Introduction à la programmation Python

Emmanuel Fleury
<emmanuel.fleury@labri.fr>

LaBRI, Université de Bordeaux, France

28 août 2018

Notes

Plan

Notes

Plan

Notes

Langage Python

- Langage de script semi-interprété ;
- Typage dynamique ;
- Langage multi-paradigmes
  (impératif, orienté-objet et fonctionnel).
- **Version 1.0** : 1990
- **Version 2.0** : 2000
- **Version 3.0** : 2008
  (les exercices sont en Python 3)
- **Auteur** : Guido van Rossum (NL)

Le Zen du Python (PEP20 [1])

La beauté est préférable à la laideur.
L'explicite est préférable à l'implicite.
La simplicité est préférable à la complexité.
Le complexe est préférable au compliqué.
La lisibilité compte.

1. PEP = Python Enhancement Proposals

Notes

Documentation Python en-ligne

Documentation Python 2 (en)
`https://docs.python.org/2/`

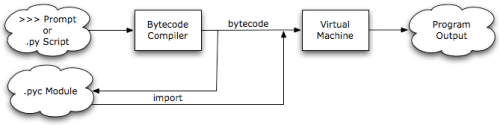
Documentation Python 3 (en)
`https://docs.python.org/3/`

Le tutoriel Python (fr)
`http://www.afpy.org/doc/python/2.7/tutorial/`

Attention !
Consultez la documentation le plus souvent possible !
Et lisez le tutoriel en entier une première fois.

Notes

Exécuter un programme Python



Interpréteur

```
$> python3
Python 3.4.2 (default, Nov 13 2014, 07:01:52)
Type "help", "copyright", ... for more information.
>>> print("Hello World!")
Hello World!
>>> quit()
$>
```

Script

Fichier 'HelloWorld.py' :

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

```
$> python3 HelloWorld.py
Hello World!
$>
```

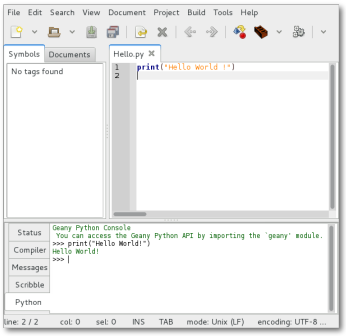
Apparition éventuelle de :
'HelloWorld.pyc'.

Notes

Environnement de développement

Geany

- Éditeur de texte :
  - Coloration syntaxique.
  - Indentation automatique (mettre à 4 espaces).
  - Repliage de code.
  - Auto-complétion.
  - Support pour le langage Python (et d'autres encore).
- Console Python.
- Exécution des scripts.
- Navigation dans le code.
- Greffons faciles à ajouter.
- Multi-plates-formes (Linux, MacOS X et MS-Windows).



Notes

Que peut-on faire avec Python ?

- **Développement Web**
  Django, Pyramid, Zope, Plone, Flask, ...
- **Bases de données**
  SQLAlchemy, DB-API, Pandas, ...
- **Calcul Scientifique**
  Sage, Numpy, Scipy, Simpy, Matplotlib...
- **GUIs**
  PyGtk, PyQt, TkInter, ...
- **Manipulation d'images**
  Pillow, OpenCV-Python, ...
- **Et plein d'autres...**

Notes

Plan

Notes

Les variables

En Python, les **variables** sont des **références** vers des objets qui résident dans la mémoire du programme. Un peu comme des étiquettes que l'on accroche à divers emplacements mémoires. Lorsqu'on écrit dans une variable, on dit que l'on fait une **affectation** (par exemple : `a = 3`).

Attention !

Les variables Python sont particulières, les choses sont différentes dans les autres langages de programmation.

Notes

Les nombres

Il existe deux types de nombres en Python, les **entiers** (`int`) et les **flottants** (`float`).

```
>>> a = 2          >>> type(2//5)    >>> 1/0
>>> type(a)        <class 'int'>     Traceback (most recent call last):
<class 'int'>      >>> print(2//5)      File "<stdin>", line 1, in <module>
>>> b = 5.0        0                 ZeroDivisionError: division by zero
>>> type(b)        >>> type(2/5)      >>> 1//0
<class 'float'>    <class 'float'>   Traceback (most recent call last):
>>> type(a * b)    >>> print(2/5)      File "<stdin>", line 1, in <module>
<class 'float'>    0.4               ZeroDivisionError: integer division by zero
```

Les **opérateurs arithmétiques** sont tous les opérateurs qui agissent sur les nombres.

Syntaxe	Sémantique
`x + y`	Addition
`x - y`	Soustraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Division entière
`x % y`	Modulo
`x ** y`	Puissance

Syntaxe	Sémantique
`(x)`	Groupe l'expression
`-x`	Négation
`abs(x)`	Valeur absolue
`int(x)`	Conversion en entier
`float(x)`	Conversion en flottant

Syntaxe	Sémantique
`~x`	Négation bit-à-bit
`x & y`	Et bit-à-bit
`x | y`	Ou bit-à-bit
`x ^ y`	Ou-exclusif bit-à-bit
`x >> y`	Décalage vers la droite
`x << y`	Décalage vers la gauche

Notes

Les booléens et 'None'

Les **variables booléennes** peuvent seulement valoir soit '`True`' (vrai), soit '`False`' (faux). Elles servent à représenter le **résultat d'un test**.

Les **opérateurs booléens** sont tous les opérateurs qui retournent vrai ou faux.

Syntaxe	Sémantique
`(x)`	Groupe l'expression
`not x`	Négation
`x or y`	Ou-logique
`x and y`	Et-logique

Syntaxe	Sémantique
`x < y`	Inférieur strict
`x <= y`	Inférieur ou égal
`x > y`	Supérieur strict
`x >= y`	Supérieur ou égal
`x == y`	Égal
`x != y`	Différent

'`None`' dénote une **variable "*vide*"**. Ou, plus précisément, le fait que la référence contenue par la variable pointe sur `None` pour représenter qu'elle ne contient rien. Il est juste possible de tester si une variable pointe dessus ou pas.

```
if x is not None:
    # In this block we are sure that x is not None
    foo(x)
```


Notes

Les chaînes de caractères

Les **chaînes de caractères** ('str') sont le plus souvent utilisées pour interagir avec l'utilisateur ou avec les fichiers. Elles sont constituées d'un enchaînement ordonné de caractères et encadrées indifféremment soit par des apostrophes (''), soit par des guillemets ('"').
Par exemple : 'abcd' ou "abcd".

Les opérateurs sur les chaînes permettent de concaténer, filtrer ou extraire des parties des chaînes de caractères entres elles.

Syntaxe	Sémantique
len(s)	Longueur de la chaîne s
s1 + s2	Concaténation de 's1' et 's2'
s * n	Répétition 'n' fois de la chaîne 's'
s in t	Teste si la chaîne 's' est présente dans 't'.
s[n]	Extraction du n-ième caractère en partant du début de la chaîne
s[-n]	Extraction du n-ième caractère en partant de la fin de la chaîne
s[:n]	Extraction des 'n' premiers caractères de la chaîne
s[n:]	Extraction des derniers caractères de la chaîne à partir du n-ème
s[-n:]	Extraction des caractères des 'n' derniers caractères de la chaîne
s[n:m]	Extraction des caractères se trouvant entre le n-ième et le m-ième

Notes

Les conversions entre les types

La **conversion de types** permet de passer d'un type à l'autre. Par exemple, lorsqu'il s'agit de convertir une chaîne de caractères en un entier ou un flottant. La règle générale est de se servir du nom du type avec des parenthèses : int(a), float(b), ...

Syntaxe	Sémantique
bool(var)	Converti 'var' en un booléen
int(var)	Converti 'var' en un entier
float(var)	Converti 'var' en un flottant
str(var)	Converti 'var' en une chaîne de caractères

```
>>> bool(10)
True
>>> bool(0)
False
>>> bool("abc")
True
>>> bool("")
False
```

```
>>> int("10")
10
>>> int(10.65)
10
>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

```
>>> float(10.567889)
10.567889
>>> float(10)
10.0
>>> float("10.5")
10.5
```


Notes

Interlude : print() et input()

print(msg) imprime à l'écran une chaîne de caractères donnée en argument.

```
>>> print("Hello")
Hello
>>> a = 10
>>> print("Hello", 1, 1.0, a)
Hello 1 1.0 10
>>> print("mesage: %s %f %i" % ('Hello', 1.0, a))
message: Hello 1.000000 10
```

input(msg) permet de récupérer une chaîne de caractères tapée par l'utilisateur.

```
>>> a = input("Donnez un chiffre: ")
Donnez un chiffre: 10
>>> print(a)
10
>>> type(a)
<class 'str'>
>>> a = a + 1
TypeError: Cant convert 'int' object to str implicitly
>>> a = int(a)
>>> type(a)
<class 'int'>
```


Notes

Plan

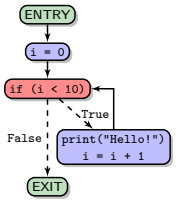
Notes

Les blocs

Un **bloc d'instructions** est une **unité de programmation** qui regroupe une **séquence ordonnée d'instructions** qui s'exécutent les unes après les autres. Un programme impératif va faire s'enchaîner les blocs d'instructions en les branchant les uns aux autres via des **structures de contrôle** telles que `if`, `for`, `while`, ...

```
# commentaire
instruction(bloc1)
statement arguments:
    instruction(bloc2)
    instruction(bloc2)
```

```
# Loop
i = 0
while (i < 10):
    print("Hello!")
    i = i + 1
```



Syntaxe Python en trois règles

1. Tout ce qui suit un caractère '#' est considéré comme un **commentaire** et est ignoré ;
2. Les instructions qui se trouvent au **même niveau d'indentation** font parti d'un même bloc ;
3. Les blocs peuvent être coordonnés entre eux via des **structures de contrôle** qui ouvrent un nouveau bloc en se terminant par un caractère ':' (deux points).

Notes

Les branchements

Les **tests** permettent de choisir un chemin d'exécution en fonction de la valeur de certaines variables. En Python, les tests se font avec trois mots clefs :

- `if` (premier test),
- `elif` (tests additionnels),
- `else` (tous les autres cas).

```
if (condition1):
    instructions1
elif (condition2):
    instructions2
else:
    instruction3
```

```
if (m < 10):
    print ("m < 10")
elif (m == 11):
    print ("m == 11")
else:
    print("m > 11")
```

Exemple

```
x = int(input("Donnez un nombre: "))

if (x > 10) or (y == 57):
    print("x>10")
elif (x == 9):
    print("x==9")
else:
    print("All other cases")
```

Que fait ce programme lorsque :

1. x = 8 et y = 6 ;
2. x = 9 et y = 57 ;
3. x = 9 et y = 18.

Notes

Les boucles 'for'

En Python, `for` est un itérateur sur des collections d'objets. Il est utilisé pour parcourir, dans l'ordre s'il y en a un, les éléments de la collection.

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

```
>>> for i in range(5):
...     print("Hello", i)
...
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
```

```
>>> for a in "Hello":
...     print(a)
...
H
e
l
l
o
```

La fonction 'range(start, stop, step)'

Permet de créer une collection de nombres pour itérer dessus.

```
range(5)        --> 0, 1, 2, 3, 4
range(2, 5)     --> 2, 3, 4
range(2, 5, 2)  --> 2, 4
```


Notes

Les boucles 'while'

Les boucles `while` permettent de continuer la boucle tant que le test est vrai. Il ne s'agit donc pas forcément d'itérer mais, le plus souvent, de chercher quelque chose dans une itérable ou de faire des boucles infinies.

```
# initialisation
while (test):
    instruction1
    instruction2
```

```
i = 0
while (i < 10):
    print("Hello World!")
    i = i + 1
```

'break', 'continue' et 'else'

Ces mots clefs sont utilisables dans les boucles 'for' et les boucles 'while'.

- Le mot clef 'break' utilisé dans une boucle permet de sortir de la boucle la plus interne ou l'on est.
- Le mot clef 'continue' permet de passer directement au tour de boucle suivant sans exécuter les instructions présentes dans la boucle.
- Mettre un 'else' en fin de boucle permet de rajouter des instructions qui seront executées si aucun break n'a été déclenché.

Notes

Interlude : Fonctions sur les chaînes

- `mystr.capitalize()` : Retourne une copie de la chaîne de caractères avec la première lettre en majuscule et les suivantes en minuscule.
- `mystr.lower()` : Retourne une copie de la chaîne de caractères toutes en minuscules.
- `mystr.upper()` : Retourne une copie de la chaîne de caractères toutes en majuscules.

```
>>> mystr = "cAchAlOt"
>>> mystr.capitalize()
'Cachalot'
>>> mystr.lower()
'cachalot'
>>> mystr.upper()
'CACHALOT'
```

- `mystr.count(motif)` : Compte le nombre de fois que le motif apparaît dans `str`.
- `mystr.find(motif)` : Retourne le premier index où l'on trouve le motif recherché.
- `mystr.replace(motif1, motif2)` : Remplace le `motif1` par le `motif2`.

```
>>> mystr = "consonance"
>>> mystr.count("on")
2
>>> mystr.find('a')
6
>>> mystr.replace('on', 'in')
'cinsinance'
```


Notes

Plan

Notes

Les tuples

Les tuples permettent de **grouper des éléments de différentes natures** pour créer une **structure** qui les englobe. On note un tuple avec des parenthèses (parfois optionnelles) et des virgules pour séparer ses éléments :

```
a = (1, 2, 'abc')
b = 1, 2, 'abc'
```

Les opérateurs sur les tuples permettent de concaténer, filtrer ou extraire des parties des tuples entres eux. Ils sont quasiment identiques à ceux des chaînes de caractères.

Syntaxe	Sémantique
`len(t)`	Nombre d'éléments dans le tuple
`t1 + t2`	Concaténation de `t1` et `t2`
`t * n`	Répétition `n` fois du tuple `t`
`t[n]`	Extraction du `n`-ième élément du tuple en partant de la gauche
`t[-n]`	Extraction du `n`-ième élément du tuple en partant de la droite
`t[:n]`	Extraction des `n` premiers éléments du tuple
`t[n:]`	Extraction des éléments `n` à la fin du tuple
`t[-n:]`	Extraction des `n` derniers éléments du tuple
`t[n:m]`	Extraction des éléments qui sont entre `n` et `m`

Notes

Les listes

Les **listes** servent à stocker des **collections ordonnées d'éléments** avec possibilité de les étendre ou de les manipuler facilement.
On note une liste avec des crochets :

```
a = [ 3, 5, 17, 26 ]
```

Les opérateurs sur les listes sont quasiment identiques à ceux des chaînes de caractères.

Syntaxe	Sémantique
`len(l)`	Nombre d'éléments dans la liste
`del(l[n])`	Détruit l'élément `n` de la liste `l`
`l.clear()`	Vide la liste `l`
`l.copy()`	Renvoie une copie de la liste `l`
`l.append(e)`	Ajoute l'élément `e` à la fin de la liste `l`
`l.insert(n, e)`	Insère l'élément `e` à la position`n` dans la liste `l`
`l1 + l2`	Concaténation de `l1` et `l2`
`l * n`	Répétition `n` fois de la liste `l`
`l[n]`	Extraction du `n`-ième élément en partant du début de la liste
`l[-n]`	Extraction du `n`-ième élément en partant de la fin de la liste
`l[:n]`	Extraction des `n` premiers éléments de la liste
`l[n:]`	Extraction des éléments `n` à la fin de la liste
`l[-n:]`	Extraction des `n` derniers éléments de la liste
`l[n:m]`	Extraction des éléments qui sont entre `n` et `m`

Notes

Les ensembles

Les **ensembles** sont des listes non ordonnées d'éléments, ils sont extensibles et ne gardent aucune notion d'ordre entre les éléments qu'ils contiennent :

```
a = {'mercredi', 'mardi', 'dimanche'}
```

Les seules opérations que l'on peut faire sur ces ensembles sont toutes les opérations ensemblistes (appartenance d'un élément à l'ensemble, intersection, différence, ...).
Par exemple : `'lundi' in a`

Les opérateurs sur les ensembles permettent de les manipuler comme en mathématique.

Syntaxe	Sémantique
`len(s)`	Taille de l'ensemble 's'
`s.add(e)`	Ajoute 'e' à 's'
`s.remove(e)`	Retire 'e' de 's' (erreur si 'e' ∉ 's')
`s.discard(e)`	Retire 'e' de 's'
`s.clear()`	Vide l'ensemble 's'
`s1 | s2`	Union de 's1' et 's2'
`s1 & s2`	Intersection de 's1' et 's2'
`s1 - s2`	Différence de 's1' et 's2'
`s1 ^ s2`	Différence exclusive de 's1' et 's2'

Syntaxe	Sémantique
`e in s`	'e' ∈ 's'
`e not in s`	'e' ∉ 's'
`s1 == s2`	's1' égal 's2'
`s1 != s2`	's1' différent de 's2'
`s1 < s2`	's1' est inclus (strict) dans 's2'
`s1 <= s2`	's1' est inclus ou égal à 's2'
`s1 > s2`	's2' inclus (strict) dans 's1'
`s1 >= s2`	's2' inclus ou égal à 's1'
`s1.isdisjoint(s2)`	's1' disjoint de 's2'
`s1.issubset(s2)`	's1' inclus dans 's2'
`s1.issuperset(s2)`	's1' inclus 's2'

Notes

Les dictionnaires

Les dictionnaires sont des **ensembles associatifs** avec en entrée les **clefs** qui sont chacune associée avec une **valeur**. Les dictionnaires sont encadrés par des accolades et chaque couple 'clef:valeur' est séparé par une virgule :

```
a = {'mardi':2, 'dimanche':7}
```

Et, on accède à une valeur ainsi : `a['mardi']`

Les opérateurs sur les dictionnaires permettent de faire des recherches sur les clefs, d'ajouter des entrées ou d'en modifier certaines.

Syntaxe	Sémantique
`len(d)`	Nombre d'éléments du dictionnaire
`del(d[k])`	Détruit l'entrée 'k' du dictionnaire
`d.clear()`	Vide le dictionnaire 'd'
`d.copy()`	Renvoie une copie du dictionnaire 'd'
`d[k]`	Accède à la valeur de la clef 'k'
`d[k] = v`	Assigne la valeur 'v' à la clef 'k'
`k in d`	Teste si la clef 'k' est présente dans le dictionnaire 'd'
`k not in d`	Teste si la clef 'k' n'est pas présente dans le dictionnaire 'd'

Notes

Interlude : Boucles 'for' avancées

Tuples, listes et ensembles

```
>>> mytuple = (1, 2, 3)
>>> for item in mytuple:
...     print(item)
1
2
3
>>> mylist = [1, 2, 3]
>>> for item in mylist:
...     print(item)
1
2
3
>>> myset = {1, 2, 3}
>>> for item in myset:
...     print(item)
1
2
3
```

Dictionnaires

```
>>> mydict = {1:'a', 2:'b', 3:'c'}
>>> for k, v in mydict.items():
...     print("%i:%s" % (k, v))
1:a
2:b
3:c
>>> for k in mydict.keys():
...     print(k)
1
2
3
>>> for v in mydict.values():
...     print(v)
a
b
c
```


Notes

Plan

Notes

Fonctions

Les fonctions servent à rendre votre programme modulaire et à éviter de répéter plusieurs fois le même code. Une fonction est définie par **son prototype** (c'est à dire : **son nom**, **ses arguments** et **son type de retour**) et, évidemment, aussi par **son code**.

```
# Function definition
def func_name(args):
    instructions
    return x

# Main program
instructions
func_name(args)
instructions
```

```
def foo(a, b, c):
    x = (a + b) * c
    if (x == a + b):
        return x
    return

x = 10
y = 2.0
z = foo(57, x, y)
```

Les arguments

Les **arguments** sont des **variables passées en paramètres** à la fonction. Ils ne sont pas typés *a priori* mais sont substituées **dynamiquement** dans le code de la fonction lors de l'exécution. Dans l'exemple ci-dessus, les variables `a` et `b` peuvent être des booléens, des entiers, des flottants, des tuples, des listes ou des dictionnaires.

Le type de retour

Le **type de retour** correspond à **ce que retourne la fonction**. Le mot clef '`return`' permet de spécifier **la fin de la fonction** et **la variable qu'elle retourne**. Le langage Python accepte qu'une fonction retourne plusieurs types différents. Dans l'exemple ci-dessus, la fonction retourne une variable du même type que `a` ou `b` ou bien `None`.

Notes

Passage par valeur

Le **passage par valeur** signifie que seules **des copies des valeurs sont utilisées dans les fonctions** mais pas les variables originales. Les changements effectués sur les variables ne sortent pas de la fonction. En fait, la seule façon de récupérer les changements opérés sur la variable est de la passer via le '`return`'.

Voici deux exemples qui explicitent le fonctionnement du passage par valeur et comment utiliser le '`return`'.

```
def foo(s):
    s = "I am from foo() !"
    print(s)

s = "I am from main() !"
foo(s)
print(s)
```

```
I am from foo() !
I am from main() !
```

L'assignation de 's' à l'intérieur de `foo()` n'est plus visible à la sortie de la fonction.

```
def foo(s):
    s = "I am from foo() !"
    print(s)
    return s

s = "I am from main() !"
s = foo(s)
print(s)
```

```
I am from foo() !
I am from foo() !
```

On réassigne 's' dans la fonction principale grâce au retour de la fonction.

Notes

Variables locales / Variables globales

La **portée d'une variable** (*variable scope*) correspond à l'ensemble des lignes dans lesquelles la variable peut être appelée. Lorsqu'on est en dehors de la portée d'une variable et que l'on fait appel à elle, le programme provoque une erreur :

```
def foo():
    s = "Me too."
    print(s)
s = "I am 's'."
foo()
print(s)
```

```
Me too.
I am 's'.
```

```
def foo():
    print(s)
    s = "Me too."
s = "I am 's'."
foo()
print(s)
```

```
UnboundLocalError: local variable 's' referenced before assignment
```

Une variable est dites "**globale**" si **sa portée recouvre l'ensemble du programme**.
Une variable est dites"**locale**" si **sa portée recouvre seulement une partie du programme**.

```
def foo():
    s = "Me too."
    print(s)

s = "I am 's'."
foo()
print(s)
```

```
Me too.
I am 's'.
```

```
def foo():
    global s # Déclare 's' comme globale
    print(s)
    s = "Me too."
s = "I am 's'."
foo()
print(s)
```

```
I am 's'.
Me too.
```


Notes

La valeur de retour

La valeur de retour d'une fonction est signalée par le mot-clef `return`.
Il n'est **pas obligatoire** (car implicite à la fin du bloc qui constitue la fonction).
Mais, il sert à **marquer la fin de la fonction** même si le bloc n'est pas achevé.

```
def no_return():
    print("No return")
```

```
def with_return(x):
    return 10*x
```

```
def empty_return(x):
    if x:
        return
    print("Empty return")
```

```
def multiple_values(x, y):
    x = 2*x + y
    y = y - x
    return (x, y)
```

On peut l'utiliser **vide**, pour retourner `None`, ou bien avec **une valeur** ou **une variable** pour retourner un contenu ou sous la forme d'**un tuple**, si on désire retourner un ensemble de variables ou de valeurs.

Notes

Les fonctions récursives

Une fonction est dites **récursive** si **elle s'appelle elle-même** pour construire le résultat qu'elle retourne.

Une fonction récursive est dites **récursive terminale** si **tout appel récursif est de la forme** `return f(...)` (et pas autre chose en dehors de l'appel à la fonction elle-même). C'est à dire que le calcul final de la fonction récursive se fait à la terminaison de celle-ci sans avoir à remonter la chaîne des appels récursifs.

```
fact(4) = 4 * fact(3) = 4 * 3 * fact(2) = 4 * 3 * 2 = 24
fact(4, 1) = fact(3, 4) = fact(2, 4*3) = fact(1, 4*3*2) = 24
```

```python
def fact1(n):
  res = 1
  # Partie itérative
  for i in range(2,n+1):
    res = res * i
  return res
```

Factorielle version itérative.

```python
def fact2(n):
  # Fin récursion
  if (n < 2):
    return 1
  # Partie récursive
  return n * fact2(n-1)
```

Factorielle version récursive non-terminale.

```python
def fact3(n, result):
  # Fin récursion
  if (n < 2):
    return result
  # Partie récursive
  return fact3(n-1,result*n)
```

Factorielle version récursive terminale.

Notes

Arguments nommés

Les **arguments nommés** permettent de donner des **noms** et des **valeurs par défaut** à certains arguments d'une fonction. Les arguments nommés deviennent **optionnels** lors de l'appel à la fonction.

```python
def foo(a, b, c=0, d=10):
    return a + b - c * d

x = foo(10, 5)
y = foo(5, 10, d=7)
z = foo(5, 10, d=7, c=7)

print(x, y, z)
```

```
15 15 -34
```

```python
def foo(a, b, c=0, d=10):
    return a + b - c * d

t = foo(c=2, 5, 10, d=7)

print(t)
```

```
  File "Example.py", line 6
    t = foo(c=2, 5, 10, d=7)
                ^
SyntaxError:
non-keyword arg after keyword arg
```

Les arguments non-nommés doivent être placés, toujours dans le même ordre, au début de la fonction et les arguments nommés toujours à la fin. Si ce n'est pas le cas, une erreur '`non-keyword arg after keyword arg`' est levée.

Notes

Fonctions variadiques

Une **fonction variadique** est une fonction dont **le nombre d'arguments est variable**. C'est à dire que l'on peut y mettre le nombre d'arguments que l'on veut (mais pas n'importe quoi !).

En Python, marquer un des arguments d'une fonction par une étoile (`*`) permet de signaler que cet argument sera un tuple dont on ne connait pas *a priori* le nombre d'éléments. Un exemple classique de fonction variadique est la fonction `print()`.

```python
def foo(s, *args):
    print(s % args)

foo("Test %i %f %s", 10, 10.5, "AAAAA")
foo("Test %i", 10)
```

```
Test 10 10.500000 AAAAA
Test 10
```


Notes

Documenter ses fonctions

Les `docstrings` sont des chaînes de caractères qui suivent immédiatement le '`def`' et qui constituent **la documentation** de la fonction en question. Cette documentation permet de **comprendre ce que fait la fonction et d'aider à l'utiliser correctement**.

Le PEP8 recommande de faire une documentation pour toutes les fonctions.

```
>>> def example(var):
...     """This is the docstring of the function example(var).
...        It is written on two lines."""
...     return 10
...
>>> help(example)
Help on function example in module __main__:

example(var)
    This is the docstring of the function example(var).
    It is written on two lines.
>>>
```


Notes

Interlude : `enumerate()` et `zip()`

Les fonctions `enumerate()` et `zip()` sont utiles pour les itérations sur des structures.

- `enumerate()` : Associe un index sur la structure lorsqu'il n'y en a pas.
- `zip()` : Permet d'itérer deux structures en parallèle.

`enumerate()`

```
myset = {'a', 'w', 'd'}

for index, item in enumerate(myset):
    print (index, item)
```

```
0 a
1 d
2 w
```

`zip()`

```
mylist1 = [5, 3, 9, 4, 1]
mylist2 = ['a', 'w', 'd', 'e']

for index, char in zip(mylist1, mylist2):
    print (index, char)
```

```
5 a
3 w
9 d
4 e
```


Notes

Plan

Notes

Les modules

Les **modules Python** permettent de **grouper** et de rendre facilement **réutilisable** du code Python. Une fois placé dans un module, le code est facilement importable au sein des programmes que vous pourrez écrire.

```
import module1, module2 # Importe les modules
...
from module import func1, func2 # Importe les fonctions
...
from module import * # Importe toutes les fonctions du module
...
```

```
>>> l = [ 1, 2, 3, 4, 7 ]
>>> choice(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'choice' is not defined
>>> random.choice(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
>>> import random
>>> random.choice(l)
3
```

```
>>> choice(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'choice' is not defined
>>> from random import choice
>>> choice(l)
4
>>> from random import *
>>> choice(l)
7
```


Notes

Écrire et utiliser des modules

Écrire un module est facile, il suffit de **créer un fichier au nom du module** que l'on désire créer (suivi de '.py'), puis d'y **inclure les fonctions désirées**.

```
# Fichier hello.py
def myhello():
    print("Hello World!")
```

```
>>> import hello
>>> hello.myhello()
Hello World!
```

Par défaut, l'interpréteur Python cherche le module dans le répertoire où il exécute les scripts, puis dans les endroits "*habituels*". Si jamais votre module se trouve dans un autre répertoire, vérifiez que la variable d'environnement `PYTHONPATH` contient bien le chemin vers le répertoire en question. Le message d'erreur en cas de problème est le suivant :

```
>>> import my_module
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'my_module'
```

Notez aussi que si du code Python se trouve en dehors de toute fonction dans le module, il sera executé à son chargement. Pour éviter que cela se produise, il faut "*protéger*" le code avec :

```
if __name__ == "__main__":
    # code exécuté seulement si on lance le fichier '.py',
    # pas si on l'importe dans un autre fichier.
    ...
```


Notes

Les packages

Les **packages** sont **des ensembles hiérachisés de modules** qui permettent d'accéder à une liste cohérente de fonctionnalités. On les utilise **exactement comme les modules** mais avec la possibilité de **spécifier la hiérarchie interne**.

```
# Importe le module du sous-package
import package.subpackage.module
...
# Importe les fonctions du module du sous-package
from package.subpackage.module import func1, func2
...
# Importe toutes les fonctions du module du sous-package
from package.subpackage.module import *
...
```

Les modules à l'intérieur des packages sont hiérarchisés suivant les répertoires. Afin de marquer les packages, on doit rajouter un fichier **__init__.py** pour ne pas les confondre à de simples répertoires. Ces fichiers contiennent le code du package, mais sont vides si le package est vide.

```
mypackage/
+-- subpackage/
|   +-- mymodule.py # myfunc()
|   \-- __init__.py
\-- __init__.py
```

```
>>> import mypackage.subpackage.mymodule
>>> mypackage.subpackage.mymodule.myfunc()
```


Notes

Interlude : Gérer les Packages

Python intègre le module 'venv' qui permet de créer un environnement virtuel isolé du reste du système dans lequel on peut installer des logiciels en tant que simple utilisateur.

```
$> python -m venv myvenv/
$> cd myvenv
$> ls
bin/  include/  lib/  lib64/
$> source bin/activate
(myenv) $> echo $PATH
.../myvenv/bin:...
(myenv) $> echo $VIRTUAL_ENV
.../myvenv
(myvenv) $> deactivate
$>
```

`pip` est un outil de gestion des packages Python. Il permet d'installer des packages, de les mettre à jours, et bien d'autres choses. Lorsqu'on développe un projet, l'idéal est de rassembler les packages dont on a besoin au sein d'un environnement virtuel.

```
(myvenv) $> pip install django
Downloading/unpacking django
  Downloading Django-1.7.3-py2.py3-none-any.whl (7.4MB): 7.4MB downloaded
Installing collected packages: django
Successfully installed django
Cleaning up...
(myvenv) $> pip freeze
Django==1.7.3
```


Notes

Plan

Notes

Paradigmes de programmation

La **programmation orienté objet** est un **paradigme** de programmation. Les trois paradigmes les plus connus sont :

- **Programmation Impérative** : Le programme est vu comme une somme d'instructions chacune agissant sur la mémoire et qui, mis bout à bout, décrivent l'algorithme (c'était le paradigme utilisé jusqu'ici).
- **Programmation Orienté-Objet** : Le programme est vu comme une somme d'objets qui communiquent et agissent les uns sur les autres, ce sont ces interactions qui décrivent l'algorithme.
- **Programmation Fonctionnelle** : Le programme est vu comme un empilement d'appels à des fonctions (au sens mathématique du terme) qui forment l'algorithme.

Mais d'autres existent : programmation logique, programmation procédurale, ...

Notes

Principes fondamentaux

université de BORDEAUX

Pour se mettre en place, la **programmation orienté objet** nécessite un certain nombre de fonctionnalités qui doivent pouvoir se trouver dans le langage. Notamment, on doit pouvoir :

1. Définir des **classes** avec des **attributs** et des **méthodes**.
2. **Instancier** des **objets** à partir d'une classe et le faire communiquer avec d'autres objets.
3. Utiliser du **polymorphisme** et de l'**introspection** sur les objets.
4. Et enfin, faire de l'**héritage** et de la **redéfinition** entre les différentes classes.

Notes

Les classes et les objets (1/2)

université de BORDEAUX

Un **objet** est un conteneur qui possède un ensemble de données qui lui sont propres (son **état**) et un ensemble de fonctions (son **comportement**). Un objet est l'**instance** d'une classe.

Une **classe** est la définition d'un **objet**. Elle sert à pouvoir savoir ce que va **contenir** l'objet et ce qu'il va pouvoir **faire** une fois crée à partir de la classe.

- **Les attributs** : Ensemble des variables attachées à l'objet dont les valeurs représente l'état de celui-ci.
- **Les méthodes** : Ensemble des fonctions attachées à l'objet qui permettent d'agir sur les attributs de l'objet ou de lui donner des comportements.

Notes

Les classes et les objets (2/2)

université de BORDEAUX

- **Standard de codage** (CamlCase) : Le nom d'une classe est en CamlCase et commence par une majuscule. Par exemple : `ClassName` ou `BlueCircle`.
- **Notation pointée** : On parle de **notation pointée** pour désigner l'accès à un attribut ou à une méthode qui est propre à un objet.
  Par exemple : `myobject.var` ou `myobject.mymethod(1,3)`.
- `self` : Par convention, la variable `self` utilisée à l'intérieur du code d'une classe fait référence à l'objet lui-même. Par exemple : `self.var`.
- **Variables privées** : Les variables privées sont des variables qui ne sont pas accessibles depuis l'extérieur de l'objet. En Python, on note les variables privées en les préfixant par "`__`". Par exemple : `__var`.
- **Constructeur** : Un constructeur est une méthode un peu à part car elle est appelée à la création de l'objet pour l'initialiser. On utilise le nom de la classe pour l'appeler, mais dans le code de la classe on l'appelle "`__init__()`".
- **Méthodes ou variables "de Classe"** : Un attribut ou une méthode est dites "**de classe**" (ou "**statique**") si elle dépend de la classe et non de l'objet. On les appelles aussi parfois des variables ou des méthodes de **classe**.

Notes

Un exemple complet

université de BORDEAUX

```
class RedCircle(object):
    color = 'Red' # Static/Class variable

    # Constructor
    def __init__(self, x, y, radius):
        self.x = x # Field
        self.y = y # Field
        self.radius = radius # Field

    # Method
    def print_color(self):
        print(RedCircle.color)

# Main program
myobject = RedCircle(1, 2, 5)

print(type(RedCircle))
print(type(myobject))
print(myobject.x)
print()

myobject.print_color()
print(myobject.color)
print()

myobject.color = 'Green'
myobject.print_color()
print(myobject.color)
```

```
$> python3 ./RedCircle.py
<class 'type'>
<class '__main__.RedCircle'>
1

Red
Red

Red
Green
$>
```


Notes

Héritage

L'**héritage** est un mécanisme particulier à la programmation orientée objet qui permet de factoriser efficacement du code. Lorsqu'on **hérite** d'une classe, on récupère tous ses attributs et toutes ses méthodes. On peut alors soit en ajouter pour **spécialiser** la classe, soit en modifier une partie par **redéfinition**.

Spécialisation

Par exemple, on définit la classe `Circle`, puis on la spécialise en créant la classe `RedCircle` en lui ajoutant un attribut `color` et une méthode `print_color`.

```
class Circle(object):
    def __init__(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius
```

```
class RedCircle(Circle):
    color = 'Red'

    def print_color(self):
        print(RedCircle.color)
```

Redéfinition

Dans cet exemple, on redéfini le constructeur pour incorporer la couleur. On utilise la fonction `super()` pour faire référence à la classe mère.

```
class Circle(object):
  def __init__(self, x, y, radius):
    self.x = x
    self.y = y
    self.radius = radius
```

```
class ColoredCircle(Circle):

  def __init__(self,x,y,radius,color):
    super().__init__(x, y, radius)
    self.color = color
```


Notes

Interlude : Les exceptions

Les **exceptions** permettent d'introduire une gestion des erreurs qui ne se mélange pas avec le code de retour des fonctions. Les objets ou n'importe quelle fonction, peuvent soit "**lever**" une exception (`raise...`), soit l'"**attraper**" (`try...except...`).

Si une exception est levée mais attrapée nulle part dans le programme, le programme s'arrête en signalant quelle exception est responsable de l'arrêt et où elle a été lancée.

Exemple de code

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
for cls in [B, C, D]:
    try:
        raise cls()
    except D as e:
        print("D", e)
    except C:
        print("C")
    except B:
        print("B")
    else:
        print("Other exceptions")
    finally:
        print("finally")
```

Message d'erreur

```
>>> e = Exception()
>>> def foo():
...     raise(e)
...
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in foo
Exception
```

Résultat de l'exemple

```
B
finally
C
finally
D
finally
```


Notes

Plan

Notes

Principes fondamentaux

La **programmation fonctionnelle** est un paradigme de programmation basé sur les **fonctions et leurs compositions**. Un **programme fonctionnel** est, donc, **la somme des fonctions qui le composent et leurs interactions entre elles**.

Les deux propriétés de base de la programmation fonctionnelle :

- **Fonctions de première classe** (*First class citizenship*) : Les fonctions sont des objets qui se manipulent comme des données ou des variables normales, par exemple, elles peuvent être affectée à une variable (`f = sin`) ou servir d'argument à d'autres fonctions (`g(10, f)`, ...).
  Avantages : Flexibilité d'utilisation, compositionalité.
- **Pureté fonctionnelle** (*Purity*) : Le résultat de la fonction ne dépend que de ses arguments d'entrée (déterminisme) et toutes ses opérations internes n'ont pas d'effet externes (absence d'effets de bords).
  Avantages : Cloisonnement, localisation, stabilité, déterminisme.

On parle de **programmation fonctionnelle** lorsqu'**au moins une de ces propriétés** est utilisée. Et, de **langage fonctionnel**, lorsque les fonctionnalités de base du langage en question favorisent l'usage de ces **deux propriétés à la fois**.

Notes

Programmation fonctionnelle en Python

Python fournit un ensemble de facilités pour la programmation fonctionnelle mais pas toutes, on peut donc faire de la programmation fonctionnelle en Python. Mais, Python ne peut être considéré comme un langage fonctionnel.

Ce qu'il est possible de faire (ou pas) en Python :

- **Fonctions de première classe** : Les fonctions Python sont des fonctions de premières classes et supportent tous les traitements habituels des fonctions en programmation fonctionnelle (voir la suite).
- **Fonctions pures** : Le langage Python n'assure pas de lui-même la pureté des fonctions, c'est au programmeur de s'assurer de cela. Par exemple, voici une fonction pure et une fonction impure :

```
>>> def foo(i):
...     return i+1
>>> foo(1)
2
>>> foo(1)
2
```

```
>>> j = 4
>>> def bar(i):
...     global j
...     j = i + j
...     return j
>>> bar(1)
5
>>> bar(1)
6
```


Notes

Fonctions de première classe (Python)

- **Affecter à une variable** :

```
f = sin
```

- **Argument d'une fonction** :

```
g(10, f)
```

- **Définition à la demande (fonction anonyme)** :

```
f = lambda x : x + 1
```

- **Code de retour d'une (autre) fonction** :

```
def h(x):
    return lambda x: g(x, f)
```

- **Stockage dans un conteneur** :

```
foo = [ sin, cos, lambda x: x**2 ]
```

- **Définir une fonction dans une fonction** :

```
def foo(x):
    def bar(y):
        return g(y, f)
    return bar(x)
```


Notes

Compréhension de listes (1/2)

La **compréhension de listes** est une syntaxe compacte pour créer ou appliquer des opérations sur des listes. Cette syntaxe permet notamment d'implémenter les fonctions **`map()` et `filter()`** qui sont très utilisées en programmation fonctionnelle.

Les standards de codage de Python recommandent d'ailleurs de passer par les compréhension de liste plutôt que d'utiliser `map()` et `filter()`.

Paradigme Impératif

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Paradigme Fonctionnel

```
squares = list(map(lambda x: x**2, range(10)))
```

Paradigme Fonctionnel (avec compréhension de listes)

```
squares = [x**2 for x in range(10)]
```


Notes

Compréhension de listes (2/2)

Les fonctions `map()` et `filter()` et comment les exprimer en tant que compréhension de listes :

- **Fonction "`map(myfunc, mylist)`"**
  Cette fonction applique la fonction `myfunc` à chacun des éléments de la liste `mylist` et retourne le résultat. L'équivalent en compréhension de listes est :

```
[myfunc(x) for x in mylist]
```

- **Fonction "`filter(myfunc, mylist)`"**
  Cette fonction filtre la liste `mylist` grâce à la fonction booléenne `myfunc`. L'équivalent en compréhension de listes est :

```
[x for x in mylist if myfunc(x)]
```

Imbrication des compréhension de listes

On a parfois besoin d'itérer sur plusieurs ensembles et créer une unique liste en retour.

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```


Notes

Gestionnaire de contexte

La structure de contrôle "`with...as..., as...:`" permet de déclarer un **gestionnaire de contexte** qui essayera de garantir la **pureté de l'environnement** (au sens fonctionnel) en fermant les objets à portée globale à la sortie du bloc.

```
with VAR as EXPR:
    BLOCK
```

```
VAR = EXPR
VAR.__enter__()
try:
    BLOCK
finally:
    VAR.__exit__()
```

Par exemple, l'ouverture d'un fichier avec `open()`, nécessite habituellement que l'on ferme le fichier après utilisation. Mais, ici, le `with` permet de garantir que le descripteur de fichier sera automatiquement refermé à la fin du bloc (qu'il y ai eut une exception ou pas). Le code complet est le suivant :

```
with open("myfile.txt") as f:
    for line in f:
        print(line)
```


Notes

Générateurs

Un **générateur** est une fonction particulière qui conserve son état en mémoire et qui, à chaque appel, reprend là où elle s'était interrompue à son dernier appel. En Python, il suffit d'utiliser le mot clef `yield` à la place de `return` pour transformer la fonction en générateur.

```
def squared():
    i = 0
    while True:
        i = i + 1
        yield i * i
```

```
>>> mygen = squared()
>>> mygen.next()
1
>>> mygen.next()
4
>>> mygen.next()
9
```

L'utilisation de `yield` transforme immédiatement la fonction en générateur. On peut construire des générateurs qui vont parcourir des ensembles infinis de manière paresseuse (sans jamais évaluer l'ensemble en une fois, il calculera seulement l'élément suivant).
Un générateur est aussi très utile pour construire des filtres, par exemple si l'on veut construire un filtre qui remplace les tabulations par 8 espaces sur un fichier :

```
def filter(stream):
    for line in stream:
        yield line.replace('\t', ' ' * 8)

with open(filename, 'r') as f:
    for line in filter(f):
        print(f)
```


Notes

Décorateurs

Les **décorateurs** permettent d'**ajouter un pré/post-traitement à des fonctions sans modifier leur code** en encapsulant la fonction dans une autre fonction (le décorateur). Pour cela, Python utilise les propriétés particulières des fonctions de première classe. Il s'agit donc d'un décorateur au sens "*programmation fonctionnelle*" et non du "*design-pattern*" issue de la *programmation objet* (même s'ils ont beaucoup en commun).

Les fonctions *builtins* `classmethod()` et `staticmethod()` sont des exemples classiques de décorateurs. Elles servent à marquer une méthode comme étant, respectivement, une méthode de classe (dont le premier argument est la classe de l'objet auquel appartient la méthode, et non `self`)) et une méthode statique (dont on omet l'argument `self`). Ainsi, les deux écritures suivantes sont équivalentes :

```
def f(...):
    ...
f = staticmethod(f)
```

```
@staticmethod
def f(...):
    ...
```


Notes

Déclarer un décorateur

Un décorateur "`@timer`" qui compte le temps d'exécution d'une fonction :

```
import time

def timer(func):
    """A 'timer' decorator function to evaluate execution time"""
    def timed(*args, **kw):
        start_time = time.time()
        result = func(*args, **kw)
        end_time = time.time()

        print('%r (%r, %r) %2.2f s' % \
              (func.__name__, args, kw, end_time-start_time))
        return result

    return timed

# Usage of the timer on a function foo()
@timer
def foo():
    time.sleep(1)
    print 'foo'
```


Notes

Plan

Notes

Et après ?

Il reste encore beaucoup à explorer par vous-même ! Pour bien maîtriser un langage de programmation, il faut le pratiquer. Et, surtout dans un contexte particulier. Quelques suggestions :

- **Développement Web** : Django, Flask, Pyramid, ...
- **Administration système** : Fabric, Salt, ...
- **Développement** : Scons, Buildbots, Tests logiciels, ...
- **Sécurité Informatique** : Fusil, Miasm2, Scappy, ...
- **Calcul Scientifique** : SciPy, Sage, ...
- **Bio-informatique** : Biopython, Scikit-bio, ...
- **...**

Notes

Notes

Notes