Cobol (1960) (applications gestions), Basic (1964), C (1970), SQL (1978) (langage de requêtes), C++ (1985), Python (1990), Java (1995), PHP (1995), $C\sharp$ (2000).

II Petite présentation

Python[1] est un langage de programmation créé dans les années 1990 par l'informaticien néerlandais Guido Van Rossum[2]. Citons quelques-une de ses caractéristiques :

- langage de script, c'est-à-dire interprété par opposition à langage compilé
- orienté objet
- multiplateforme (linux, windows, mac os, android,...)
- typage dynamique
- libre et gratuit
- doté de nombreuses librairies

C'est un véritable langage de programmation utilisé dans l'industrie, par des entreprises telles que Google, Youtube, EDF, NASA, CEA (commissariat à l'énergie atomique)... Odoo (une «Enterprise Ressource Planning» pour la gestion des entreprises) ou Mercurial sont des applications entièrement développées en Python.

1) Version 3.X Vs Version 2.X

Quelques différences :

- les entiers ne sont pas bornés en version 3.
- `print` est une fonction en version 3
- la fonction `raw_input` de la version 2 est remplacé par `input` en version 3.
- le symbole / est en version 3, celui de la division flottante.

2) Pour démarrer

On peut utiliser Python de deux manières différentes :

- de manière intéractive avec la console (shell en anglais), on tape une instruction et on obtient directement la réponse de Python. Pratique pour découvrir de nouvelles fonctions
- en tapant l'ensemble des instructions dans un fichier (script) que l'on sauvera avec l'extension .py puis que l'on exécutera avec l'exécutable python.exe.

Nous utiliserons un IDE, en anglais Integrated Development Environment qui signifie «Environnement de développement intégré», qui permet de faire tout ceci. Notre choix s'est porté sur IEP, qui est directement intégré à la distribution Pyzo que l'on téléchargera.

1. Le nom Python est un hommage à la série anglaise des Monty Python.
2. Après avoir travaillé chez Google de 2005 à 2012, Guido vient de rejoindre l'entreprise Dropbox.

III Une introduction possible : la «calculatrice» Python

Les trois chevrons >>> sont une invite de commande (prompt en anglais), ils signifient que Python est prêt à exécuter une commande.

1) Les cinq opérations

On peut utiliser Python comme une calculatrice. L'addition, la soustraction, la multiplication, la division et l'exponentiation (calculer la puissance d'un nombre) sont notées respectivement : $+, -, *, /$ et $**$. Les règles de calcul (parenthèses et priorités opératoires) sont bien sûr celles que nous connaissons.

Quelques exemples :

```
48 + 23   # le symbole dièse précède un commentaire
48 * 23
6 - 4 * 8
(6 - 4) * 8
45.3 * 21.2
30 / 2     # attention, comportement différent sous Python 2
64 / 5
5 ** 3 # calcule 5 puissance 3
45 ** 78
```

Remarquons que les entiers en Python n'ont pas de taille limite (ce n'est pas le cas en C ou en Java où les entiers sont codés sur 64 bits et sont donc inférieurs à $2^{64} - 1$).

Le résultat de la division `30 / 2` donné par Python [3] peut paraître surprenant. Python renvoie `15.0`, et non pas `15`. De même Python renvoie `1.0` comme résultat de `2 * 0.5`. Les instructions suivantes nous éclairent :

```
>>> 1 == 1.0  # on teste l'égalité entre 1 et 1.0
True
>>> type(1), type(1.0)
<class 'int'>, <class 'float'>
>>> type(1) == type(1.0)  # 1 et 1.0 ont des types différents
False
```

Les nombres `1` et `1.0` ont bien la même valeur mais sont deux objets de **type** différent : le premier est un entier (integer en anglais) tandis que le second est un nombre flottant (float en anglais). Ils ne sont pas représentés de la même manière en mémoire [4]. Si l'on fait des opérations avec un flottant, le résultat sera un flottant. Par exemple, `5 * 2.0` renvoie `10.0`. La division avec `/` est une division flottante. Pour effectuer, `30 / 2` Python convertit `30` et `2` en nombres flottants (par exemple `float(30)`) et renvoie pour résultat un flottant.

Un dernier petit calcul sur les flottants qui laisse augurer certaines difficultés que nous soulèverons dans le chapitre **??** «Représentation numérique de l'information».

3. pour la version Python 3.X. Attention sous Python 2.X, il renverra bien 15.
4. Nous détaillerons ces représentations dans le chapitre Représentation numérique de l'information.

```
>>> 3 * 0.1
0.30000000000000004
>>> 3 * 0.1 == 0.3
False
```

2) Division euclidienne et opérateur modulo

Dans 23, il y a 3 fois 7, et il reste 2. C'est la division du primaire, appelé division euclidienne que l'on peut écrire $23 = 3 \times 7 + 2$. L'entier 3 est le quotient et l'entier 2 le reste. Python sait faire cette division :

```
>>> 23 // 7
3
>>> 23 % 7
2
```

Cette fois ci, le résulat renvoyé par la division de 30 par 2 est bien l'entier 15.

```
>>> 30 // 2
15
```

La division euclidienne est très utile pour tester la divisibilité d'un entier par un autre entier. Par exemple 2047 est divisible par 23 car le reste de la division de 2047 par 23 vaut 0 :

```
>>> 2047 % 23
0
```

On dit aussi que 2047 est égal à 0 modulo 23, c'est pourquoi l'opérateur `%` est aussi appelé opérateur `modulo`. En particulier, un entier pair est un entier égal à 0 modulo 2 et un entier impair est un entier égal à 1 modulo 2.

3) D'autres fonctions ? Importation de modules

Comment calculer la racine carrée de 9 (en anglais, «square root of 9») ? La fonction `sqrt` le permet. Notre permier essai est un échec.

```
>>> sqrt(9)
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    sqrt(5)
NameError: name 'sqrt' is not defined
```

En fait la fonction `sqrt` n'est pas directement disponible, elle fait partie d'un **module** intitulé math que l'on doit importer. Pour cela, on peut taper

```
>>> import math
>>> math.sqrt(9)
3.0
```

Remarquer le préfixe math.

Quelles sont les fonctions du module math ? On peut les afficher à l'aide de `help(math)`. Voici quelques exemples :

```
>>> math.exp(45)
3.4934271057485095e+19
>>> math.factorial(4) # calcule 4!= 1*2*3*4
24
>>> math.floor(45.3) # arrondi à l'entier inférieur
45
>>> math.ceil(45.3) # arrondi à l'entier supérieur
46
```

Remarques :

- les fonctions partie entière `floor` (signifie en français «sol») et partie entière supérieure `ceil` (en anglais «ceiling» est le plafond) sont importantes et à retenir.
- remarquer la valeur renvoyé pour e^{45} : c'est un nombre flottant 3.4934271057485095e+19, qui correspond à l'écriture scientifique $3.4934271057485095 \times 10^{19}$.

Python peut aussi générer des nombres aléatoires. Il faut pour cela importer le module `random`.

```
>>> import random
>>> random.randint(1,6)  # génère un entier compris entre 1 et 6
1
>>> random.random() # génère un flottant compris entre 0 et 1
0.6469912430701616
```

IV L'affectation et les variables

1) Introduction

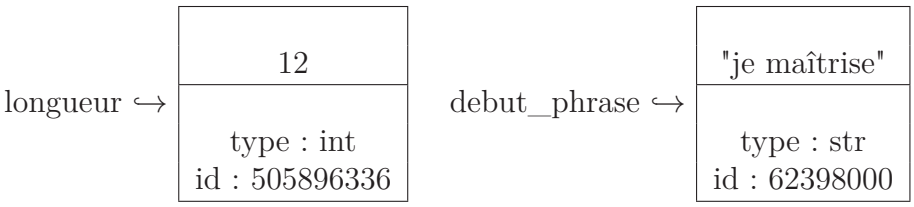
On peut mémoriser la valeur des objets dans des variables dont le nom peut-être constitué de lettres, de chiffres et du caractère spécial «souligné». Attention, le nom ne doit pas commencer par un chiffre et Python est sensible à la casse, il différencie les lettres minuscules et majuscules. Le symbole d'affectation est =. Par exemple,

```
longueur = 12
largeur = 10
aire = longueur * largeur
```

```
print(aire)

debut_phrase ="je maîtrise"
fin_phrase = "le python"
print(debut_phrase + ' ' +  fin_de_phrase) # l'opérateur + concatène les chaînes de cara
print(fin_de_phrase + ' ' + debut_phrase )
```

Remarquons que contrairement à certains langages (C, java...), il n'est pas nécessaire d'indiquer à Python le type de l'objet, on parle de **typage dynamique**. Dans le premier exemple, 12 et 10 sont des objets de type int, tandis que `"je maîtrise"` et `"le python"` sont de type string (chaîne de caractère). Selon le type d'objet, la quantité de mémoire allouée pour le stockage sera différente. On peut connaître cette adresse mémoire à l'aide d'un **identifiant** (id) : l'instruction `id(longueur)` nous la renvoie, par exemple 505896336. On peut schématiser par



2) Les principaux types de variables

Les principaux types d'objets que nous utiliserons sont :

1. Types de base :
   - nombres : entiers (int), flottants (float)
   - booléen (bool)
2. Types structurés :
   - chaînes de caractère (str), tuples (tuple) qui sont des objets non mutables
   - listes (list), objet mutable
   - le dictionnaire (dict)

3) L'affectation n'est pas une égalité

Deviner la valeur de **b** à la fin du script.

```
>>>a = 128
>>> a, id(a)
(128, 505896336)
>>> b = 2* a
>>> b, id(b)
(256, 505898384)
>>> a = 0
```

La réponse est donnée par le script suivant

```
>>> id(a)
505894288
>>> b,id(b)
(256, 505898384)
```

Explication : il y a deux étapes dans la phase d'affectation `b = 2 *a` : l'expression `2*a` placée après le symbole `=` est d'abord évaluée par Python, c'est un objet de **type** int dont la **valeur** va être mémorisée et associée à la variable `b`. Lorsque que l'on a modifié la valeur de `a` par `a = 0`, nous n'avons pas modifié la valeur de `b` qui vaut toujours 256 et non pas le double de `a`. Remarquons enfin que lorsque l'on a modifié la valeur de `a`, sa valeur a été mémorisée dans une autre case mémoire (l'id de `a` a été aussi modifié).

4) L'affectation sous Python est un étiquetage

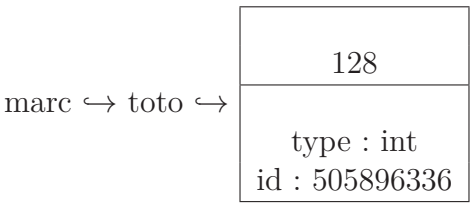
Ce qui suit est extrêmement important : considérons les affectations

```
>>> toto = 128
>>> marc = toto
>>> print(toto,marc)
128 128
```

On a créé une nouvelle variable dont la valeur est bien sûr la même que celle de toto. Mais que s'est-il passé au niveau de la machine ? Deux comportements sont possibles, cela varie selon le langage de programmation :

- l'affectation est une «copie», et dans ce cas, l'entier 128 a été mémorisé une seconde fois à une adresse mémoire différente. C'est ce qui ce produit par exemple en langage C.
- l'affectation est un «étiquetage». Il n'a pas été réalisé de copie de l'entier 128, on a juste crée un nouveau nom `marc` (une nouvelle étiquette) qui pointe tout comme `toto` vers le même objet, l'entier 128.

En Python, c'est ce deuxième comportement qui se produit.



La variable `marc` est un étiquetage de l'objet pointé par `toto`, donc un nouvel étiquetage de l'objet 128. La variable `marc` a ainsi la même valeur mais aussi la même référence (ou id) que `toto`.

Vérifions avec

```
>>> id(toto), id(marc)
(505896336, 505896336)
```

On dit que `marc` et `toto` appartiennent à **l'espace des noms**, tandis que l'entier 128 appartient à l'**espace des objets**. Si l'objet occupe une grande quantité de mémoire, par exemple une liste de très grande taille, ce système d'étiquetage permet d'économiser de la mémoire.

Nous reviendrons sur cette problématique lorsque nous essaierons de fabriquer des copies de listes qui sont des objets mutables.

On pourra consulter avec profit l'excellent site suivant `http://www.pythontutor.com/` qui permet de «tracer» les programmes.

5) Quelques exemples pour terminer

```
>>> a = 5
>>> del(a) # écrase la variable a
>>> a,b = 4,5 # affectations multiples
>>> x = y = z = 1
```

Comment échanger le contenu de deux variables ? On peut utiliser une troisième variable nommé `temp` comme temporaire.

```
>>> x = 5
>>> y = 6
>>> temp = x
>>> x = y
>>> y =temp
>>> print((x,y))
(6,5)
```

Signalons enfin aussi qu'il est possible de dépaqueter des tuples (tuple unpacking) `x, y, z = a, b, c`. Cela permet notamment «la pythonnerie» suivante pour permuter le contenu de deux variables

```
x = 5
y = 6
x,y = y,x
```

Interprétation : python évalue le couple d'objets `y,x` qui vaut 6,5 et l'affecte au couple de variables `x,y` (un tuple de noms).

V On peut créer ses propres fonctions

1) La syntaxe

Nous avons déjà vu des fonctions comme `print` ou comme `math.floor`. On peut bien sûr créer ses propres fonctions, par exemple :

```
>>> def aire_rectangle(largeur,longueur):
        return largeur * longueur

>>> print(aire_rectangle(4,5))
20
```

Le mot clé `def` indique que l'on crée une fonction nommée `aire_rectangle`, qui renvoie l'aire d'un rectangle dont les côtés mesurent `longueur` et `largeur` (paramètres de la fonction). On remarquera les «deux points :» qui terminent la première ligne, l'indentation (décalage du texte vers la droite) qui délimite le corps de la fonction et enfin le mot clé `return` qui signifie «renvoie».

Remarques en vrac :

- Contrairement à des langages comme C ou Java, il n'est pas nécessaire de préciser le type des paramètres d'une fonction ni le type des valeurs qu'elle renvoie. Python le fait par lui-même, on parle de typage dynamique. C'est pratique, mais cela peut poser des problèmes. Par exemple,

  ```
  >>> def double(x):
       return(2*x)
  >>> double(5), double(3.1), double("to")
  (10, 6.2, 'toto')
  >>> double(5) + double(3.1)
  16.2
  ```

  On voit que l'argument `x` de la fonction `double` a été successivement un entier (`5`), un flottant (`3.1`) et une chaîne de caractère (`"to"`).

- On peut toutefois vérifier une condition exigée sur l'argument à l'aide de l'instruction `assert`. Par exemple assert (type(x) == int) provoquera une erreur si x n'est pas de type entier.

- après un `return`, l'exécution de la fonction est interrompue, les éventuelles instructions après un `return` ne seront pas lues par l'interpréteur.

Par exemple, la fonction suivante est équivalente à la précédente.

```
>>> def double(x):
     return(2*x)
     return(3*x)
>>> double(5)
10
```

2) Fonction Vs Procédure : Renvoyer $\neq$ Afficher

Considérons maintenant une autre fonction et expliquons le message d'erreur final

```
>>> def double_2(x):
     print(2*x)
>>> double_2(5)
>>> double_2(5) + double_2(3.1)
```

Moralité, **attention afficher n'est pas renvoyer**. La fonction `double_2` ne renvoie rien (elle renvoie l'objet `None`), comme le confirme `type(double_2(5))` qui donne `NoneType`. On parle parfois alors de procédure à la place de fonction.

Une fonction vaut quelque-chose, une procédure fait quelque-chose.

3) Notice d'une fonction

On peut rédiger une «notice d'utilisation» d'une fonction dans le corps d'une fonction à l'aide des délimitateurs «triple quote».

```
>>> def hypotenuse(adjacent,oppose):
     """ Calcule la longueur de l'hypoténuse d'un triangle rectangle.
     adjacent et oppose sont les longueurs des côtés de l'angle droit.
     return  math.sqrt(adjacent **2 + oppose**2)"""
>>> hypotenuse(3,4)
```

On peut lire la notice en tapant `help(hypotenuse)`.

4) Portée des variables

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.

```
>>> a = 5
>>> def f(x):
    a = 10
    return x + a
>>> f(1), a
(11, 5)
```

```
>>> a = 5
>>> def f(x):
    return x + a
>>> f(1), a
(6, 5)
```

5) Appel de fonctions par valeurs

L'exécution de $f(x)$ évalue d'abord x puis exécute f avec la valeur calculée. On détaillera ce point lorsque x sera une liste, exemple d'objet mutable.

VI Intéraction avec l'utilisateur : fonctions input et print (hors-programme)

Apprenons à utiliser la fonction `input` qui prend en paramètre une chaîne de caractères :

```
>>> nom = input("Entrez votre prénom: ")
Entrez votre prénom: Paul
>>> print("Bonjour,", nom)
Bonjour, Paul
```

Un deuxième exemple déconcertant puisque 4545 n'est pas le double de 45.

```
>>> nombre = input("Donne un entier: ")
Donne un entier: 45
>>> print("le double du nombre est:", 2 * nombre)
le double du nombre est: 4545
```

Où est le problème ? Voici de quoi comprendre :

```
>>> 3* nombre
'454545'
>>> type(nombre)
<class 'str'>
```

L'objet pointé par `nombre` n'est pas l'entier `45` mais la chaîne de caractère `'45'`. La fonction input renvoie des chaînes de caractères.

L'opération `2 * nombre` n'est donc pas une multiplication entre deux nombres mais l'opération de **concaténation** entre deux chaînes. Quelques exemples pour comprendre :

```
>>> nombre + nombre
'4545'
>>> "toto" + " " + "vient"
'toto vient'
```

Bien, mais comment récupérer l'entier 45 et non pas la chaîne de caractère ? Et bien on va convertir le type à l'aide de la fonction `int`.

```
>>> nombre = input("Donne un entier: ")
Donne un entier: 45
>>> nombre = int(nombre)
>>> print("le double du nombre est:", 2 * nombre)
le double du nombre est: 90
```

Selon les besoins, on peut convertir en nombre flottant à l'aide de la fonction `float` :

```
>>> nombre = input("Quelle est ta taille (en m)? ")
Quelle est ta taille (en m)? 1.80
>>> nombre = float(nombre)
>>> print("le double de ta taille est:", 2 * nombre)
le double de ta taille est: 3.6
```