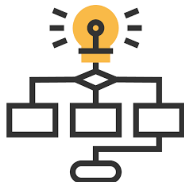
Python pour les scientifiques
Partie IV – Instructions composées

Cyril Desjouy

June, 2016
Updated : March 26, 2019



Qu'est ce qu'une instruction composée ?

- Un bloc de code contenant des instructions
- Typiquement une *instruction algorithmique* ou une *fonction/classe*

Structure des instructions composées sous Python :

- Caractère `":"` à la fin de la déclaration (1ère ligne) d'une instruction composée
- Les instructions suivantes sont indentées du **même nombre de caractères espace "␣"** et sont considérées comme faisant partie du **même bloc de code**
- Python utilise l'**indentation** comme méthode pour grouper les instructions

Avantages de l'indentation :

- Pas d'instruction `end` : Pas de temps perdu avec le *debug* dû aux `end` manquants !
- Force les bonnes pratiques : Lisibilité du code !

L'indentation sous Python

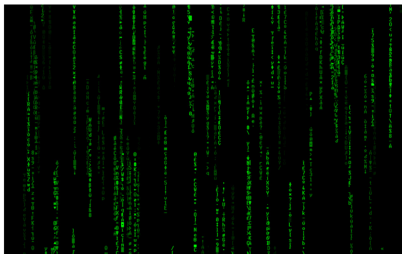
- Nécessaire pour les instructions composées (`if`, `for`, `while`, `with`, `try`, `def`, `class`)
- Niveau d'indentation : nombre de caractères "␣" au début d'une ligne logique
- Norme proposée par la PEP-8 : 4 caractères "␣" pour 1 niveau d'indentation
- Mélange de différents types d'indentation dans un même groupe : IndentationError

Illustration : Instructions composées imbriquées

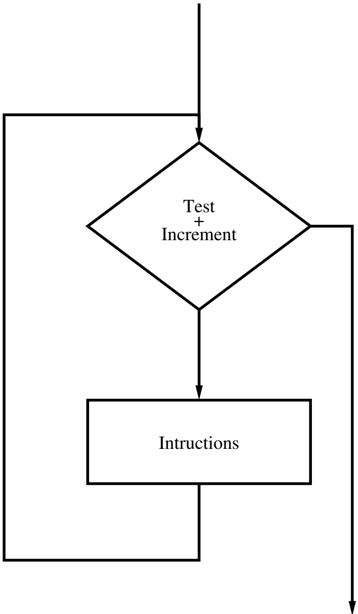
```
def f(x):              # Instruction composée L0
    y = x**2           #      | Début bloc instructions indentées L1
    def g(z):          #      | Instruction composée L1
        z = z**3       #      |      | Début bloc instruction indentée L2
        return z       #      |      | Fin bloc instruction indentée L2
    return y + g(y)    #      | Fin bloc instructions indentées L1
```

Gardez en tête : la notion d'indentation est l'une des notions les plus importantes sous Python. C'est un élément clé de sa syntaxe !

Instructions algorithmiques



Screenshot by Gamaliel Espinoza Macedo


Test
+
Increment
Intructions

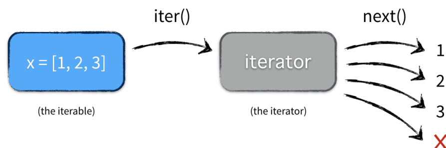
Syntaxe : for ... in ...

```
for p in range(10):  # Définition de l'instruction composée
    p = p**2         # Bloc d'instructions répétées
    print(p)         # ...
print('Fin')         # Instruction non incluse dans la boucle

0                    # Exécution de l'instruction composée for
1                    # ...
4                    # ...
...                  # ...
64                   # ...
81                   # Fin de l'exécution du for
Fin                  # Exécution des instructions suivantes
```

- Exécution d'un bloc d'instructions pour un nombre fixé d'itérations
- Utilisation du mot clé `in` proche du langage humain
- Instructions `break` (sortie immédiate) et `continue` (l'itération suivante) acceptées
- Peut itérer sur différents types d'objets :
  - Des itérables (`list`, `tuple`, `dictionary`, `array`, `string`, `range`, fichier, ...)
  - Des itérateurs
  - Toute fonction retournant un objet itérable ou un itérateur

Note sur les objets itérables et les itérateurs



Extrait de www.nvie.com

- ***Itérer* : Action de lire les éléments d'un *objet itérable* un par un**
- ***Itérable* : hérite de la méthode `__iter__()`**
  - Objet qui peut retourner un itérateur grâce à la méthode `iter`
  - Objet sur lequel on peut utiliser **`for`** ... **`in`** ...
  - L'itérable est l'objet qui contient les données
- ***Itérateur* : hérite des méthodes `__iter__()` et `__next()__`**
  - Objet qui génère la prochaine valeur d'un itérable avec la méthode `next`
  - L'itérateur est itérable : il peut retourner un itérateur grâce à la méthode `iter`
  - Une fois l'itérateur vide, l'exception **StopIteration** est levée
  - L'itérateur n'est pas l'objet qui contient pas les données

Illustration : ce qui se passe dans une boucle for

```
In  : iterable = [1, 2, 4]
In  : iterator = iter(iterable)    # Équivalent à iterable.__iter__()
In  : next(iterator)               # Équivalent à iterator.__next__()
Out : 1
In  : next(iterator)
Out : 2
In  : next(iterator)
Out : 4
In  : next(iterator)               # Itérateur vide : Exception levée
-----------------------------------
StopIteration Traceback (most recent call last)
```

```
In  : iterable = [1, 2, 4]
In  : type(iterable)
Out : list
In  : for i in iterable :
In  : ...  print(i)
Out : 1
      2
      4
```

```
In  : iterator = iter([1, 2, 4])
In  : type(iterator)
Out : list_iterator
In  : for i in iterator :
In  : ...  print(i)
Out : 1
      2
      4
```

Quelques exemples d'objets itérables

La liste

```
lst = [1, "a"]
for p in lst:
  print(p)       # Return 1, "a"
```

Le tuple

```
tup = (1, "a")
for p in tup:
  print(p)       # Return 1, "a"
```

Le dictionnaire

```
dict = {"one":  1, "two":  2}
for p in dict:
  print(p)
  # Return one, two
```

La chaîne de caractères

```
string = "abc"
for p in string:
  print(p)
  # Return a, b, c
```

Le fichier

```
file = open("myfile")
for line in file:
  print(line)
  # Return each line of the file
```

Le set

```
myset = {1, 1, 2, 4}
for p in myset:
  print(p)
  # Return 1, 2, 4
```

Comment garder trace de l'indice ?

La fonction enumerate

```
mylist = ["Python", "Is", "Amazing"]

for i in range(len(mylist)):
        print(i, mylist[i])

# L'exécution de ce bloc de code donne :

0 Python
1 Is
2 Amazing
```

Comment garder trace de l'indice ?

La fonction enumerate

```
mylist = ["Python", "Is", "Amazing"]

for i in range(len(mylist)):
        print(i, mylist[i])

# L'exécution de ce bloc de code donne :

0 Python
1 Is
2 Amazing

# Ecriture directe avec la fonction enumerate qui retourne un tuple :

for index, item in enumerate(mylist):
        print(index, item)
0 Python
1 Is
2 Amazing
```

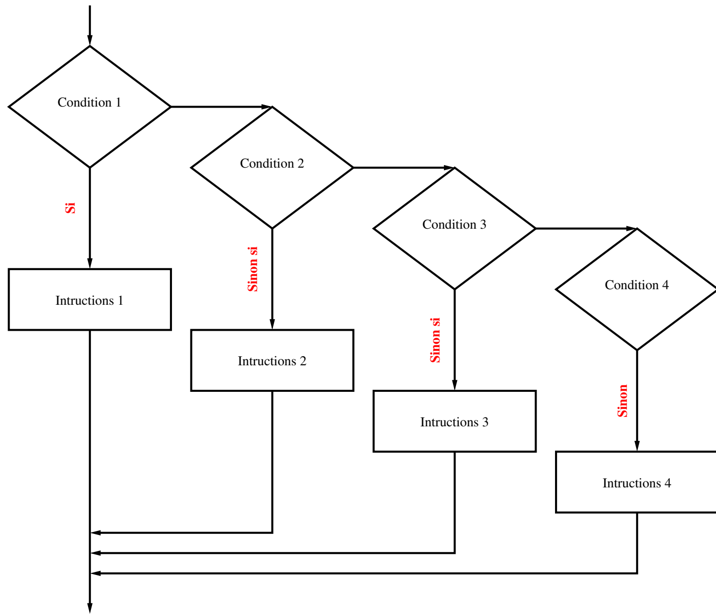
La fonction zip

```
import string

letters = string.ascii_lowercase
indexes = range(len(letters))
rletters = reversed(letters)

for i, j, k in zip(indexes, letters, rletters):
    print(i, j, k)

    0 a z
    1 b y
    2 c x
    3 d w
    4 e v
    5 f u
    6 g t
    (...)
    22 w d
    23 x c
    24 y b
    25 z a
```


Condition 1
Condition 2
Condition 3
Condition 4
Si
Sinon si
Sinon si
Sinon
Intructions 1
Intructions 2
Intructions 3
Intructions 4

Syntaxe : if ... elif ... else ...

```
if condition is True:                #
    print("Verified")                #
elif condition is False:             # elif optionnel
    print("Not Verified")            #
else:                                # else optionnel - n'accepte pas ...
    print("Other possibility ?")     # ...de condition
```

Opérateurs relationnels : Compare les valeurs de chaque côté		
Égalité/Différence	`==` et `!=`	–
Supériorité/Infériorité	`>` / `<` *ou* `>=` / `<=`	–
Opérateurs logiques classiques		
.AND. logique	`and`	–
.OR. logique	`or`	–
.NOT. logique	`not`	Inverse l'opérateur accolé
Opérateurs d'appartenance : Test d'appartenance dans une séquence		
Inclusion	*var* `in` *seq*	`True if` *var* `in` *seq*, `False` otherwise
Exclusion	*var* `not in` *seq*	`True if` *var* `not in` *seq*, `False` otherwise
Opérateurs identité : Compare l'adresse mémoire d'objets		
Égalité	*a* `is` *b*	`True if` `id`$(a) ==$ `id`(b), `False` otherwise
Différence	*a* `is not` *b*	`True if` `id`(a) != `id`(b), `False` otherwise

En mathématiques, la formulation suivante est la méthode commune pour décrire des listes :

- $B = \{x^2 : x \in \{0 \dots 9\}\}$
- $T = \{x : x \in B \text{ et } x \text{ impair}\}$

Un concept pour construire des listes naturellement

```
In  : # Liste des carrés de [0, 9]
In  : B = [x**2 for x in range(10)]
In  : print(B)
Out : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In  : # Filtrage de B :
In  : print([x for x in B if x % 2 == 0])
Out : [0, 4, 16, 36, 64]

In  : # Un exemple avec un objet de type str :
In  : print(["a" * i for i in range(5)])
Out : ["", "a", "aa", "aaa", "aaaa"]

In  : # Un exemple de compréhensions de liste imbriquées :
In  : print([[i*i for i in range(x)] for x in range(5)])
Out : [[], [0], [0, 1], [0, 1, 4], [0, 1, 4, 9]]
```

Les compréhensions de dictionnaires et de sets

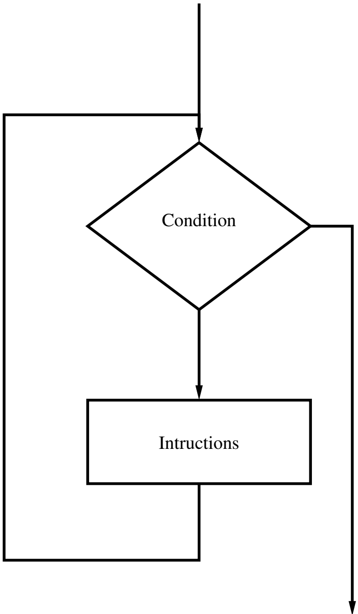
```
In  : {x: x**2 for x in range(3)}
Out : {0: 0, 1: 1, 2: 4}          # type == dict
In  : {x**2 for x in range(3)}
Out : {0, 1, 4}                   # type == set
```

- Compréhension de dictionnaire : `{key: value for (key, value) in iterable}`
- Compréhension de set : `{expression for element in iterable}`
- Expression génératrice : `(expression for element in iterable)`

Les expressions génératrices

```
In  : gen = (x**2 for x in range(3))
In  : type(gen)                   # Retourne "generator"
In  : for i in gen:
      ...  print(i)               # Retourne 0, puis 1, puis 4
```

- Expressions génératrices : version `generator` des compréhensions de listes
- Un objet de type `generator` génère des valeurs à la demande, une par une !
- Le type `generator` est en fait un sous type du type `iterator` (`generator` ≡ `yield`)
- Moyen efficace en terme d'empreinte mémoire de générer des séquences volumineuses
- Nous verrons également dans la suite les ***fonctions génératrices*** …


Condition
Intructions

Syntaxe : while ...

```
n = 0             # Initialisation de la variable utilisée dans la suite
while n != 10:    # Définition de l'instruction composée
    n += 1        # Bloc d'instructions répétées
    print(n)      # ...
print("Fin")      # Instruction non incluse dans la boucle
```

- Exécution théorique infinie (`while True: ...`)
- Définition d'une condition de sortie pour arrêter l'exécution
- Instructions `break` (sortie immédiate) et `continue` (l'itération suivante) acceptées
- **Note :** Opérateur `"+="` : assignation et addition de manière compacte :
  - `n += 1` ⟶ `n = n+1`
  - `n -= 1` ⟶ `n = n-1`
  - `n *= 2` ⟶ `n = n*2`
  - `n **= 2` ⟶ `n = n**2`
  - `n /= 2` ⟶ `n = n/2`
  - `n //= 2` ⟶ `n = n//2`
  - `n %= 2` ⟶ `n = n%2`

Syntaxe : `try` ... `except` ... `else` ... `finally` ...

```
try:                           # Essaye le prochain bloc indenté
  n = float(input( "Enter a number :  "))
  a = 1/n
except ValueError:             # Si ValueError est levée, exécute :
  print("Not a Number !  Fix value to 1")
  a = 1
except ZeroDivisionError:      # Si ZeroDivisionError est levée, exécute :
  print("Div.  by 0 !  Fix value to 1!")
  a = 1
else:                          # Si aucune exception n'est levée, exécute :
  print("No error :  good job !")
finally:                       # Quoi qu'il arrive, exécute :
  print("Square of {} is {}".format(a, a**2))
```

- Exceptions levées par des erreurs : différentes exceptions pour différentes erreurs
- `try`/`except` spécifie la gestion des exceptions : `Essaye ceci et attends toi à cela`
- Les instructions `else` et `finally` sont optionnelles
- **Attention :** capturer une exception et ne pas la lever peut rendre le debug difficile !

Syntaxe : with ... as ...

```
with open("myfile.txt") as f:
    data = f.read()
    do something with data
```

- L'instruction **with** gère les ressources
  - Une fonction est appelée en entrant dans le bloc de code
  - Une autre fonction est appelée en sortant du bloc de code
- Permet de ne pas oublier de supprimer les ressources et gère également des cas plus compliqués

Dans le cas de la gestion de fichiers, équivalent à :

```
f = open("myfile.txt")
data = f.read()
do something with data
...
...
f.close()                # Souvent oublié !
```

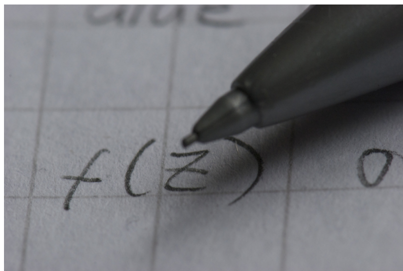
```
for i in sequence :
    instructions
```

```
while condition:
    instructions
```

```
if condition1:
    instructions
elif condition2:    # Optionnel
    instructions
else:               # Optionnel
    instructions
```

```
try:
    instructions
except Exception:
    instructions
else:                # Optionnel
    instructions
finally:             # Optionnel
    instructions
```

Fonctions et Classes



Photography by Vestman

Syntaxe : `def ... return ...`

```
def mult(x, y=2):   # x :  positional argument, y :  keyword argument
    return x*y
```

```
In  : mult(3)
Out : 6
```

```
In  : mult(3, 4)
Out : 12
```

- L'instruction `def` permet de définir une fonction
- Une fonction est un objet qui retourne un objet
- Par défaut, une fonction retourne `None` s'il n'y a pas d'instruction `return`
- Une fonction peut être utilisée comme tout autre objet. Elle peut être :
  - un argument ou un `return` d'une autre fonction
  - référencée par une variable
  - un élément d'une structure (`tuple`, `list`, `dict`, ...)
- Une définition de fonction accepte des paramètres :
  - obligatoires, appelés ***positional arguments***
  - optionnels, appelés ***keyword arguments*** spécifiant des valeurs par défaut[1]

[1] Ces valeurs sont évaluées au moment où la fonction est définie, pas lorsqu'elle est appelée

Syntaxe : `class ... def __init__(self) ...`

```python
class Guitar:                         # La classe et ...
    """ My Great Guitar Class """     # ...  sa DocString !

    def __init__(self):               # Le constructeur de classe
        self.brand = "Schecter"       # Un attribut d'instance ...
        self.color = "Cherry"         # ...un second...
        self.owner = "Me"             # ...et un troisième

    def description(self):            # Une méthode et ...
        """ My Great method """       # ...  sa DocString !

        print("Une {} couleur {}".format(self.brand, self.color)
```

- Une `class`, c'est comme une usine d'objets !
- Instances d'une classe : construites avec la méthode spéciale `__init__`
- Méthodes : définies comme des fonctions mais dans la classe et avec comme premier argument, l'argument spécial `self`
- Attributs : définis en utilisant l'objet `self`
- Les classes sont normalement nommées en utilisant la convention *CapWords*

Syntaxe : class ... def __init__(self) ...

```
class Guitar:                              # La classe et ...
    """ My Great Guitar Class """          # ...  sa DocString !

    def __init__(self):                    # Le constructeur de classe
        self.brand = "Schecter"            # Un attribut d'instance ...
        self.color = "Cherry"              # ...un second...
        self.owner = "Me"                  # ...et un troisième

    def description(self):                 # Une méthode et ...
        """ My Great method """            # ...  sa DocString !

        print("Une {} couleur {}".format(self.brand, self.color)
```

Utilisation des class

```
» My1stGuitar = Guitar()        # Crée un nouvel objet Guitar ...
» My1stGuitar.brand             # ...appellé instance de la classe Guitar
  Schecter                      # ...et qui hérite de tous ses attributs ...
» My1stGuitar.description()     # ...et de ses méthodes !
  Une Schecter couleur Cherry
```