Introduction au langage Python

Notes du cours sur diaporama.

I LANGAGES DE PROGRAMMATION

1 Vous avez dit langages de programmation ?

Il y en a beaucoup !!!

– D'après

http://fr.wikipedia.org/wiki/Liste_des_langages_de_programmation,

il y en aurait au moins 661 !

– Vous avez compté ?????

– Non !

```
fichier = open('liste_langages.txt', 'r')

compteur = 0

for ligne in fichier:
    if len(ligne) > 1 and ligne[1:-1] != '[modifier | modifier le code]'):
        compteur += 1

print(compteur)

fichier.close()

# La même chose en une ligne...

with open('liste_langages.txt', 'r') as fichier:
    print(len([0 for ligne in fichier  if len(ligne) > 1 and \
          ligne[1:-1] != '[modifier | modifier le code]')]))
```

2 Du bas niveau au haut niveau...

Les langages de plus bas niveau sont ceux qui permettent de contrôler tout ce qui se passe dans la machine pendant l'exécution du code (assembleur, C...)

Dans les langages de plus haut niveau, tout est transparent pour faciliter la programmation (java, python, caml...)

Mais on ne contrôle pas tout !

II INTERPRÉTEUR VS. COMPILATEUR

Deux types de langages :

- **Interprété** :

$$\text{Code source} \underbrace{\dashrightarrow}_{\text{Interpréteur}} \text{Résultat}$$

  Avantages : test immédiat

  Inconvénients : plus lent

- **Compilé** :

$$\text{Code source} \underbrace{\dashrightarrow}_{\text{Compilateur}} \text{Code machine} \underbrace{\dashrightarrow}_{\text{Exécuteur}} \text{Résultat}$$

  Avantages : beaucoup plus rapide à l'exécution

  Inconvénients : étape de compilation supplémentaire

Nous utiliserons python avec un interpréteur.

III LA CONSOLE ET L'IDE IDLE

1 Interprétation directe

Nous utiliserons dans le cadre de ce cours python version 3.x[1], sa console, et son environnement de développement (IDE) appelé Idle.

- La console permet une utilisation interactive, comme une grosse calculatrice. On y accède à partir d'un terminal grâce à l'instruction `python3`, ou en passant par l'interface graphique. C'est un interpréteur...

  PRATIQUE : la commande `help(instruction)`...

- L'IDE Idle permet de taper tout un programme (une suite d'instructions, donc) qui sera sauvegardé dans un fichier (avec, comme extension, `.py`) puis de le faire interpréter par la console (en pressant F5).

1. avec $x = 2$ ou 3 ou 4...

2 Exécuter un script Python

Remarque : On peut exécuter un fichier `.py` directement depuis la console

- en tapant `python3 nomFichier.py`
- ou alors en tapant directement le nom du fichier si on a pris le soin de rajouter au début de celui `#! /usr/bin/python3` et si on a le droit en exécution (`x`) sur le fichier.

`#!` = sha-bang

IV LES TYPES DE BASE EN PYTHON

1 Les entiers

Ils se notent comme d'habitude (en base 10 !) et peuvent avoir une longueur quelconque. (Ce n'est pas le cas dans tous les langages de programmation ! !)

```
>>> 42
42

>>> 12345678912345678913245678912345678 9
12345678912345678913245678912345678 9

>>> type(13)
<class 'int'>
```

2 Opérations sur les entiers

- Opérations de base : `+` `-` `*`

```
>>> 1 + 1
2
>>> 1 - 2
-1
>>> 3 * 4
12
```

- Division entière (quotient) : `//`

```
>>> 25 // 7
3
```

- Reste (modulo) : `%`

```
>>> 25 % 7
4
```

- Exposant : `**`

```
>>> 2**30
1073741824
```

3 Python2 vs Python3

Attention !

- En Python 2 : l'opérateur / sur des entiers donne la division entière (comme //).

```
Python 2.7.5+ (default, Jun  2 2013, 13:26:34)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 25/7
3
```

- En Python 3 : l'opérateur / sur des entiers donne la division réelle (ou plutôt flottante...)

```
Python 3.2.4 (default, May  8 2013, 20:55:18)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 25/7
3.5714285714285716
```

4 Les flottants

Certains nombres à virgules peuvent être représentés en machine[1] : on parle de nombres à virgule flottante, ou de flottants.

```
>>> 1234.1234549
1234.1234549

>>> type(3.141596)
<class 'float'>
```

Les opérateurs + - * / ** fonctionnent sur les flottants. A noter : si au moins un terme est flottant (donc s'il y a un . quelque part), le résultat sera flottant.

5 Curiosités

Comportement étrange :

```
>>> 0.6 - 0.6
0.0

>>> 0.3 + 0.3 - 0.6
0.0

>>> 0.1 + 0.2 + 0.3 - 0.6
1.1102230246251565e-16

>>> 0.1 + 0.2
0.30000000000000004

>>> 1.24563e308
1.24563e+308

>>> 1.24563e309
inf
```

sera expliqué avec le codage des flottants...

On retiendra qu'il ne faut JAMAIS tester d'égalité entre deux flottants...

1. On le détaillera dans un cours ultérieur

6 Fonctions mathématiques : le module math

```
>>> sin(3.141592)
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    sin(3.141592)
NameError: name 'sin' is not defined
```

```
>>> import math # On importe le module math

>>> sin(3.141592)
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    sin(3.141592)
NameError: name 'sin' is not defined

>>> math.sin(3.141592)
6.535897930762419e-07
```

```
>>> import math as m # on lui donne le doux nom "m"

>>> m.sin(3.141592)
6.535897930762419e-07
```

```
>>> from math import sin # On n'importe que la fonction sinus

>>> sin(3.141592)
6.535897930762419e-07
```

```
>>> from math import * # On importe tout !

>>> sqrt(2)
1.4142135623730951

>>> sin(pi)
1.2246467991473532e-16
```

Quoi d'autre dans le module math ?

→ `help(math)`...

7 Conversion (casting)

Les instructions `int()` et `float()` permettent de convertir en entier et en flottant respectivement. La conversion dans n'importe quel type d'objet fonctionne de la même manière.

8 Les booléens

Deux constantes : `True` et `False`.

Trois opérations principales : `not`, `and`, `or`.

Pour créer des assertions : `==` `!=` `<` `>` `<=` `>=`

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> 0.1 + 0.2 == 0.3 # hé hé !
False
>>> 1 < 2
True
>>> 2 > 1
True
>>> 1 <= 2
True
>>> 2 >= 1
True
>>> 0 < 1 < 2
True
>>> 4 <= 4 < 7 == 12 - 5 < 22
True
```

9 Entiers et booléens

Remarque : les entiers dans des expressions logiques sont automatiquement convertis.
`0` devient `False` et tout autre entier devient `True`.

V PYTHON ET LES AFFECTATIONS

1 L'affectation

Il s'agit de définir un emplacement dans la mémoire, de lui donner une étiquette (le nom de la variable), et d'y stocker quelque chose.

Elle se fait avec l'opérateur = :

```
nomDeVariable = valeur
```

Un nom de variable doit toujours :

- Commencer par une lettre,
- ne pas contenir de caractère spécial, d'accent, etc. Seul le souligné _ est autorisé.
- Attention : Python respect la casse, il fait la différence entre minuscule et majuscule.

```
>>> x = 42
>>> x
42
>>> id(x)    # Adresse mémoire sur laquelle pointe x
9183328
>>> type(x) # typage dynamique : pas à déclarer.
<class 'int'>
```

```
>>> x, y = 12, 42 # Affectations parallèles
>>> x
12
>>> y
42
```

```
>>> x = y = 1.5 # Affectations simultanées...
>>> x
1.5
>>> y
1.5
>>> id(x)
28436864
>>> id(y)    # ...ils pointent sur la même adresse
28436864
>>> x = 2      # Si on change l'un....
>>> y        # ...l'autre ne change pas...
1.5
>>> id(x)
9182048
>>> id(y)    # ...ils ne pointent plus sur la même adresse.
28436864
```

Exercice

Comment échanger le contenu de deux variables `x` et `y` ?

2 Raccourci opération puis affectation

Pour effectuer l'opération `x = x + 1`, il suffit de taper `x += 1`. Cela peut s'avérer très pratique !! On a ainsi des opérateurs `+=`, `-=`, `*=`, `/=`, `//=`, `\%=`, `**=`.

VI QUELQUES TYPES COMPOSÉS

1 Des non mutables

a Les chaînes de caractères

Chaîne de caractères = string en anglais = suite finie de caractères. (Du texte, quoi.)

Délimitées par des apostrophes (`'`), des guillemets (`"`) ou des triples guillemets (`"""`) si on veut aller à la ligne à l'intérieur.

```
>>> 'Une première chaîne contenant des guillemets : """'
'Une première chaîne contenant des guillemets : """'
>>> "Une deuxième chaîne, contenant des apostrophe : '''"
"Une deuxième chaîne, contenant des apostrophe : '''"
>>> "Une troisième chaîne contenant des guillemets quand même ! \"\"\""
'Une troisième chaîne contenant des guillemets quand même ! """'
>>> """Une dernière
| Mais
| sur
| plusieurs
| lignes avec des caractères spéciaux : ' """
'Une dernière\nMais\nsur\nplusieurs\nlignes avec des caractères spéciaux : ' '
```

Caractère spéciaux : `\n` = saut de ligne, `\t` = tabulation, `\'`, `\"`.
Chaîne vide : `""`.

On peut accéder au i^e caractère d'une chaîne `s` par `s[i]`. Attention : la numérotation commence à 0 et elle accepte des indices négatifs : -1 = le dernier, -2 = l'avant dernier, etc.

```
>>> s = "Hello World"
>>> s[2], s[-1], s[-3], s[10]
('l', 'd', 'r', 'd')
```

Mais pas le modifier (une chaîne n'est pas mutable).

```
>>> s[2] = 't'
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    s[2]='t'
TypeError: 'str' object does not support item assignment
```

On peut faire du *slicing* : `s[i:j]` renvoie la sous-chaîne depuis l'indice `i` jusqu'à l'indice `j-1`. `s[i:j:k]` donne la même chose mais avec un pas de `k`.

```
>>> s[2:8]
'llo Wo'
>>> s[0:11:2]
'HloWrd'
>>> s[0:11:3]
'HlWl'
>>> s[2:]
'llo World'
>>> s[:3]
'Hel'
```

On peut concaténer des chaînes (les mettre bout-à-bout) grâce à l'opérateur + et calculer la longueur d'une chaîne grâce à l'opérateur `len` :

```
>>> s = "J'aime "
>>> t = "Python"
>>> s + t
"J'aime Python"
>>> len(s)
7
```

On peut convertir des objets en chaîne de caractère grâce à l'instruction `str` , la méthode `format` permet de former facilement des chaînes de caractères à partir d'autres données.

```
>>> "Ceci est un entier " + str(231456) + " ; ceci est un flottant "
+ str(3.141592)
'Ceci est un entier 231456 ; ceci est un flottant 3.141592'
>>> "Ceci est un entier {}  ; ceci est un flottant {}".format(231456,3.141592)
'Ceci est un entier 231456 ; ceci est un flottant 3.141592'
```

L'opérateur `in` permet de faire un test d'inclusion :

```
>>> 'cou' in "coucou !"
True
>>> 'cu' in "coucou !"
False
```

b Les tuples

Tuple est le mot anglais pour *n*-uplet : ce sont des éléments (pas forcément de même type) séparés par des virgules, entourés de parenthèses. Tuple vide : `()`. Tuple à un élément : `(elem,)`.

Les propriétés et opérations sont similaires à celles sur les chaînes.

```
>>> 3, 5, 26
(3, 5, 26)
>>> t = (1, 2.65, 1<=3, "azerty")
>>> t[1]
2.6
>>> t[2] = 12
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    t[2]=12
TypeError: 'tuple' object does not support item assignment
>>> len(t)
4
>>> t + (1, 2, 3)
(1, 2.65, True, 'azerty', 1, 2, 3)
>>> 1 in t
True
>>> (1, 2.65) in t
False
```

Attention : `in` permet de savoir si on a un élément, et non un « sous-tuple ».

Pratique !

```
>>> a, b, c = 12, 36, 25
>>> a
12
>>> b
36
>>> c
25
>>> a, b = b, a
>>> a
36
>>> b
12
```

Pratique !

```
>>> t
(1, 2.65, True, 'azerty')
>>> for i in t: print(i)
1
2.65
True
azerty
```

Pratique !

```
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple(range(1, 12))
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
>>> tuple(range(1, 12, 2))
(1, 3, 5, 7, 9, 11)
```

Pratique !

```
>>> t
(1, 2.65, True, 'azerty')
>>> 3 * t
(1, 2.65, True, 'azerty', 1, 2.65, True, \
 'azerty', 1, 2.65, True, 'azerty')
```

2 Des mutables

a Les listes

Les listes se présentent comme les tuples, sauf qu'elles sont délimitées par des crochets.
Une différence fondamentale est qu'elles sont mutables : on peut changer leurs éléments.

```
>>> L = [1,2,3]
>>> L[2]
3
>>> L[1] = 0 # Mutable
>>> L
[1, 0, 3]
>>> L[1:3] # Slicing
[0, 3]
>>> len(L)
3
>>> L + [1,2,3] # Concaténation
[1, 0, 3, 1, 2, 3]
>>> 3 in L # Appartenance
True
```

```
>>> tuple(L) # Casting
(1, 0, 3)
>>> list(t)
[1, 2.65, True, 'azerty']
>>> str(L)
'[1, 0, 3]'
>>> list("Bonjour")
['B', 'o', 'n', 'j', 'o', 'u', 'r']
>>> L.append(5) # Ajouter un élément à la fin
>>> L
[1, 0, 3, 5]
>>> a = L.pop() # Récupérer le dernier élément
>>> a
5
>>> L
[1, 0, 3]
>>> [1, 2, 3] * 3 # Pratique !
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Attention à la copie de liste ! Deux variables peuvent référencer une même liste...

```
>>> L2 = L
>>> id(L), id(L2)
(140352491149648, 140352491149648)
>>> id(L) == id(L2)
True
>>> L[1] = 12
>>> L
[1, 12, 0]
>>> L2
[1, 12, 0]
>>> L3 = L.copy() # ou bien L[:] ou \
 encore list(L)
>>> id(L) == id(L3)
False
>>> L[1] = 36
>>> L
[1, 36, 0]
```

```
>>> L3
[1, 12, 0]
>>> LL = [[1],[2],[3]]
>>> LL2 = LL.copy() # copie superficielle
>>> LL2
[[1], [2], [3]]
>>> LL2[0][0] = 0
>>> LL2
[[0], [2], [3]]
>>> LL
[[0], [2], [3]]
>>> LL2[0] = [1]
>>> LL
[[0], [2], [3]]
>>> LL2
[[1], [2], [3]]
```

Pour faire une copie à tous les niveaux (liste de listes de ...), on peut utiliser la méthode `deepcopy()` du module `copy`.

b Les ensembles

On dispose en python d'un type ensemble : `set`. Comme en maths, l'ordre des éléments n'importe pas, et il n'apparaissent qu'une seule fois. C'est un type mutable.

- `{}` ou `set()` est l'ensemble vide.
- `s={1,2,3}` affecte l'ensemble {1,2,3}.
- Cardinal : `len(s)`
- Appartenance : `x in s`, `x not in s`.
- Inclusion : `s1.issubset(s2)` ou `s1 <= s2`
- Union : `s1.union(s2)` ou `s1 | s2`
- Intersection : `s1.intersection(s2)` ou `s1 & s2`
- Différence : `s1.difference(s2)` ou `s1 - s2`
- `s.add(x)`, `s.remove(x)`...

Pour la copie, les mêmes règles que pour les listes s'appliquent.

c Les dictionnaires

C'est une structure de données, appelée `dict`, qui permet d'associer un objet à une étiquette (appelée clé) autre que l'indice d'une case comme pour les listes. C'est un type mutable. Pour la copie, les mêmes règles que pour les listes s'appliquent.

```
>>> capitale = {}
>>> capitale['France'] = 'Paris'
>>> capitale['Japon'] = 'Tokyo'
>>> capitale['Allemagne'] = 'Berlin'
>>> capitale
{'France': 'Paris', 'Allemagne': 'Berlin',
'Japon': 'Tokyo'}

>>> coord = {(0, 0):'origine', (0, 1):'point A',
(1, 0):'point B'}
>>> coord[0, 1]
'point A'
```

Fonctions et méthodes usuelles sur les dictionnaires :

- `dict()` ou `{}` construit un dictionnaire vide.
- `len(dico)` donne le nombre d'éléments de `d`.
- `cle in dico` : test d'appartenance d'une clé.
- `dico.values()`, `dico.keys()`, `dico.items()` renvoient respectivement les valeurs, les clés, et les couples (clé, valeur) du dictionnaire `dico`. Attention : ce sont des structures de données particulières, à transformer en liste ou tuple si nécessaire (par exemple : `liste_cles=list(dico.keys())`).

```
>>> for key in coord.keys(): print(key,coord[key])
(0, 1) point A
(1, 0) point B
(0, 0) origine

>>> for val in coord.values(): print(val)
point A
point B
origine
```

```
>>>  for key, val in coord.items():
        print('{} se trouve en coordonnées {} !'.format(val, key))
point A se trouve en coordonnées (0, 1) !
point B se trouve en coordonnées (1, 0) !
origine se trouve en coordonnées (0, 0) !
```

VII ENTRÉES ET SORTIES EN PYTHON

Pour faire afficher du texte dans la console, on utilise `print`.

```
>>> print('Bonjour')
Bonjour
>>> print('1 + 1 = ',2)
1 + 1 = 2
>>> print(1, 2, 3, sep=' - ')
1 - 2 - 3
```

Pour obtenir une entrée au clavier on utilise `input`, cela renvoie une chaîne de caractères.

```
>>> reponse = input('Entrée ? ')
Entrée ? 12
>>> print(reponse)
12
>>> type(reponse)
<class 'str'>
>>> reponse+2
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module>
    reponse+2
TypeError: Can't convert 'int' object to str implicitly
>>> int(reponse) + 2
14
```

Nous utiliserons très peu ces instructions, en tirant avantage de la notion de fonction. Elle ne sont utiles que pour faire de l'interfaçage avec l'utilisateur ce qui est secondaire pour nous.

VIII BOUCLES ET TESTS

1 Les tests

Pour effectuer des tests du type « Si … Alors … FinSi » ou « Si … Alors … Sinon … FinSi » ou encore « Si … Alors … Sinon-Si … Alors … FinSi » la structure Python est la suivante :

```
if <condition_1>:
    <instructions_1>
elif <condition_2>:
    <instructions_2>
elif <condition_3>:
    <instructions_3>
...
else:
    <instruction>
```

Les `elif` et le `else` sont facultatifs.

Attention en particulier aux tabulations (indentation) qui doivent être scrupuleusement respectées si on veut que notre programme soit interprété correctement.

2 Boucles non conditionnelles

a Boucle "pour"

Les boucles non conditionnelles sont les boucles pour lesquelles on connaît à l'avance le nombre d'itérations. Elle sont décrites en pseudo-langage par « Pour … Faire … FinPour ».

En Python, la syntaxe est :

```
for <variable> in <iterable>:
    <instructions>
```

où `<iterable>` peut être une liste, une chaîne de caractères, un tuple, un ensemble, un dictionnaire… ou :

- `range(n)` pour les entiers entre 0 et $n-1$ (il y a donc n termes)
- `range(k,n)` pour les entiers entre k et $n-1$
- `range(k,n,p)` pour les entiers entre k et $n-1$ avec un pas de p.

Exemple

`range(2, 25, 2)` permet d'obtenir tous les nombres pairs entre 2 et 24.

Exemple

Calcul de la somme des entiers entre 1 et 150

```
somme=0
for i in range(1, 151): somme += i
```

b Itérables en compréhension

Une utilisation un peu particulière est de permettre[1] la définition d'itérables en compréhension :

```
>>> [i ** 2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> tuple(i ** .5 for i in range(10))
(0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979, 2.\
 449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0)
>>> {(i,j) for i in range(5) for j in range(6)}
{(1, 3), (3, 0), (2, 1), (0, 3), (2, 5), (4, 0), (1, 2), (3, 3), (4, 4), (1, 5), (0,\
  4), (2, 2), (4, 1), (1, 1), (3, 2), (0, 0), (4, 5), (1, 4), (0, 5), (2, 3), (4, 2)\
 , (1, 0), (3, 5), (0, 1), (3, 1), (2, 0), (4, 3), (3, 4), (0, 2), (2, 4)}
```

3 Boucles conditionnelles

Les boucles non conditionnelles sont les boucles pour lesquelles on sait à l'avance le nombre d'itérations. Elle sont décrites en pseudo-langage par « Tant que … Faire … FinTantQue ».

En Python, la syntaxe est :

1. Comme en maths !

```
while <condition>:
    <instructions>
```

Exemple

Que fait cette boucle ?

```
i = 132435
while i != 0:
    print(i % 10)
    i //= 10
```

IX FONCTIONS EN PYTHON

1 Définition d'une fonction

On peut définir en Python des fonctions demandant en entrée des arguments et renvoyant ou non quelque chose en sortie.

```
def <nom_de_la_fonction>(<argument1>, <argument2>,...):
    """Description de la fonction"""
    <instruction1>
    <instruction2>
    ...
    return <instruction>
```

Toujours bien documenter la fonction, l'instruction `return` peut être omise, tout ce qui se trouve après cette instruction n'est pas exécuté lors de l'appel à la fonction. On peut utiliser `return None` pour sortir de la fonction sans rien renvoyer ou ne pas mettre de `return` du tout.

Attention à l'indentation (tabulations de 4 espaces) qui doit être rigoureusement respectée pour être correctement interprété.

On peut rajouter dans le corps de la fonction une ligne `assert <test>` renvoyant une erreur si le test n'est pas vérifié. Cela permet par exemple de s'assurer que les données entrées comme paramètres de la fonction vérifient certaines conditions.

2 Typage des fonctions

Le **typage d'une fonction** est la donnée des types de ses arguments ainsi que du type de ce qu'elle retourne.

Une fonction correctement écrite devra en général retourner **des résultats toujours de même type**, afin de faciliter son utilisation ultérieure.

3 Portée des variables

a Variables locales

Les variables utilisées à l'intérieur d'une fonction sont en général des variables **locales** : elle n'existent que dans le contexte de l'exécution de la fonction et disparaissent lorsque l'on sort de celle-ci [1]. Si une variable du même nom existait hors-contexte, celle-ci ne sera pas modifiée.

1. On dit que les fonctions possèdent leur propre « espace des noms de variables ».

```
>>> x = 3
>>> def fonction():
        x = 2
        y = 3
        print(x, y)
>>> fonction()
2 3
>>> x
3

>>> y
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

b Variables globales

Il se peut que l'on ait besoin d'utiliser des variables définies hors-contexte à l'intérieure d'une fonction.

```
>>> x = 3
>>> def fonction():
        print(x)
>>> fonction()
3
>>> def fonction2():
        x += 1
        print(x)
>>> fonction2()
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    fonction2()
  File "<pyshell#30>", line 2, in fonction2
    x += 1
UnboundLocalError: local variable 'x' referenced before assignment
```

Dans la première fonction, on ne modifie pas `x` et il est bien pris en compte. Mais lorsque l'on essaye de le modifier dans la deuxième, on obtient une erreur.

Une solution est de déclarer la variable comme **globale** [1].

```
>>> x = 3

>>> def fonction():
        global x
        x += 1
        print(x)

>>> fonction() ; x
4
4
```

Mais attention, cela n'est pas recommandé. En général, on préfère passer les variables en arguments de fonctions.

1. Ce qui signifie que l'interpréteur doit aller chercher la variable dans l'espace des noms immédiatement supérieur.

c Cas des types mutables

Attention : ce qui vient d'être dit n'est valable que pour les types **non mutables**. Pour les variables de type mutable (par exemple les listes), le comportement est différent.

```
>>> L = [3]

>>> def fonction():
        L.append(2)

>>> fonction()

>>> L
[3, 2]
```

Ici, le contenu de la liste a bien pu être modifié même si la variable n'est pas déclarée comme globale.

Morale : ne jamais modifier les paramètres d'une fonction, sauf si c'est totalement maîtrisé et intentionnel !

d Passage par valeur / par référence

Une recommandation importante est d'éviter en général de modifier les arguments d'une fonction.
En fait les règles sont les mêmes qu'avec les variables extérieures :

- si elles sont de type non mutable, cela n'aura pas d'effet sur la variable extérieure passée en argument (car c'est la valeur qui est en argument, et non la variable)
- si elles sont de type mutable, le contenu sera effectivement modifié.

```
>>> L = [1, 2, 3]

>>> x = 1

>>> def fonction(liste, a):
        liste.append(4)
        print(liste)
        a += 1
        print(a)

>>> fonction(L, x)
[1, 2, 3, 4]
2

>>> L
[1, 2, 3, 4]

>>> x
1
```

Cela peut poser un problème lorsque l'on veut travailler sur une copie de la liste (ou de dictionnaire, ou d'ensemble...) et non la liste directement.

On rappelle qu'il est possible de créer une copie d'une liste grâce à `L[:]` ou `list(L)` (ou encore `L.copy()` en Python $\geqslant$ 3.3.)

4 Des fonctions dans les fonctions

Un cas un peu particulier de variable locale est celui des fonctions locales.
Une fonction dans une fonction ne sera définie que lors de l'exécution de celle-ci.

```
def selectionSort(L):
    """Tri selection"""
    def echange(L, i, j):
        """échange de L[i] et L[j]"""
        L[i], L[j] = L[j], L[i]
    def indiceMin(L, i):
        """Indice du minimum à partir du numéro i"""
        mini, indice=L[i], i
        for k in range(i+1, n):
            if mini>L[k]: mini, indice = L[k], k
        return indice
    n = len(L)
    for i in range(n):
        echange(L, i, indiceMin(L, i))
```

5 Fonctions lambda

Il peut parfois être pratique de définir une fonction en une ligne. Voilà comment procéder :

```
sqrt = lambda x:x**.5
```

Cela peut s'avérer très pratique dans certains contextes :

```
>>> foncList = [lambda n: n ** .5, lambda n: n, lambda n: n ** 2, lambda n: n ** 3,
                lambda n: 2 ** n]
>>> for f in foncList:
        print(f(10))
3.1622776601683795
10
100
1000
1024
>>> for f in foncList:
        print(f(100))
10.0
100
10000
1000000
1267650600228229401496703205376
```

X MANIPULATION DE FICHIERS

1 Objet-fichier

On peut facilement en python lire ou écrire dans des fichiers.

Pour cela, il faut au préalable créer un objet-fichier dont on connaît le nom (chaîne de caractères), avec un mode parmi

- `'r'` pour *read*
- `'w'` pour *write* (écrasement)
- `'a'` pour *append* (ajout à la suite)

Deux possibilités :

- En utilisant `with` qui gère l'ouverture et la fermeture du fichier (à préférer) :

```
with open(nomFichier, 'r') as fich:
    <instructions>
```

- En affectant classiquement une variable :
  Cela se passe avec une commande du type `fich = open(nomFichier, 'r')` par exemple.
  Une fois le traitement terminé, il faut refermer l'objet-fichier avec `fich.close()` pour libérer de la mémoire et/ou modifier/créer réellement le fichier.

Notons que par défaut le répertoire de travail est celui dans lequel le script se trouve, mais on peut le modifier à l'aide de la commande `chdir` du module `os`.

2 Écriture séquentielle

Pour écrire dans un objet-fichier, on utilise la méthode `write` :
`fich.write('ceci est un texte')` ajoute les caractères de l'argument à la suite de ce qui a déjà été ajouté (donc sans espaces, saut de ligne, etc.)

Rappelons qu'un saut de ligne dans une chaîne de caractère s'obtient avec `'\n'` (et `'\t'` pour une tabulation).

```
>>> with open('test.txt', 'a') as fich:
        fich.write('Un peu de texte ici...')
        fich.write("Et encore un peu là.\n")
        fich.write("Nous voilà à la ligne.")
22
21
22
```

3 Lecture séquentielle

Pour lire dans un objet-fichier, on peut utiliser :

- la méthode `read` : `fich.read()` renvoie la chaîne de tous les caractères du fichier, `fich.read(n)` renvoie les `n` suivant la position du "curseur", et place celui-ci après.
- la méthode `readline` : `fich.readline()` renvoie la chaîne de la ligne courante à partir du "curseur", et le déplace au début de la ligne suivante. Renvoie une chaîne vide si on atteint la fin du fichier.
- la méthode `readlines` : `fich.readlines()` renvoie liste de toutes les lignes (pratique pour des boucles...)
- on peut aussi parcourir chaque ligne sans avoir à créer une (éventuellement longue) liste avec `for ligne in fich:`

Remarque

Chaque ligne sauf éventuellement la dernière se termine par le caractère `'\textbackslash n'`. On peut s'en débarrasser avec `ligne[:-1]`. Notons que la méthode `rstrip` appliquée à une chaîne de caractère, sans argument, enlève tous les « caractères invisibles » (whitespaces) en fin de ligne.

```
>>> with open('test.txt', 'r') as fich: fich.read()

'Un peu de texte ici...Et encore un peu là.\nNous voilà à la ligne.'

>>> with open('test.txt', 'r') as fich: print(fich.read())

Un peu de texte ici...Et encore un peu là.
Nous voilà à la ligne.
```

```
>>> with open('test.txt', 'r') as fich: print(fich.readline())

Un peu de texte ici...Et encore un peu là.
```

```
>>> fich = open('test.txt', 'r')
>>> fich.readline()
'Un peu de texte ici...Et encore un peu là.\n'
>>> fich.readline()
'Nous voilà à la ligne.'
>>> fich.readline()
''
>>> fich.close()
```

```
>>> with open('test.txt', 'r') as fich:
        L = fich.readlines()
        print(L)
        print()
        for ligne in L:
            print("""==> La ligne \n****\n{}****
contient {} caractères !\n""".format(ligne,len(ligne)))

['Un peu de texte ici...Et encore un peu là.\n', 'Nous voilà à la ligne.']

==> La ligne
****
Un peu de texte ici...Et encore un peu là.
****
contient 43 caractères !

==> La ligne
****
Nous voilà à la ligne.****
contient 22 caractères !
```

4 Sérialisation des données avec le module Pickle

On a parfois besoin de sauvegarder autre chose que du texte !

Le module Pickle permet d'enregistrer les données concernant des structures de données de Python dans un fichier binaire, et bien sûr de pouvoir le lire depuis le fichier.

a Sérialisation

Le module `pcikle` possède une fonction `dump` permettant d'enregistrer dans un fichier binaire (et non un fichier texte) n'importe quelle structure de donnée Python.

Exemple :

```python
import pickle

L=[i ** 2 for i in range(100)]

with open('liste_carres.bin', 'wb') as fich: # b pour binaire
    pickle.dump(L, fich)                     # sérialisation
```

b Désérialisation

La fonction `load` du module `pickle` permet de lire les données stockées dans un tel fichier binaire.

```python
import pickle

with open('liste_carres.bin', 'rb') as fich: # Toujours le b...
    L = pickle.load(fich)                    # désérialisation
```

Présentation-type en Python

```
# nom_du_fichier.py
# auteur - date

#########
# Sujet #
#########

"""
Le mettre ici si ça vaut le coup... (ex : project euler...)
"""

############
# Modules #
############

from ... import ....
import ...
import ... as ...

#############
# Fonctions #
#############

def nom_fonction1(arguments):
    ...


def nom_fonction2(arguments):
    ...


###################
# Prog. principal #
###################

....
....
```