COURS PYTHON

July 26, 2024

1 Sommaire

> **Rappels : programmes et instructions** - en Python un programme est une suite d'*instructions* - les instructions sont exécutées une à une *séquentiellement*, c'est-à-dire dans l'ordre où elles sont écrites sauf si on utilise une *structure de contrôle*, à savoir une *condition*, une *boucle* ou une *fonction*.

2 Présentation du langage

Retour au sommaire

Caractéristiques de Python

- existe en deux versions incompatibles (2 et 3) : ici on parle de Python 3
- c'est un langage orienté objet, qu'on utilisera essentiellement comme un langage impératif dans le cadre du programme de prépa.
- C'est un langage interprété : la traduction du programme en langage machine est réalisée avant *chaque* exécution
- Le typage est dynamique (voir plus loin)

Avantages

- facile à apprendre (syntaxe facile)
- portable (s'exécute sur tous les systèmes d'exploitation : Linux, Mac, Windows)
- gratuit
- il existe des modules pour faire à peu près n'importe quoi avec Python

Inconvénients

- lent par rapport aux langages compilés (C, Java)
- les facilités des débuts se payent sur les plus gros programmes

Utilisation dans le monde professionnel

- Interface avec des langages rapides pour le calcul scientifique, notamment pour l'IA
- prototypage de gros projets
- automatisation de tâches d'administration
- développement d'applications
- deuxième langage le plus utilisé dans le monde après javascript en 2020

Trois manières d'utiliser le langage

- en mode interactif dans une console Python (pratique pour les petits tests)
- en interprétant des scripts (des fichiers texte qui stockent les programmes)
- en combinant ces deux approches dans un notebook jupyter

Dans **Capytale**, on peut utiliser des scripts ou des notebook.

3 Variables, types et opérateurs

Retour au sommaire

3.1 Variables et types de base

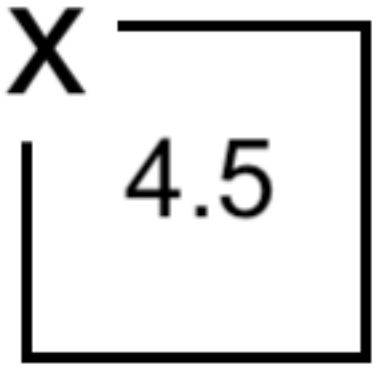
- une variable = un nom auquel on associe une valeur : cette opération s'appelle une *affectation*.
- nécessaire comme en maths pour augmenter le niveau d'abstraction : on va manipuler des noms (x par exemple) plutôt que des valeurs numériques
- l'étape d'abstraction suivante est la définition de fonctions (voir plus loin)

En Python, on déclare le type de la variable implicitement au moment où on lui affecte une valeur : le typage est *dynamique*. Attention le symbole `=` est celui de *l'opérateur d'affectation* (le test d'égalité sera vu plus loin). Il faut penser le symbole `=` comme l'ordre de placer une *valeur* (opérande de droite) dans une case mémoire (une portion de la RAM) qui porte *le nom d'une variable* (opérande de gauche), autrement dit `x = 2`doit se comprendre comme $x \leftarrow 2$.

Exemples

```
test = True
n = 2
x = 4.5
ch = 'abc'
```

Dans les exemples précédents, la boîte qui a pour nom x contient la valeur 4.5, ce qui correspond à la réprésentation ci-dessous :



Remarque

Pour distinguer la variable `abc` de la chaîne de caractères `'abc'` (ou `"abc"`), les chaînes sont entourées de guillements simples ou doubles.

On affiche la valeur d'une variable dans la console avec la fonction `print` :

[1]:
```
x = 4.5
```

[2]:
```
print(x)
```

```
4.5
```

Une variable doit être définie (une valeur doit lui être affectée) pour être utilisée, sinon l'exécution provoque une erreur :

[3]:
```
print(b) # la valeur de la variable b n'a pas été définie au préalable, on ne␣
↪peut pas l'afficher
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'b' is not defined
```

Remarque : les commentaires

Dans l'exemple précédent, le texte précédé par `#` est un *commentaire*, il n'est pas interprété comme du code Python. C'est très pratique pour donner des indications utiles à celui qui lit le code, et aussi pour ne pas exécuter certaines portions du code, sans pour autant les supprimer. Si on a besoin d'écrire des commentaires sur plusieurs lignes, il suffit d'encadrer le texte à commenter entre deux balises `"""`.

Exemples

```
x = 3 # 3 est un nombre premier
```

```
"""
Tout ce texte n'est pas exécuté, y compris
l'instruction suivante :
x = 3
, pourtant valide en Python
"""
```

[4]:
```
print('abc') # on affiche la chaîne de caractère abc
```

```
abc
```

[5]:
```
print(abc) # on tente d'afficher la valeur de la variable abc, qui n'est pas␣
↪définie
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'abc' is not defined
```

Que stocke-t-on ? On imite autant que possible les ensembles mathématiques, ce qui correspond à la notion de *type* :

- les valeurs logiques : type `bool` (deux valeurs logiques `True` et `False`)
- des entiers relatifs : type `int` (integer : entier)
- des nombres décimaux (réels) : type `float` (float : représentation à virgule flottante)
- des chaînes de caractères (textes) : type `str` (string : chaîne)
- l'ensemble vide : `None`

Dans un second temps on abordera plus particulièrement le cas des séquences (dont font partie les chaînes)

Remarques

- on peut afficher le type d'une variable avec la fonction `type`
- le point `.` est le séparateur décimal, il permet de différencier les `float` des `int`, même s'il n'y a rien après la virgule.

[6]:
```
x = 3.
type(x)
```

[6]:
```
<class 'float'>
```

[7]:
```
n = 3
type(n)
```

[7]:
```
<class 'int'>
```

Plus précisément, un type correspond :

- au choix d'un *codage* interne : une association entre un objet mathématique et une suite de 0 et de 1. Par exemple, on peut choisir $4 \to 100$

- à la définition d'*opérateurs* spécifiques (en fait une fonction qu'on appelle à l'aide d'un symbole pratique à l'utilisation). Par exemple, pour additionner deux entiers `n` et `m` on écrira `n+m`.

Dans un premier temps on va s'intéresser uniquement aux opérateurs.

3.2 Opérateurs

3.2.1 Opérateurs logiques (type `bool`)

Opérateur	Fonction logique
`not`	NON
`and`	ET
`or`	OU

Ces opérateurs renvoient toujours un `bool`

[8]:
```
True and False
```

[8]:
```
False
```

[9]:
```
a = False
b = False
a and not(b)
```

[9]:
```
False
```

3.2.2 Opérateurs arithmétiques communs aux types `int` et `float`

Opérateur	Fonction arithmétique
`+`	addition
`-`	soustraction et opposé
`*`	multiplication
`**`	puissance
`/`	division

Attention, l'opérateur `/` produit *toujours* un float, même quand le résultat "tombe juste".

[10]:
```
2.2**3
```

[10]:
```
10.648000000000003
```

[11]:
```
1-3
```

[11]:
```
-2
```

[12]:
```
6/3
```

[12]: 2.0

3.2.3 Division euclidienne (type `int`)

Opérateur	Fonction arithmétique
`//`	quotient
`%`	reste (modulo)

Ces opérateurs renvoient toujours un `int`

[13]:
```
10//3
```

[13]: 3

[14]:
```
10%3
```

[14]: 1

3.2.4 Opérateurs de comparaison

Opérateur	Fonction de comparaison
`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

Ces opérateurs renvoient toujours un `bool`

[15]:
```
2 != 3
```

[15]: True

[16]:
```
2 > 3
```

[16]: False

3.3 Conversion entre types

On peut convertir un type en un autre à l'aide de la fonction qui porte le nom du type que l'on souhaite en sortie. Toutes les conversions ne sont pas possibles !

[17]:
```
float(2) # conversion de l'int 2 en float : une partie fractionnaire nulle est␣
↪ajoutée
```

[17]:
```
2.0
```

[18]:
```
int(2.5) # conversion du float 2.5 en int : la partie fractionnaire disparaît
```

[18]:
```
2
```

[19]:
```
int('2') # conversion possible d'une chaîne en int
```

[19]:
```
2
```

[20]:
```
int('a') # conversion impossible d'une chaîne en int
```

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

3.4 Expressions

- Comme en maths, on peut écrire des expressions telle que $2x+3$, mais contrairement à ce qui se passe en maths, l'expression est systématiquement *évaluée* (= remplacée par une valeur) lors de l'exécution. Sur l'exemple précédent, il faut donc que la valeur de la variable x soit préalablement définie pour éviter une erreur.
- l'évaluation est *paresseuse* (on arrête l'évaluation dès qu'on peut conclure sur le résultat)
- comme en maths il faut utiliser les parenthèses au moindre doute sur la priorité des opérateurs
- un cas particulier important : les expressions booléennes, qui sont évaluées à `True` ou `False`

[21]:
```
-2*3 + 4
```

[21]:
```
-2
```

[22]:
```
not(2 < 3) and (2 != 3)
```

[22]:
```
False
```

[23]:
```
not(2 < 3) and int('a')
```

[23]:
```
False
```

L'évaluation de l'expression précédente ne renvoie pas d'erreur, alors que la conversion `int('a')` en renvoie une (voir plus haut). On voie ici à l'oeuvre le processus d'évaluation paresseuse : comme la sous-expression `not(2<3)` est évaluée à `False`, le reste de l'expression n'est pas évaluée puisque `False and ...` ne peut que renvoyer `False`.

4 Instructions conditionnelles et boucles

Retour au sommaire

Comme toutes les structures de contrôle, elles permettent de dépasser le principe *une instruction écrite = une instruction exécutée.*

- instructions conditionnelles : constructions d'arbres de décision
- instructions répétitives ou *boucles* : automatisation de tâches répétitives donc *factorisation du code*

Dans les deux cas, l'exécution des blocs de code est conditionnée par l'évaluation à `True` ou `False` d'une expression booléenne (= la condition).

4.1 Instructions conditionnelles

En fonction des besoins :

- instruction `if` (Si...)
- instruction `if...else` (Si... Sinon...)
- instruction `if...elif...else` (Si... Sinon si...Sinon...)

Syntaxe

- les deux points terminent la condition
- les blocs de code sont définis par le *niveau d'indentation* (décalage du texte par la touche TAB)
- les instructions `else` et `elif` sont facultatives (on peut avoir un `if` seul par exemple)

```
if expression_booleenne_1:
    # ligne de code
    # ligne de code
    # ligne de code

elif expression_booleenne_2:
    # ligne de code
    # ligne de code
    # ligne de code

else:
    # ligne de code
    # ligne de code
    # ligne de code
```

Quelques exemples :

[24]:
```
x = 2
```

[25]:
```
print('ligne avant')
if x < 3: # ici la condition est vraie, le bloc de code du if est exécuté
    print('x est inférieur à 3')
print('ligne après')
```

```
ligne avant
x est inférieur à 3
ligne après
```

[26]:
```
x = 4
```

[27]:
```
print('ligne avant')
if x < 3: # ici la condition est fausse, le bloc de code du if n'est pas exécuté
    print('x est inférieur à 3')
print('ligne après')
```

```
ligne avant
ligne après
```

[28]:
```
x = 4
y = 2
```

[29]:
```
if x < y:
    print('x < y')
else: # ici on traite implicitement le cas x >= y (négation logique de x < y)
    print('x >= y')
```

```
x >= y
```

[30]:
```
if x < y:
    print('x < y')
elif x > y:
    print('x > y')
else: # ici on traite le seul cas non traité précédemment, donc le cas x = y
    print('x = y')
```

```
x > y
```

4.2 Instructions répétitives (boucles)

Principe

L'exécution d'un bloc de code est répétée (= une *itération*) tant qu'une condition est vraie (= une expression booléenne évaluée à `True`).

Deux types de boucles en fonction des besoins :

- boucle non déterministe `while` (*Tant que*) : le nombre d'itérations est *à priori* inconnu
- boucle déterministe `for..in` (*Pour*) : le nombre d'itérations est *à priori* connu

4.2.1 Boucles `while` (Tant que)

Syntaxe Même syntaxe que pour les instructions conditionnelles, on retrouve les deux points et l'indentation :

```
while expression_boolenne:
    # ligne de code
    # ligne de code
    # ligne de code
```

Exemple

Afficher les entiers de 0 à 4

[31]:
```
i = 0
while i < 5:
    print(i)
    i = i+1
```

```
0
1
2
3
4
```

Sur l'exemple précédent :

- `i` est la variable de boucle (appelée conventionnellement `i` pour itération)
- `i = 0` est *l'initialisation* de la variable de boucle
- `i < 10` est la condition de répétition (…ou d'arrêt, question de point de vue)
- Le bloc de code :

```
print(i)
i = i + 1
```

correspond aux lignes de code dont l'exécution est répétée par la boucle

- `i = i + 1` est *l'incrémentation* de la variable de boucle : la valeur de `i` est augmentée de 1

  Remarques

  Une boucle `while` peut être infinie (= ne se termine jamais) si :

  - on se trompe sur la condition d'arrêt
  - on oublie d'incrémenter `i`
  - on se trompe sur l'initialisation de `i`

  De plus, l'initialisation et l'incrémentation sont systématiques dans un tel cas, il est donc pénible d'écrire à chaque fois ces lignes.

  Tous ces inconvénients justifient l'existence de la boucle `for` (voir plus loin).

Boucles infinies

- un programme qui ne se termine pas peut avoir des conséquences catastrophiques (penser aux avions, aux centrales nulcéaires, etc)
- mais on ne peut pas se passer des boucles non déterministes, par exemple toutes les applications doivent tourner tant qu'on ne les arrête pas :

Exemple

```
Tant qu'on ne ferme pas l'application
        faire tourner l'application
```

Il faut garder en tête que derrière l'instruction `while` on a au final un booléen (évaluation de la condition).

Exemple

```
while True: # ici la boucle est infinie...
    print('pouet')
```

Cela peut être très utile en combinaison avec l'emploi de fonctions qui renvoient des booléens (voir plus loin)

4.2.2 Boucles `for..in`

- plus sûres et plus simples à écrire
- à utiliser systématiquement dès que c'est possible

Syntaxe avec l'utilisation de range, pour parcourir un intervalle d'entiers :

```
for i in range(imin, imax, pas):
    # ligne de code
    # ligne de code
    # ligne de code
```

[32]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Remarques

Garder en tête l'équivalence avec la boucle `while` pour comprendre le fonctionnement du `range` :

- `imin` est la valeur de l'initialisation (égale à 0 si elle n'est pas précisée)
- `imax` correspond à la condition d'arrêt, équivalente à `i < imax`
- le `pas` (l'incrément) est égal à 1 s'il n'est pas précisé

Exemple

[33]:
```
for i in range(4, -1, -1): # ici on compte à l'envers
    print(i)
```

```
4
3
2
1
0
```

4.2.3 Instruction `break`

Placée dans le bloc de code qui dépend d'une boucle, cette instruction permet de sortir prématurément de celle-ci. Par exemple :

[34]:
```
for i in range(10): # à priori on fait 10 itérations
    print(i)
```

```
    if i == 5:
        break # on sort prématurément de la boucle pour i = 5
```

```
0
1
2
3
4
5
```

5 Fonctions

Retour au sommaire

Notion d'une importance extrême. Les fonctions permettent :

- de définir des fonctions mathématiques, comme en maths
- de découper les problèmes complexes en sous-problèmes plus simples par *composition* de fonctions
- de factoriser le code (= éviter de faire du copier-coller)
- d'encapsuler du code (= donner un nom à un bloc de code)
- d'implémenter des algorithmes récursifs

5.1 Définition d'une fonction

Comme en maths : une fonction = association entre une valeur d'entrée et une *unique* valeur de sortie. De façon plus abstraite, tout *programme* est censé être une fonction, d'où le nom d'*application* donné aux gros programmes (synonyme de fonction en maths).

Exemple

$$f : x \mapsto x^2$$

Traduction en Python :

```
def f(x:float)->float:
    """
    Retourne le carré du nombre x.
    """
    return x**2
```

Vocabulaire

- on écrit le nom de la fonction après le mot-clé `def` (pour définition - de la fonction)
- entre parenthèses on trouve les noms des *paramètres* de la fonction (en maths on emploie le terme de *variables*)
- après le mot-clé `return` on trouve la valeur renvoyée par la fonction lorsqu'elle est *appelée* (utilisée dans un calcul)

Remarques

- on peut n'avoir aucun paramètre (= fonction constante en maths)
- mais il est très fréquent d'avoir plusieurs paramètres (= fonction de plusieurs variables en maths)
- on ne peut renvoyer qu'une seule valeur (cf la définition)
- l'absence de `return` est équivalent à la ligne

```
return None
```

Dans ce cas la fonction est qualifiée de *procédure*, elle ne renvoie rien (`None`).

- la définition de la fonction n'est exécutée *que lors de l' appel*

5.2 Appel d'une fonction

Appel d'une fonction = *évaluation* de la fonction pour un ensemble d'*arguments* (= valeurs d'entrée qui remplacent les paramètres lors de l'appel).

[35]:
```
def add(x:float, y:float)->float: # Définition de la fonction
    """
    Retourne la somme des nombres x et y.
    """
    return x + y
```

[36]:
```
print(add(2, 3)) # appel avec les arguments 2 et 3
```

```
5
```

Remarques

- un appel de fonction est caractérisé par l'emploi des parenthèses, même si la définition de la fonction ne comporte pas de paramètre
- l'instruction `return` ne peut être exécutée qu'une seule fois par appel, elle fait *sortir* de la fonction (elle stoppe son évaluation)

[37]:
```
def constante()->None:
    """
    Retourne la valeur 3.
    """
    return 3
```

[38]:
```
print(constante()) # appel de la fonction et affichage du résultat
```

```
3
```

[39]:
```
def maxi(x:float, y:float)->float:
    """
    Retourne le maximum des deux nombres x et y.
    """
    if x > y: # un seul des deux return sera exécuté puisqu'ensuite on sort de␣
 ↪la fonction
```

```
        return x
    else:
        return y
```

[40]:
```
maxi(2, 3)
```

[40]: 3

Utilisation de `return` pour sortir prématurément d'une boucle

[41]:
```
def stop5()->None:
    """
    Affiche les 6 premiers entiers.
    """
    for i in range(0, 10): # on devrait à priori faire 10 itérations
        print(i)
        if i == 5: # mais on sort prématurément pour i = 5 à cause du return
            return None
```

[42]:
```
stop5()
```

```
0
1
2
3
4
5
```

5.3 Composition de fonctions

Une fonction peut en appeler une autre, ce qui permet de décomposer des problèmes complexes en sous-problèmes plus simples.

Exemple

En utilisant la fonction `max`, on peut définir une fonction `max3`, qui renvoie le maximum de trois nombres :

[43]:
```
def maxi3(x:float, y:float, z:float)->float:
    """
    Retourne le maximum des trois nombres x, y, et z.
    """
    return maxi(maxi(x, y), z)
```

[44]:
```
maxi3(2, 4, 3)
```

[44]: 4

5.4 Portée des variables

A chaque fonction et chaque script est associé un *espace de noms* (= l'ensemble des noms de variables connus dans le contexte où on est). Celui du script est l'espace de noms *global*, tandis qu'il existe un espace de noms *local* à chaque fonction.

Si une variable est évaluée dans un script hors du code d'une fonction, si son nom n'est pas trouvé dans l'espace de nom global, elle est non définie et l'interpréteur renvoie une erreur (variable non définie).

En revanche si une variable est évaluée lors de l'exécution du code d'une fonction, l'interpréteur cherche d'abord sa définition *localement* à la fonction, et en cas d'échec cherche dans l'espace de nom gobal. Un nouvel échec conduit l'interpréteur à renvoyer une erreur.

Exemple (à bien comprendre)

```
def sub(x:int, y:int)->float:
    """
    Une fonction exemple.
    """
    z = 2
    return z*(x - y)/2
x = 2
y = 3
sub(y, x)
```

Sur l'exemple précédent :

- les paramètres de la fonction `x` et `y` sont des variables *locales*
- la variable `z` définie dans la fonction est *locale*
- les variables `x` et `y` définies hors de la fonction sont *globales*

Exemple

A quelle valeur est évaluée `sub(y, x)` ?

Les valeurs de `y` et `x` sont cherchées dans l'espace de noms global. Les valeurs sont trouvées dans cet espace ($y = 3$ et $x = 2$) et donc la fonction est appelée avec les arguments 3 et 2 (**dans cet ordre**), autrement dit tout se passe comme si on avait appelé `sub(3, 2)`. Dans l'espace de nom local de la fonction, les paramètres `x` et `y` ont donc pour valeurs 3 et 2 (**dans cet ordre**). Lors de l'évaluation de l'expression `z*(x - y)/2` retournée par la fonction, les valeurs des variables sont d'abord cherchées localement (et sont toutes trouvées puisque `x`, `y`et `z`ont des définitions locales) et donc la valeur de retour sera $2 \times (3 - 2)/2 = 1$.

6 Séquences : listes, tuples et chaînes

Retour au sommaire

- séquence = une *suite finie* : chaque *élément* d'une séquence est accessible via son *indice* (= *index* en anglais)

- permettent de stocker des *vecteurs* (au sens large), ce sont les plus simples des *structures de données* (autres exemples : arbres, graphes…)
- 3 types de séquences en Python : listes, tuples et chaînes de caractères

Séquence	Type	Délimiteurs	Modifiable	Type des éléments	Exemple
listes	`list`	`[ ]`	OUI	quelconque	`li = [2.4, True, 12]`
tuple	`tuple`	`( )` ou rien	NON	quelconque	`tp = 2.4, True, 12`
chaînes de caractères	`str`	`' 'ou " "`	NON	caractères	`ch = 'abc'`

6.1 Propriétés communes

Tous ce qui suit est valide sur **toutes les séquences** (listes, tuples et chaînes), même si les exemples choisis ne traitent que le cas des listes et des chaînes.

Taille d'une séquence

- renvoyée par la fonction `len`

[45]:
```
li = [1, 2, 3]
print(len(li))
```

```
3
```

[46]:
```
ch = 'abc'
print(len(ch))
```

```
3
```

Indexation

- on accède à un élément via l'*opérateur d'extraction* `[]`
- Les éléments sont indexés de 0 à taille-1
- les indices négatifs permettent une indexation en partant de la fin (revient à omettre `len(li)` entre les crochets)

[47]:
```
li[0]
```

[47]: 1

[48]:
```
ch[0]
```

[48]: 'a'

[49]:
```
li[len(li)-1]
```

[49]: 3

[50]:
```
li[-1]
```

[50]: 3

[51]:
```
ch[-1]
```

[51]:
```
'c'
```

[52]:
```
li[-2]
```

[52]:
```
2
```

Slicing

- cas général de l'extraction (découpage de *tranches*)
- syntaxe identique à celle de `range` : imin (inclus), imax (exclu), pas

[53]:
```
li = [0, 1, 2, 3, 4, 5]
ch = 'abcdefgh'
```

[54]:
```
li[2:5]
```

[54]:
```
[2, 3, 4]
```

[55]:
```
ch[1:4]
```

[55]:
```
'bcd'
```

[56]:
```
li[:-1]
```

[56]:
```
[0, 1, 2, 3, 4]
```

[57]:
```
ch[2:]
```

[57]:
```
'cdefgh'
```

[58]:
```
li[::2]
```

[58]:
```
[0, 2, 4]
```

Parcours

- à savoir faire impérativement, permet l'automatisation
- utilisation d'une boucle `for..in` :
  - pour parcourir les indices des éléments (avec un `range`)
  - pour parcourir les éléments eux-mêmes

[59]:
```
li = [1, 2, 3]
ch = 'abc'
```

[60]:
```
for i in range(len(li)): # ici on parcourt l'ensemble des indices
    print(li[i])
```

```
1
2
3
```

[61]:
```
for i in range(len(ch)):
    print(ch[i])
```

```
a
b
c
```

[62]:
```
for el in li: # ici on parcours l'ensemble des élements
    print(el)
```

```
1
2
3
```

[63]:
```
for car in ch:
    print(car)
```

```
a
b
c
```

Concaténation

Avec l'opérateur `+` :

[64]:
```
[1, 2] + [3, 4]
```

[64]:
```
[1, 2, 3, 4]
```

[65]:
```
'ab' + 'cd'
```

[65]:
```
'abcd'
```

Avec l'opérateur `*` (multiplication au sens de la concaténation), pour créer des séquences d'éléments identiques :

[66]:
```
li = [1] * 3
print(li)
```

```
[1, 1, 1]
```

[67]:
```
ch = 'a' * 3
print(ch)
```

```
aaa
```

Pourquoi utiliser des tuples (plutôt que des listes) ?

À première vue, les tuples sont des listes en moins bien (cf tableau comparatif des séquences). MAIS :

- non modifiables, ils permettent de protéger les données d'une modification accidentelle : on est assuré de leur contenu qui ne peut pas changer par définition

[68]:
```
tp = (1, 2, 3)
tp[1] = 0 # on essaye d'écraser la deuxième valeur du tuple
```

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
```

[69]:
```
tp.append(4) # on essaye d'ajouter un élément au tuple comme on le ferait avec␣
↪une liste
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

- leurs délimiteurs `()` sont optionnels (sauf dans le cas du tuple vide) : simplicité d'écriture pour affecter plusieurs variables d'un coup

[70]:
```
a, b = 2, 3 # équivalent aux deux affectations a = 2 et b = 3
```

[71]:
```
print(a, b)
```

```
2 3
```

- Ils sont très souvent employés comme valeur de retour des fonctions pour renvoyer indirectement plusieurs valeurs, et affecter en une ligne le résultat à plusieurs variables :

[72]:
```
def min_max(x:int, y:int)->(int, int):
    """
    Retourne le couple (min, max) qui correspond à (x, y).
    """
    if x < y:
        return x, y
    else:
        return y, x
```

[73]:
```
m, M = min_max(5, 3)
print(m, M)
```

```
3 5
```

6.2 Cas particulier des listes

Méthodes des listes

- 11 *méthodes* (= fonctions spécifiques des listes) intégrées au langage pour les manipuler :
  - ajout d'un élément
  - suppression d'un élément
  - tri
  - ...
- taper `help(list)` dans une console Python pour afficher la liste de toutes les méthodes et leur documentation
- seulement deux méthodes à connaître absolument : `append` (ajout d'un élément à la fin de la liste) et `pop` (supression et retour du dernier élément)

[74]:
```
li = []
```

[75]:
```
li.append(2) # attention à la syntaxe de l'appel des méthodes
```

[76]:
```
li
```

[76]:
```
[2]
```

[77]:
```
li.append(3)
```

[78]:
```
li
```

[78]:
```
[2, 3]
```

[79]:
```
li = []
for i in range(10):
    li.append(i)
```

[80]:
```
li
```

[80]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[81]:
```
der = li.pop()
print(der)
```

```
9
```

[82]:
```
print(li)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Listes en compréhension

En mathématiques, on définit fréquemment de nouveaux ensembles "en compréhension", à partir d'un ensemble déjà défini et d'une propriété. Par exemple pour les entiers pairs :

$$\{n \in \mathbb{N} | n \equiv 0 \pmod 2\}$$

Par analogie, on peut définir des listes en compréhension, par exemple pour la liste des entiers pairs compris entre 1 et 10 :

[83]:
```
li = [n for n in range(0, 10) if n % 2 == 0] # n % 2 == 0 : n est divisible par␣
↪2
print(li)
```

```
[0, 2, 4, 6, 8]
```

6.3 Séquences et adresses mémoire

- On a vu : une variable = association entre une valeur et une chaîne de caractère (son *nom*)
- Mais en fait : une variable = association entre une valeur et une adresse mémoire (sa *référence*).

Le nom n'est nécessaire que pour les humains, la machine n'a besoin que des adresses. On peut avoir accès à l'adresse d'une variable avec la fonction `id` :

[84]:
```
x = 2
id(x)
```

[84]:
```
2810116
```

Une valeur fait une certaine taille en mémoire (ex : un `float` = 64 bits). Une valeur occupe au plus un *mot mémoire* (64 bits pour les systèmes 64 bits).

Mais comment faire quand on a plusieurs valeurs (cas des séquences) ? Et qu'en plus le nombre de valeurs peut varier (cas des listes) ?

Solution : la valeur qu'on stocke est l'*adresse mémoire* de la première valeur. Cela a beaucoup de conséquences...

Copie de listes

- avec deux variables ordinaires, pas de problème

[85]:
```
x = 2
y = x
print(x, y)
x = 3
print(x, y)
```

```
2 2
3 2
```

- avec des listes, il faut se souvenir qu'on ne copie que l'adresse, pas les valeurs ! Donc écrire `li = li2` créé juste un deuxième nom pour le même ensemble d'éléments :

[86]:
```
li = [1, 2, 3]
li2 = li
print(li, li2)
li[1] = 0 # on modifie un élément de li
```

```
print(li, li2) # on s'aperçoit que li2 est aussi modifiée, puisque ce sont les␣
↪mêmes
```

```
[1, 2, 3] [1, 2, 3]
[1, 0, 3] [1, 0, 3]
```

Solution : utiliser la méthode `copy` des listes pour copier les valeurs.

[87]:
```
li = [1, 2, 3]
li2 = li.copy()
print(li, li2)
li[1] = 0
print(li, li2) # cette fois seule li a été modifiée
```

```
[1, 2, 3] [1, 2, 3]
[1, 0, 3] [1, 2, 3]
```

Listes comme arguments de fonctions

A cause du passage des adresses en argument, une liste définie à l'extérieur d'une fonction est quand même modifiable par la fonction (connaître l'adresse d'une variable permet de l'atteindre de n'importe quel endroit du code). Les listes sont donc techniquement des variables globales, qu'on le veuille ou non.

[88]:
```
def f(x):
    x = x + 1 # pas de return, seule la variable locale x est modifiée
```

[89]:
```
x = 2
f(x)
print(x) # la valeur de la variable x n'a pas changée
```

```
2
```

[90]:
```
def f(li):
    li.append(4) # pas de return, mais li est une liste, donc une adresse
```

[91]:
```
li = [1, 2, 3]
f(li)
print(li) # la "valeur" de la variable li a changé
```

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

7 Dictionnaires

Retour au sommaire

- un dictionnaire est un ensemble de paires *clé : valeur* (on accède à un élément par sa *clé*)
- le type correspondant est `dict`
- en pratique les clés sont des nombres, des chaînes ou des tuples, les valeurs associées sont quelconques

Définition

On peut définir un dictionnaire en utilisant des accolades comme délimiteurs `{ }` :

[92]:
```
dic = {'red':0, 'green':1, 'blue':2}
```

On accède à un élément par sa clé :

[93]:
```
dic['blue']
```

[93]:
```
2
```

On peut ajouter une paire en affectant une valeur à une clé qui n'existe pas déjà :

[94]:
```
dic['yellow'] = 3
```

[95]:
```
dic
```

[95]:
```
{'red': 0, 'green': 1, 'blue': 2, 'yellow': 3}
```

Parcours

- Parcours des clés
- Parcours des paires (clé, valeur)

[96]:
```
for k in dic: # parcours des clés
    print(k)
```

```
red
green
blue
yellow
```

[97]:
```
for k, v in dic.items(): # parcours des paires clé, valeur
    print(k, v)
```

```
red 0
green 1
blue 2
yellow 3
```

> **Remarques** - l'ensemble des clés est aussi accessibles par `dic.keys()` - l'ensemble des valeurs est directement accessibles par `dic.values()` - on peut aussi définir les dictionnaires en compréhension :

[98]:
```
dic = {i : i**2 for i in range(4)}
```

[99]:
```
print(dic)
```

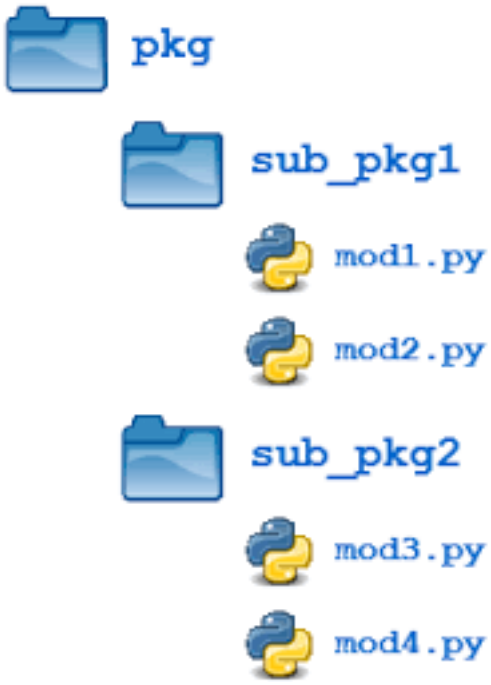
```
{0: 0, 1: 1, 2: 4, 3: 9}
```

8 Modules et packages

Retour au sommaire

8.1 Organisation d'un projet

- un projet = un gros programme = une *arborescence* de dossiers et de fichiers
- un *package* = un dossier
- un *module* = un script `.py` (fichier) = un ensemble de définitions de fonctions



- tout script Python (fichier avec extension `.py`) est un module
- on peut *importer* tout ou partie des fonctions d'un module pour qu'elles soient utilisables dans un autre
- permet l'utilisation d'une fonction qu'on a pas écrit soi-même
- permet de répartir le code d'un projet sur plusieurs fichiers de taille raisonnable (lisible par un humain)

Exemple

Imaginons l'organisation d'une appplication de traitement d'images (Photoshop par exemple) en 3 sous-parties :

- calcul matriciel (une image = une matrice) : fonctions pour réaliser les opérations (rotation, niveaux de gris, floutage, contours, etc)
- gestion des entrées et des sorties : fonctions pour lire et écrire les fichiers images
- interface graphique : apparence du logiciel à l'écran, ergonomie, affichage des images

Découpage logique du code à écrire :

- regroupage thématique des fonctions par modules
- regroupage thématique des modules en packages

8.2 Imports

Import d'un module

- c'est le *nom* du module qui est en fait importé dans l'espace de noms global
- on peut voir l'ensemble des noms connus avec l'instruction `dir()`

[100]:
```
import math # import d'un module
```

[101]:
```
print(dir()) # seul le nom math est importé
```

```
['In', 'M', 'Out', '_', '__', '___', '__builtins__', '__doc__', '__eval_data__',
'__name__', 'a', 'add', 'b', 'car', 'ch', 'constante', 'der', 'dic', 'el', 'f',
'i', 'k', 'li', 'li2', 'm', 'math', 'maxi', 'maxi3', 'min_max', 'n', 'stop5',
'tp', 'v', 'x', 'y']
```

Après cet import, pour appeler une fonction du module, il faut préfixer le nom de la fonction par le nom du module par exemple :

```
math.sin(2)
```

pour calculer le sinus de 2 (radians).

Import de quelques fonctions d'un module

[102]:
```
from math import cos, sin # on sépare les noms des fonctions par des virgules
```

Après cet import, on peut utiliser les fonctions cos et sin comme si on les avait écrites dans le script, c'est-à-dire sans préfixe :

```
sin(2)
```

Ce qui est moins lourd à la lecture / écriture du code.

[103]:
```
print(dir())  # les noms des fonctions cos et sin sont importées
```

```
['In', 'M', 'Out', '_', '__', '___', '__builtins__', '__doc__', '__eval_data__',
'__name__', 'a', 'add', 'b', 'car', 'ch', 'constante', 'cos', 'der', 'dic',
'el', 'f', 'i', 'k', 'li', 'li2', 'm', 'math', 'maxi', 'maxi3', 'min_max', 'n',
'sin', 'stop5', 'tp', 'v', 'x', 'y']
```

Import de toutes les fonctions d'un module

[104]:
```
from math import *
```

Après cet import, tous les noms des fonctions du module sont importés. C'est à déconseiller, car ces noms pourraient rentrer en conflit avec les noms des fonctions ou variables qui sont définies dans notre script.

[105]:
```
print(dir()) # à déconseiller, beaucoup de noms sont importés et on ne les␣
↪connaît pas à priori
```

```
['In', 'M', 'Out', '_', '__', '___', '__builtins__', '__doc__', '__eval_data__',
'__name__', 'a', 'acos', 'acosh', 'add', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'b', 'car', 'cbrt', 'ceil', 'ch', 'comb', 'constante', 'copysign',
'cos', 'cosh', 'degrees', 'der', 'dic', 'dist', 'e', 'el', 'erf', 'erfc', 'exp',
'exp2', 'expm1', 'f', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'i', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan',
'isqrt', 'k', 'lcm', 'ldexp', 'lgamma', 'li', 'li2', 'log', 'log10', 'log1p',
'log2', 'm', 'math', 'maxi', 'maxi3', 'min_max', 'modf', 'n', 'nan',
'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'stop5', 'tan', 'tanh', 'tau', 'tp', 'trunc', 'ulp', 'v', 'x', 'y']
```

Import d'un module d'un package et définition d'un alias

[106]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[107]:
```
fig, ax = plt.subplots() # l'alias plt remplace la chaîne matplotlib.pyplot
X = np.linspace(-2*np.pi, 2*np.pi, 100) # l'alias np remplace la chaîne numpy
ax.plot(np.sin(X))
plt.show()
```

8.3 Organisation d'un script

Il est très important de s'astreindre à respecter certaines règles d'organisation du code pour à terme gagner du temps à la relecture. Conventionnellement, on choisira l'ordre suivant :

1. Les imports
2. Les définitions de fonctions
3. le reste

```
# Imports

import matplotlib.pyplot as plt
import numpy as np
from math import cos, sin, pi

# Fonctions

def sinc(x):
    if x == 0
        return 1
    else:
        return sin(x)/x

# Le reste

print(sinc(0))
print(sinc(2))
```

9 Fichiers

Retour au sommaire

9.1 Fichiers et formats

- mémoires de masse = des mémoires informatiques qui stockent l'information même après coupure de l'alimentation électrique.
- un fichier = une portion d'une mémoire de masse (mémoire flash, disque dur, DVD…) à laquelle on associe un nom (par ex. `toto.pdf`, `titi.txt`, etc).
- format d'un fichier = la façon dont sont codées les données sous forme binaire.
- généralement le suffixe du nom de fichier (= son extension) correspond au format

Formats ouverts et propriétaires

- formats *ouverts* : tout le monde peut lire ou écrire, le format est *public*, par ex. `.odf` utilisé par le logiciel libre OpenOffice de Sun
- formats *propriétaires* : seul l'auteur du format peut lire ou écrire, par ex. `.doc` utilisé par le logiciel Word de Microsoft

Fichiers texte

- le plus simple des formats ouverts : on peut interpréter la suite de 0 et de 1 stocké sur le disque comme une chaîne de caractères lisible par un humain
- souvent on les distingue par l'extension `.txt`

9.2 Ouverture et fermeture d'un fichier

- le FS (File System) est un des services majeurs fournis par les OS (Operating System)
- un FS est une arborescence de fichiers et dossiers
- un FS ne gère pas la concurrence d'accès : si une ressource (programme) accède à un fichier (*ouverture*), il empêche les autres programmes d'y avoir accès tant qu'il n'est pas *fermé*.

Il faut donc toujours penser à fermer un fichier après ouverture !

9.3 Avec Python

Ouverture avec la fonction `open`. Syntaxe :

```
f = open(nom_fichier, mode_ouverture)
```

- mode d'ouverture : `'r'` pour read, `'w'` pour write, `'a'` pour append…
- le fichier doit être dans le même dossier que le script, sinon il faut préciser le chemin.
- l'objet `f` renvoyé par `open` est de type `file`, il possède des méthodes (fonctions) qui permettent essentiellement de lire, d'écrire et de fermer le fichier.

Exemple : ouverture / création d'un fichier puis écriture

[108]:
```
f = open('data.txt', 'w')
ch = 'blablabla' # la chaîne qu'on va stocker dans le fichier
f.write(ch)
```

```
f.close()
```

Exemple : ouverture puis lecture

[109]:
```
f = open('data.txt', 'r')
ch = f.read()
f.close()
print(ch)
```

```
blablabla
```

Fichiers `.csv`

- pour les données scientifiques tabulées, on utilise très souvent le format `csv` (Comma Separated Values)
- on choisit une *convention* de lecture et d'écriture : un caractère séparateur de colonnes (souvent `;` ou `,` d'où le nom), les lignes sont séparées par le caractère de retour à la ligne (noté `\n`).

Supposons qu'on veuille stocker dans le fichier `data.csv` le tableau suivant :

x	y
0	0
1	1
3	9
4	16

Il suffit d'écrire :

[110]:
```
ch = 'x,y\n0,0\n1,1\n3,9\n4,16\n'

f = open('data.csv', 'w')
f.write(ch)
f.close()
```

On peut s'en assurer en ouvrant le fichier pour le lire :

[111]:
```
f = open('data.csv', 'r')
ch = f.read()
f.close()
print(ch)
```

```
x,y
0,0
1,1
3,9
4,16
```

Tous les tableurs (LibreOffice Calc, Excel, etc) permettent d'enregistrer les données sous le format csv, ce qui permet ensuite de traiter les données avec un programme Python.

Exemple de programme qui stocke les données tabulées dans deux listes de nombres

[112]:
```
f = open('data.csv', 'r')
ch = f.readlines() # on récupère le contenu du fichier sous forme d'une liste␣
 ↪de lignes (chaînes)
print(ch)
li_x, li_y = [], []
for line in ch[1:]: # on "saute" le premier élément qui correspond à l'en-tête␣
 ↪du fichier
    sx, sy = line.split(',') # on découpe les lignes suivant le caractère␣
 ↪séparateur (la virgule)
    li_x.append(float(sx)) # on ajoute la première chaîne convertie en float à␣
 ↪la liste des abscisses
    li_y.append(float(sy[:-1])) # idem pour les ordonnées, en se débarassant du␣
 ↪caractère '\n' de retour à la ligne
f.close()
```

```
['x,y\n', '0,0\n', '1,1\n', '3,9\n', '4,16\n']
```

[113]:
```
li_x, li_y
```

[113]:
```
([0.0, 1.0, 3.0, 4.0], [0.0, 1.0, 9.0, 16.0])
```