



Formation Python

Partie 1 : Syntaxe du langage Python



Introduction

Introduction

Aperçu historique?



1989 - 1991



Guido van Rossum



Introduction

Caractéristiques du Python

01

Multiplateforme

02

Gratuit

03

Haut niveau

04

Interprété

05

Orienté-Objet

06

Data Science



Variables

Variables

Définition

Une **variable** est une **zone de la mémoire** de l'ordinateur dans laquelle une valeur est stockée.

La variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

```
1 | >>> x = 2
2 | >>> x
3 | 2
```



Variables

Définition

$X = 2$

- Signifie qu'on attribue la valeur située à droite de l'opérateur = (ici, 2) à la variable située à gauche (ici, x).



Variables

Arrière plan

- Python a « deviné » que la variable était un entier. On dit que Python est un langage au typage dynamique.
- Python a alloué (réservé) l'espace en mémoire pour y accueillir un entier.
- Chaque type de variable prend plus ou moins d'espace en mémoire.
- Python a aussi fait en sorte qu'on puisse retrouver la variable sous le nom x.
- Enfin, Python a assigné la valeur 2 à la variable x.

Variables

Définition

$$X = Y - 3$$



Variables

Définition

$$X = Y - 3$$

- L'opération $y - 3$ est d'abord évaluée et ensuite le résultat de cette opération est affecté à la variable x .



Variables

Types de variables

- Les entiers (integer ou int)
- Les nombres décimaux que nous appellerons floats.
- Les chaînes de caractères (string ou str)



Variables

Types de variables

```
1 >>> y = 3.14
2 >>> y
3 3.14
4 >>> a = "bonjour"
5 >>> a
6 'bonjour'
7 >>> b = 'salut'
8 >>> b
9 'salut'
10 >>> c = """girafe"""
11 >>> c
12 'girafe'
13 >>> d = '''lion'''
14 >>> d
15 'lion'
```



Variables

Nommage

- Le nom de variable ne doit pas débiter par un chiffre et il n'est pas recommandé de le faire débiter par le caractère _ (sauf cas très particuliers).
- Il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par exemple : print, range, for, from, etc.).
- Python est sensible à la casse, ce qui signifie que les variables Test, test ou TEST sont différentes.



Variables

Nommage

- Le nom des variables en Python peut être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de nombres (0 à 9) ou du caractère souligné (_).
- Le nom d'une variable ne doit pas contenir une espace.
- Les chaînes de caractères (string ou str).



Variables

Ecriture scientifique

Ecrire des nombres très grands ou très petits avec des puissances de 10 en utilisant le symbole e.

```
1 >>> 1e6  
2 1000000.0  
3 >>> 3.12e-3  
4 0.00312
```



Variables

Ecriture scientifique

```
1 >>> 1e6
2 1000000.0
3 >>> 3.12e-3
4 0.00312
```

- Signifie 1×10^6 à la puissance ou 3.12×10^{-3} à la puissance -3 respectivement.
- Même si on ne met que des entiers à gauche et à droite du symbole e (comme dans 1e6), Python génère systématiquement un float.

Variables

Ecriture scientifique

```
1 >>> avogadro_number = 6.022_140_76e23
2 >>> print(avogadro_number)
3 6.02214076e+23
4 >>> humans_on_earth = 7_807_568_245
5 >>> print(humans_on_earth)
6 7807568245
```

```
1 >>> print(7_80_7568_24_5)
2 7807568245
```



Variables

Opérations sur les types numériques

```
1 >>> x = 45
2 >>> x + 2
3 47
4 >>> x - 2
5 43
6 >>> x * 3
7 135
8 >>> y = 2.5
9 >>> x - y
10 42.5
11 >>> (x * 10) + y
12 452.5
```



Variables

Opérations sur les types numériques

```
1 | >>> 3 / 4
2 | 0.75
3 | >>> 2.5 / 2
4 | 1.25
```



Variables

Opérations sur les types numériques

```
1 | >>> 2**3  
2 | 8  
3 | >>> 2**4  
4 | 16
```



Variables

Opérations sur les types numériques

```
1 >>> 5 // 4
2 1
3 >>> 5 % 4
4 1
5 >>> 8 // 4
6 2
7 >>> 8 % 4
8 0
```



Variables

Opérations sur les types numériques

```
1 >>> i = 0
2 >>> i = i + 1
3 >>> i
4 1
5 >>> i += 1
6 >>> i
7 2
8 >>> i += 2
9 >>> i
10 4
```



Variables

Opérations sur les chaînes de caractères

```
1 >>> chaine = "Salut"  
2 >>> chaine  
3 'Salut'  
4 >>> chaine + " Python"  
5 'Salut Python'  
6 >>> chaine * 3  
7 'SalutSalutSalut'
```



Variables

Opérations illicites

```
1 >>> "toto" * 1.3
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: can't multiply sequence by non-int of type 'float'
5 >>> "toto" + 2
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: can only concatenate str (not "int") to str
```



Variables

La fonction type()

Argument

```
1 >>> x = 2
2 >>> type(x)
3 <class 'int'>
4 >>> y = 2.0
5 >>> type(y)
6 <class 'float'>
7 >>> z = '2'
8 >>> type(z)
9 <class 'str'>
```



Variables

Conversion de types

En programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
1 | >>> i = 3
2 | >>> str(i)
3 | '3'
4 | >>> i = '456'
5 | >>> int(i)
```



Variables

Conversion de types

En programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
6 | 456
7 | >>> float(i)
8 | 456.0
9 | >>> i = '3.1416'
10 | >>> float(i)
11 | 3.1416
```



Variables

Minimum & Maximum

```
1 >>> min(1, -2, 4)
2 -2
3 >>> pi = 3.14
4 >>> e = 2.71
5 >>> max(e, pi)
6 3.14
7 >>> max(1, 2.4, -6)
8 2.4
```





Affichage

Affichage

La fonction print ()

```
1 >>> print("Hello world!")  
2 Hello world!  
3 >>> print("Hello world!", end="")  
4 Hello world!>>>
```



Affichage

La fonction print ()

```
1 >>> print("Hello") ; print("Joe")
2 Hello
3 Joe
4 >>> print("Hello", end="") ; print("Joe")
5 HelloJoe
6 >>> print("Hello", end=" ") ; print("Joe")
7 Hello Joe
```



Affichage

La fonction print ()

```
1 | >>> var = 3
2 | >>> print(var)
3 | 3
```



Affichage

La fonction print ()

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print(nom, "a", x, "ans")
4 John a 32 ans
```



Affichage

La fonction print ()

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print(nom, "a", x, "ans", sep="")
4 Johna32ans
5 >>> print(nom, "a", x, "ans", sep="-")
6 John-a-32-ans
```

```
1 >>> ani1 = "chat"
2 >>> ani2 = "souris"
3 >>> print(ani1, ani2)
4 chat souris
5 >>> print(ani1 + ani2)
6 chatsouris
7 >>> print(ani1, ani2, sep="")
8 chatsouris
```



Affichage

Ecriture formatée

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un certain format, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les floats.

L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés en « belles colonnes »



Affichage

Ecriture formatée

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un certain format, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les floats.

L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés en « belles colonnes »



Affichage

Ecriture formatée

L'équivalent en f-string est tout simplement la même chaîne de caractères précédée du caractère f sans espace entre les deux :

f"Ceci est une chaîne de caractères "

Ce caractère f avant les guillemets va indiquer à Python qu'il s'agit d'une f-string permettant de mettre en place le mécanisme de l'écriture formatée, contrairement à une string normale.



Affichage

Ecriture formatée

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print(f"{nom} a {x} ans")
4 John a 32 ans
```

```
1 >>> print("{nom} a {x} ans")
2 {nom} a {x} ans
```



Affichage

Ecriture formatée

```
1 >>> var = "to"
2 >>> print(f"{var} et {var} font {var}{var}")
3 to et to font toto
4 >>>
```

```
1 >>> print(f"J'affiche l'entier {10} et le float {3.14}")
2 J'affiche l'entier 10 et le float 3.14
3 >>> print(f"J'affiche la chaine {'Python'}")
4 J'affiche la chaine Python
```



Affichage

Ecriture formatée

```
1 >>> prop_GC = (4500 + 2575) / 14800
2 >>> print("La proportion de GC est", prop_GC)
3 La proportion de GC est 0.4780405405405405
```

```
1 >>> print(f"La proportion de GC est {prop_GC:.2f}")
2 La proportion de GC est 0.48
3 >>> print(f"La proportion de GC est {prop_GC:.3f}")
4 La proportion de GC est 0.478
```



Affichage

Ecriture formatée

```
1 >>> nb_G = 4500
2 >>> print(f"Ce génome contient {nb_G:d} guanines")
3 Ce génome contient 4500 guanines
```

```
1 >>> nb_G = 4500
2 >>> nb_C = 2575
3 >>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, soit une prop de GC de {prop_GC:.2f}")
4 Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
5 >>> perc_GC = prop_GC * 100
6 >>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, soit un %GC de {perc_GC:.2f} %")
7 Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```



Affichage

Ecriture formatée

```
1 >>> print(10) ; print(1000)
2 10
3 1000
4 >>> print(f"{10:>6d}") ; print(f"{1000:>6d}")
5      10
6      1000
7 >>> print(f"{10:<6d}") ; print(f"{1000:<6d}")
8 10
9 1000
10 >>> print(f"{10:^6d}") ; print(f"{1000:^6d}")
11      10
12      1000
13 >>> print(f"{10:*^6d}") ; print(f"{1000:*^6d}")
14 **10**
15 *1000*
16 >>> print(f"{10:0>6d}") ; print(f"{1000:0>6d}")
17 000010
18 001000
```



Affichage

Ecriture formatée

```
1 >>> print("atom HN") ; print("atom HDE1")
2 atom HN
3 atom HDE1
4 >>> print(f"atom {'HN':>4s}") ; print(f"atom {'HDE1':>4s}")
5 atom    HN
6 atom    HDE1
```

```
1 >>> print(f"{perc_GC:7.3f}")
2 47.804
3 >>> print(f"{perc_GC:10.3f}")
4 47.804
```



Affichage

Ecriture formatée

```
1 >>> print(f"Accolades littérales {{{}} ou {{ ou }}" et pour le formatage {10}")
2 Accolades littérales {} ou { ou } et pour le formatage 10
```

```
1 >>> print(f"accolades sans variable {}")
2 File "<stdin>", line 1
3 SyntaxError: f-string: empty expression not allowed
```



Affichage

Ecriture formatée

```
1 >>> f"{perc_GC:10.3f}"  
2 '      47.804'  
3 >>> type(f"{perc_GC:10.3f}")  
4 <class 'str'>
```



Affichage

Ecriture formatée

```
1 >>> print(f"Le résultat de 5 * 5 vaut {5 * 5}")
2 Le résultat de 5 * 5 vaut 25
3 >>> print(f"Résultat d'une opération avec des floats : {(4.1 * 6.7)}")
4 Résultat d'une opération avec des floats : 27.47
5 >>> print(f"Le minimum est {min(1, -2, 4)}")
6 Le minimum est -2
7 >>> entier = 2
8 >>> print(f"Le type de {entier} est {type(entier)}")
9 Le type de 2 est <class 'int'>
```



Affichage

Ecriture formatée

```
1 >>> print(f"{1_000_000_000:e}")  
2 1.0000000e+09  
3 >>> print(f"{0.000_000_001:e}")  
4 1.0000000e-09
```



Affichage

Ecriture formatée

```
1 | >>> avogadro_number = 6.022_140_76e23
2 | >>> print(f"{avogadro_number:.0e}")
3 | 6e+23
4 | >>> print(f"{avogadro_number:.3e}")
5 | 6.022e+23
6 | >>> print(f"{avogadro_number:.6e}")
7 | 6.022141e+23
```



Affichage

Ecriture formatée

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print("%s a %d ans" % (nom, x))
4 John a 32 ans
5 >>> nb_G = 4500
6 >>> nb_C = 2575
7 >>> prop_GC = (nb_G + nb_C)/14800
8 >>> print("On a %d G et %d C -> prop GC = %.2f" % (nb_G, nb_C, prop_GC))
9 On a 4500 G et 2575 C -> prop GC = 0.48
```

1. Désigner l'endroit où sera placée la variable dans la chaîne de caractères.
2. Préciser le type de variable à formater, d pour un entier (i fonctionne également) ou f pour un float.
3. Éventuellement pour indiquer le format voulu. Ici .2 signifie une précision de deux décimales



Affichage

Ecriture formatée

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print("{} a {} ans".format(nom, x))
4 John a 32 ans
5 >>> nb_G = 4500
6 >>> nb_C = 2575
7 >>> prop_GC = (nb_G + nb_C)/14800
8 >>> print("On a {} G et {} C -> prop GC = {:.2f}".format(nb_G, nb_C, prop_GC))
9 On a 4500 G et 2575 C -> prop GC = 0.48
```

1. Dans la chaîne de caractères, les accolades vides {} précisent l'endroit où le contenu de la variable doit être inséré.
2. Juste après la chaîne de caractères, l'instruction `.format(nom, x)` fournit la liste des variables à insérer, d'abord la variable `nom` puis la variable `x`.
3. On peut éventuellement préciser le formatage en mettant un caractère deux-points : puis par exemple ici `.2f` qui
4. signifie 2 chiffres après la virgule.
5. La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est attachée par le point.





Listes

Listes

Définition

Une liste est une structure de données qui contient une série de valeurs.

Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

```
1 >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> tailles = [5, 2.5, 1.75, 0.15]
3 >>> mixte = ["girafe", 5, "souris", 0.15]
4 >>> animaux
5 ['girafe', 'tigre', 'singe', 'souris']
6 >>> tailles
7 [5, 2.5, 1.75, 0.15]
8 >>> mixte
9 ['girafe', 5, 'souris', 0.15]
```



Listes

Définition

Une liste est une structure de données qui contient une série de valeurs.

Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

```
1 >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> tailles = [5, 2.5, 1.75, 0.15]
3 >>> mixte = ["girafe", 5, "souris", 0.15]
4 >>> animaux
5 ['girafe', 'tigre', 'singe', 'souris']
6 >>> tailles
7 [5, 2.5, 1.75, 0.15]
8 >>> mixte
9 ['girafe', 5, 'souris', 0.15]
```



Listes

Définition

Une liste est une structure de données qui contient une série de valeurs.

Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

```
1 | liste : ["girafe", "tigre", "singe", "souris"]
2 | indice :      0          1          2          3
```

```
1 | >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 | >>> animaux[0]
3 | 'girafe'
4 | >>> animaux[1]
5 | 'tigre'
6 | >>> animaux[3]
7 | 'souris'
```

```
1 | >>> animaux[4]
2 | Traceback (innermost last):
3 |   File "<stdin>", line 1, in ?
4 | IndexError: list index out of range
```



Listes

Opérations sur les listes

```
1 >>> ani1 = ["girafe", "tigre"]
2 >>> ani2 = ["singe", "souris"]
3 >>> ani1 + ani2
4 ['girafe', 'tigre', 'singe', 'souris']
5 >>> ani1 * 3
6 ['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```



Listes

Opérations sur les listes

```
>>> liste=[]
>>> liste
[]
>>> liste=liste+['premier element']
>>> liste
['premier element']
>>> liste=liste+[4]
>>> LISTE
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    LISTE
NameError: name 'LISTE' is not defined
>>> liste
['premier element', 4]
>>> |
```



Listes

Opérations sur les listes

```
>>> liste.append('nouveau element')
>>> liste
['premier element', 4, 'nouveau element']
>>> liste.append('encore un nouveau element')
>>> liste
['premier element', 4, 'nouveau element', 'encore un nouveau element']
```



Listes

Indiçage négatif

```
1 liste      : ["girafe", "tigre", "singe", "souris"]
2 indice positif :      0      1      2      3
3 indice négatif :     -4     -3     -2     -1
```

ou encore :

```
1 liste      : ["A", "B", "C", "D", "E", "F"]
2 indice positif :    0    1    2    3    4    5
3 indice négatif :   -6   -5   -4   -3   -2   -1
```



Listes

Indiçage négatif

```
>>> liste[1]
4
>>> liste[-1]
'encore un nouveau element'
>>> liste[-4]
'premier element'
>>> |
```



Listes

Tranches

```
>>> liste=['Marie','Patricia','Jon','Florian']
>>> liste[0:4]
['Marie', 'Patricia', 'Jon', 'Florian']
>>> liste[0:2]
['Marie', 'Patricia']
>>> liste[1:]
['Patricia', 'Jon', 'Florian']
>>> liste[1:-1]
['Patricia', 'Jon']
>>> liste[:]
['Marie', 'Patricia', 'Jon', 'Florian']
>>> |
```



Listes

Tranches

```
>>> liste[0:3:2]
['Marie', 'Jon']
>>> liste[::2]
['Marie', 'Jon']
>>> liste[::3]
['Marie', 'Florian']
>>> liste[1:4:1]
['Patricia', 'Jon', 'Florian']
~::~
```

Liste[début:fin:pas]



Listes

Fonction len()

```
>>> len(liste)  
4  
>>> |
```



Listes

Fonction range()

```
>>> list(range(20))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]  
>>> list(range(1,15,2))  
[1, 3, 5, 7, 9, 11, 13]  
>>> list(range(2,14,3))  
[2, 5, 8, 11]  
>>> list(range(2,-10,-2))  
[2, 0, -2, -4, -6, -8]  
>>> |
```

```
>>> list(range(30,0))  
[]  
>>> list(range(30,0,-1))  
[30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17,  
 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  
>>> |
```



Listes

Fonction range()

```
>>> list(range(20))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]  
>>> list(range(1,15,2))  
[1, 3, 5, 7, 9, 11, 13]  
>>> list(range(2,14,3))  
[2, 5, 8, 11]  
>>> list(range(2,-10,-2))  
[2, 0, -2, -4, -6, -8]  
>>> |
```

```
>>> list(range(30,0))  
[]  
>>> list(range(30,0,-1))  
[30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17,  
 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  
>>> |
```



Listes

Listes de liste

```
>>> etudiant1=['Jon',20]
>>> etudiant2=['Marie',15]
>>> notes=[etudiant1,etudiant2]
>>> notes
[['Jon', 20], ['Marie', 15]]
>>> notes[1]
['Marie', 15]
>>> notes[0]
['Jon', 20]
>>> notes[0][0]
'Jon'
>>> notes[0][1]
20
>>> |
```



Listes

Minimum, Maximum, Somme d'une liste

```
>>> min(liste)
'Florian'
>>> max(liste)
'Patricia'
>>> notes=list(range(30,10,-1))
>>> notes
[30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
>>> min(notes)
11
>>> max(notes)
30
>>> min(10,11)
10
>>> max(12,15)
15
>>> min('Jon',7)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    min('Jon',7)
TypeError: '<' not supported between instances of 'int'
and 'str'
>>> |
```



Listes

Méthodes associées aux listes

```
>>> Une_liste=['a','b','c','d']
>>> #ajouter un élément à la fin d'une liste
>>> Une_liste.append('e')
>>> Une_liste
['a', 'b', 'c', 'd', 'e']
>>> Une_liste=Une_liste+['f']
>>> Une_liste
['a', 'b', 'c', 'd', 'e', 'f']
>>> #insérer un objet dans une liste à un indice déterminé
>>> Une_liste.insert(3,'i')
>>> Une_liste
['a', 'b', 'c', 'i', 'd', 'e', 'f']
>>> #supprimer un élément d'une liste à un indice déterminé
>>> del Une_liste[0]
>>> Une_liste
['b', 'c', 'i', 'd', 'e', 'f']
>>> #supprimer un élément d'une liste à partir de sa valeur
>>> Une_liste.remove('i')
>>> Une_liste
['b', 'c', 'd', 'e', 'f']
>>> #trier les éléments d'une liste du plus petit au plus grand
>>> Une_liste.sort()
>>> Une_liste
['b', 'c', 'd', 'e', 'f']
>>> a=[0,45,3,7,20,10,45]
>>> a.sort()
>>> a
[0, 3, 7, 10, 20, 45, 45]
>>> a.remove(45)
>>> a
[0, 3, 7, 10, 20, 45]
>>> a.remove(45)
>>> a
[0, 3, 7, 10, 20]
>>> #tri_inverse
>>> a.sort(reverse=True)
>>> a
[20, 10, 7, 3, 0]
```



Listes

Méthodes associées aux listes

```
>>> b=[0, 3, 7, 10, 20, 45]
>>> sorted(b)
Traceback (most recent call last):
  File "<pysHELL#30>", line 1, in <module>
    sorted(b)
NameError: name 'sorted' is not defined. Did you mean: 'sort'?
>>> sort(b)
[0, 3, 7, 10, 20, 45]
>>> sort(b,reverse=True)
[45, 20, 10, 7, 3, 0]
>>> b
[0, 3, 7, 10, 20, 45]
>>> #inverser une liste
>>> a=[13,11,12]
>>> a.reverse()
>>> a
[12, 11, 13]
>>> # compter le nombre d'éléments (passés en argument) dans une liste
>>> a.count(12)
1
>>> a.append([45,45])
>>> a
[12, 11, 13, [45, 45]]

>>> a=a+[45,45]
>>> a
[12, 11, 13, [45, 45], 45, 45]
>>> a.remove([45, 45])
>>> a
[12, 11, 13, 45, 45]
>>> a.count(45)
2
|
```



Listes

Test d'appartenance

```
>>> a
[12, 11, 13, 45, 45]
>>> 12 in a
True
>>> 'cours' in a
False
>>> |
```



Listes

Copie de liste

```
>>> x=[1,2,3]
>>> y=x
>>> x[0]=7
>>> x
[7, 2, 3]
>>> y
[7, 2, 3]
>>> |
```

```
>>> x=[1,2,3]
>>> y=x[:]
>>> x[0]=7
>>> x
[7, 2, 3]
>>> y
[1, 2, 3]
>>> x=[1,2,3]
>>> y=list(x)
>>> x
[1, 2, 3]
>>> y
[1, 2, 3]
>>> x[0]=7
>>> x
[7, 2, 3]
>>> y
[1, 2, 3]
>>> |
```

les deux astuces précédentes ne fonctionnent que pour les listes à une dimension!!



Listes

Copie de liste

```
>>> x=[[1,2],[3,4]]
>>> y=x[:]
>>> x[1][1]=45
>>> x
[[1, 2], [3, 45]]
>>> y
[[1, 2], [3, 45]]
>>> x=[[1,2],[3,4]]
>>> y=list(x)
>>> x[1][1]=78
>>> x
[[1, 2], [3, 78]]
>>> y
[[1, 2], [3, 78]]
```

```
>>> import copy
>>> x=[[1,2],[3,4]]
>>> y=copy.deepcopy(x)
>>> y
[[1, 2], [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> x[1][1]=9
>>> x
[[1, 2], [3, 9]]
>>> y
[[1, 2], [3, 4]]
>>> |
```



Listes

Copie de liste

```
>>> x=[[1,2],[3,4]]
>>> y=x[:]
>>> x[1][1]=45
>>> x
[[1, 2], [3, 45]]
>>> y
[[1, 2], [3, 45]]
>>> x=[[1,2],[3,4]]
>>> y=list(x)
>>> x[1][1]=78
>>> x
[[1, 2], [3, 78]]
>>> y
[[1, 2], [3, 78]]
```

```
>>> import copy
>>> x=[[1,2],[3,4]]
>>> y=copy.deepcopy(x)
>>> y
[[1, 2], [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> x[1][1]=9
>>> x
[[1, 2], [3, 9]]
>>> y
[[1, 2], [3, 4]]
>>> |
```

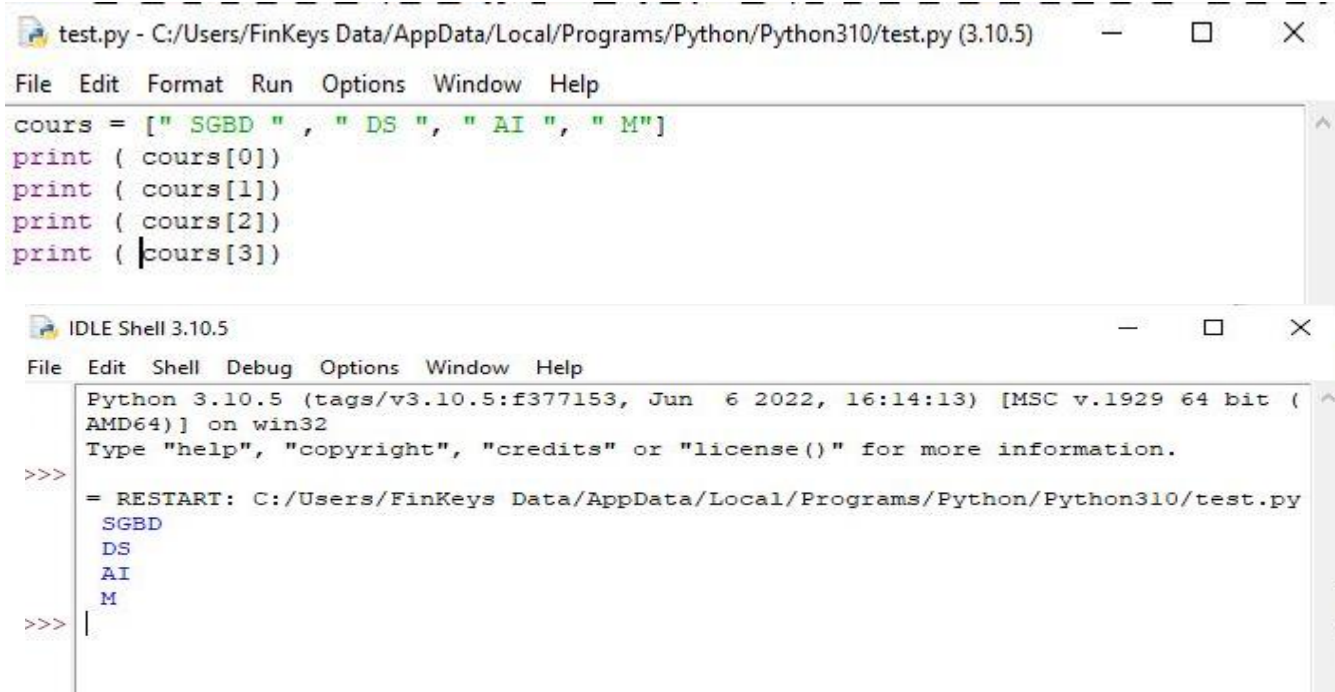




Boucles et comparaisons

Boucles et comparaisons

La boucle *for*



The screenshot shows two windows from a Python IDE. The top window, titled 'test.py', contains a list of course names and four print statements. The bottom window, titled 'IDLE Shell 3.10.5', shows the execution output of the code.

```
test.py - C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py (3.10.5)
File Edit Format Run Options Window Help
cours = [" SGBD ", " DS ", " AI ", " M"]
print ( cours[0])
print ( cours[1])
print ( cours[2])
print ( cours[3])

IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py
SGBD
DS
AI
M
>>> |
```

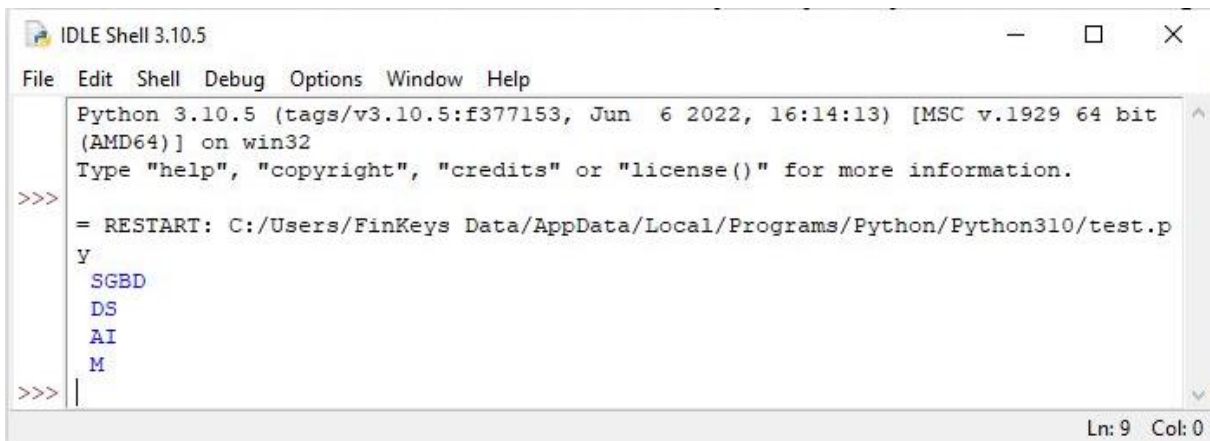
Comment
optimiser ce
code ?



Boucles et comparaisons

La boucle *for*

```
cours = [" SGBD ", " DS ", " AI ", " M"]  
for element in cours:  
    print(element)
```



```
IDLE Shell 3.10.5  
File Edit Shell Debug Options Window Help  
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit  
(AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
= RESTART: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.p  
y  
SGBD  
DS  
AI  
M  
>>> |  
Ln: 9 Col: 0
```

Avec la boucle
for



Boucles et comparaisons

La boucle for

```
cours = [" SGBD ", " DS ", " AI ", " M"]  
for element in cours:  
    print(element)  
    print(element*2)  
  
print('fini')
```

```
>>>  
=====
```

SGBD
SGBD SGBD
DS
DS DS
AI
AI AI
M
M M
fini

```
>>> |
```

Avec la boucle
for



Boucles et comparaisons

La boucle *for*

```
test.py - C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py (3.10.5)  
File Edit Format Run Options Window Help  
cours = [" SGBD ", " DS ", " AI ", " M"]  
for element in cours:  
print(element)
```

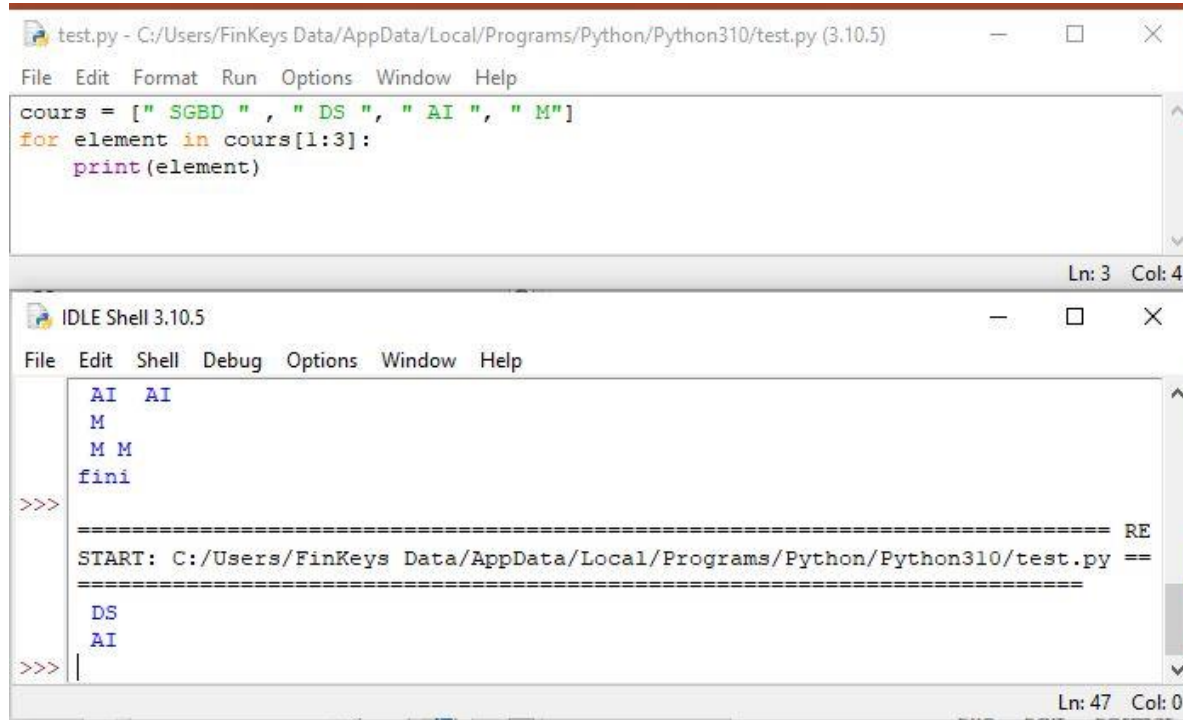


Avec la boucle
for



Boucles et comparaisons

La boucle *for*



The screenshot displays the Python IDLE 3.10.5 environment. The top window, titled 'test.py', contains the following code:

```
File Edit Format Run Options Window Help
cours = [" SGBD ", " DS ", " AI ", " M"]
for element in cours[1:3]:
    print(element)
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the execution output:

```
File Edit Shell Debug Options Window Help
AI AI
M
M M
fini

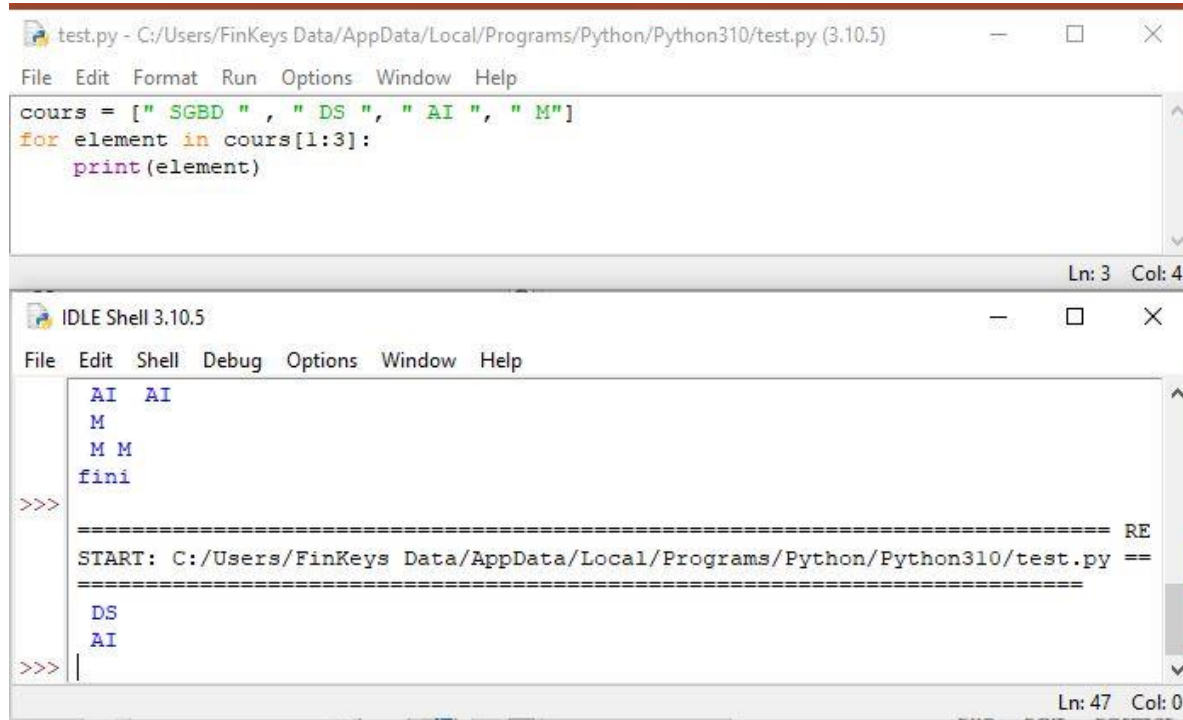
>>>
===== RE
START: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py ==
=====
DS
AI
>>> |
```

Avec la boucle
for



Boucles et comparaisons

La boucle *for*



The screenshot displays the Python IDLE 3.10.5 environment. The top window, titled 'test.py', contains the following code:

```
File Edit Format Run Options Window Help
cours = [" SGBD ", " DS ", " AI ", " M"]
for element in cours[1:3]:
    print(element)
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the execution output:

```
File Edit Shell Debug Options Window Help
AI AI
M
M M
fini

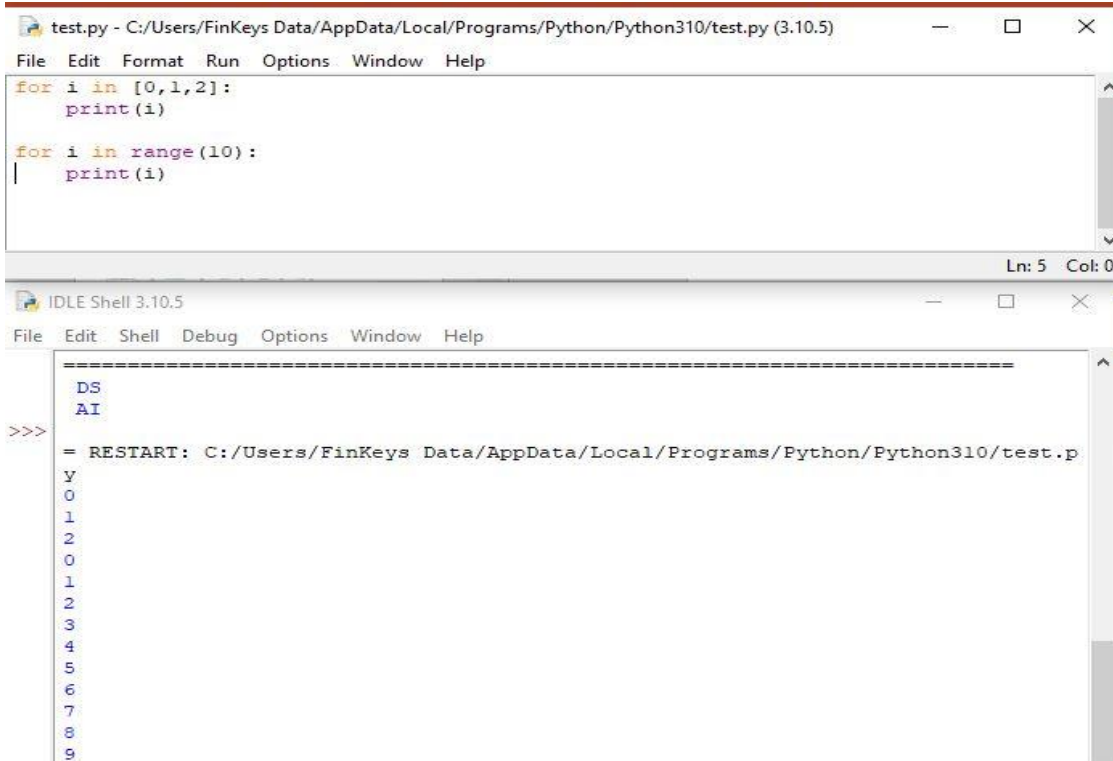
>>>
===== RE
START: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py ==
=====
DS
AI
>>> |
```

Avec la boucle
for



Boucles et comparaisons

La boucle *for* : *Fonction range()*



The screenshot displays the Python IDLE 3.10.5 environment. The top window, titled 'test.py - C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py (3.10.5)', contains the following code:

```
for i in [0,1,2]:  
    print(i)  
  
for i in range(10):  
    print(i)
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the output of the script. It includes a 'DS AI' header, a 'RESTART' message, and a list of numbers from 0 to 9, each on a new line. The shell also shows line numbers 0 through 9 on the left margin.

Avec la boucle
for



Boucles et comparaisons

La boucle *for* : *Itération sur les indices/les éléments*

```
File Edit Format Run Options Window Help
list=[14,14,15,23,28,29,33,44,75,89,12]
for i in range(10):
    print(list[i])
```



```
IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
4
5
6
7
8
9
>>>
= RESTART: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.p
Y
14
14
15
23
28
29
33
44
75
89
```

Avec la boucle
for



Boucles et comparaisons

La boucle *for* : *Itération sur les indices/les éléments*

```
File Edit Format Run Options Window Help
list=[14,14,15,23,28,29,33,44,75,89,12]
for i in range(10):
    print(list[i])

IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
4
5
6
7
8
9
>>>
= RESTART: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.p
Y
14
14
15
23
28
29
33
44
75
89
```

Avec la boucle
for



Boucles et comparaisons

Comparaisons

```
>>> a=10
>>> b=9
>>> a > b
True
>>> a < b
False
>>> 'a' < 'aa'
True
>>> 'ram' < 'zoo'
True
>>> a == 10
True
>>> a == 8
False
>>> |
```



Boucles et comparaisons

Boucles while



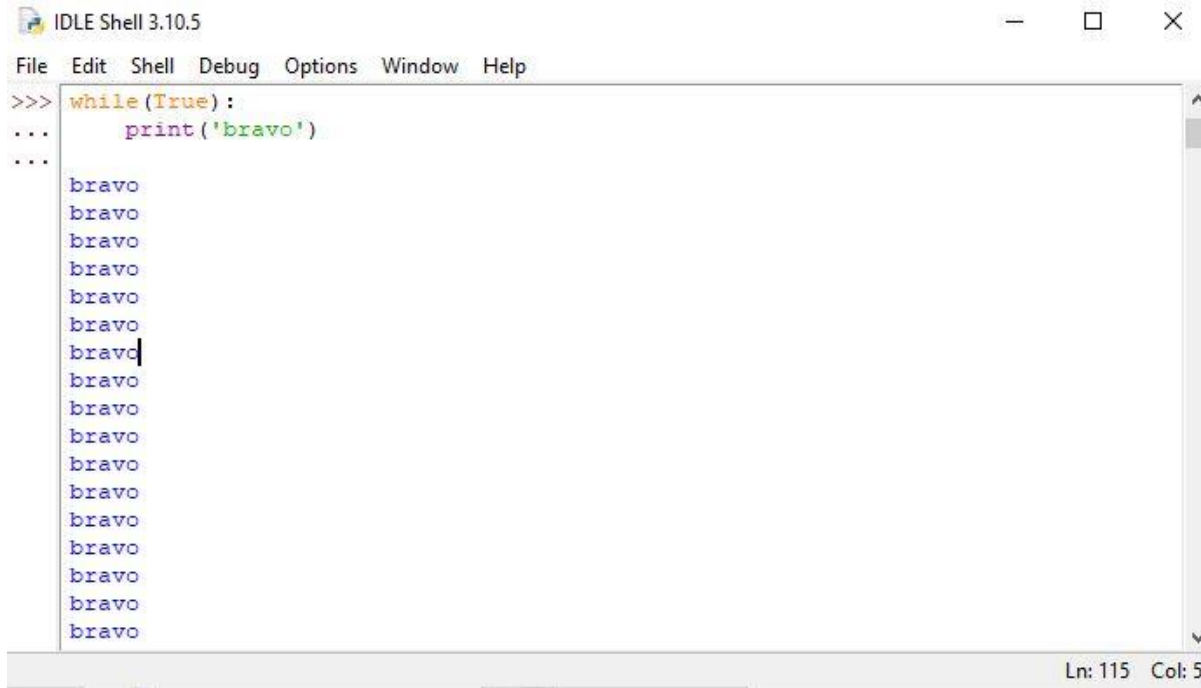
The screenshot shows the IDLE Shell 3.10.5 window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell prompt is >>>. The code entered is a while loop that prints 'Bravo' 20 times. The output shows the word 'Bravo' printed six times, with the rest of the loop iterations truncated by the image's width.

```
>>> while (i<20):  
...     print('Bravo')  
...     i=i+1  
...  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo  
Bravo
```



Boucles et comparaisons

Boucles while



The screenshot shows the IDLE Shell 3.10.5 window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code entered is:

```
>>> while (True):  
...     print('bravo')  
...  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo  
bravo
```

The status bar at the bottom right indicates "Ln: 115 Col: 5".

**Interrompre
l'exécution**





Tests

Tests

Condition If ... Un seul cas

```
>>> x='Python'
>>> if x=='Python':
...     print('True')
...
True
>>> if x=='python':
...     print('True')
...
>>> |
```



Tests

Condition If ... Plusieurs cas

```
>>> if cours=='Python':  
...     print("c'est le cours que je cherche")  
... else:  
...     print("Ce n'est pas mon cours")  
...  
c'est le cours que je cherche  
>>> |
```



Tests

Condition If ... Plusieurs cas

```
>>> if cours=='SGBD':
...     print('Yes')
... elif cours=='Python':
...     print('Non')
...
Non
>>> if cours=='SGBD':
...     print('Yes')
... elif cours=='python':
...     print('Non')
... elif cours=='Python':
...     print('Yes')
...
Yes
>>> |
```



Tests

Condition If ... Importance de l'indentation

```
>>> for element in cours:
...     if element=='Machine Learning':
...         print("c'est un cours intéressant")
...         print(f'Inscrivez vous sur le cours {element}')
...
c'est un cours intéressant
Inscrivez vous sur le cours Machine Learning
>>> for element in cours:
...     if element=='Machine Learning':
...         print("c'est un cours intéressant")
...         print(f'Inscrivez vous sur le cours {element}')
...
Inscrivez vous sur le cours SGBD
c'est un cours intéressant
Inscrivez vous sur le cours Machine Learning
Inscrivez vous sur le cours Big Data
>>> |
```



Tests

Condition If ... Tests multiples

Rappel sur le fonctionnement de l'opérateur OU

Condition 1	Opérateur	Condition 2	Résultat
Vrai	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Faux	OU	Vrai	Vrai
Faux	OU	Faux	Faux



Tests

Condition If ... Tests multiples

Rappel sur le fonctionnement de l'opérateur ET

Condition 1	Opérateur	Condition 2	Résultat
Vrai	ET	Vrai	Vrai
Vrai	ET	Faux	Faux
Faux	ET	Vrai	Faux
Faux	ET	Faux	Faux



Tests

Condition If ... Tests multiples

```
>>> Cours1='SGBD'
>>> Cours2='ML'
>>> if Cours1=='SGBD' and Cours2=='ML':
...     print('Super')
...
Super
```

```
>>> if Cours1=='SGBD':
...     if Cours2=='ML':
...         print('Super')
...
Super
>>> True or False
True
```



Tests

Condition If ... Tests multiples

```
>>> False and False
False
>>> True and True
True
>>> if Cours1=='SGBD' or Cours2=='Big data':
...     print('true')
...
true
>>> |
```

```
>>> not True
False
>>> not False
True
>>> not(True and True)
False
>>> |
```



Tests

Instructions break et continue

```
>>> for i in range(10):  
...     if i<=5:  
...         print(i)  
...         break  
...  
0  
>>> for i in range(10):  
...     if i<=5:  
...         print(i)  
...     else:  
...         break  
...  
0  
1  
2  
3  
4  
5
```

```
>>> for i in range (10):  
...     if i == 5:  
...         continue  
...     print (i)  
...  
0  
1  
2  
3  
4  
6  
7  
8  
9  
>>> |
```



Tests

Instructions break et continue

```
>>> for i in range(10):  
...     if i<=5:  
...         print(i)  
...         break  
...  
0  
>>> for i in range(10):  
...     if i<=5:  
...         print(i)  
...     else:  
...         break  
...  
0  
1  
2  
3  
4  
5
```

```
>>> for i in range (10):  
...     if i == 5:  
...         continue  
...     print (i)  
...  
0  
1  
2  
3  
4  
6  
7  
8  
9  
>>> |
```





Fichiers

Fichiers

La fonction readlines()

Créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom Notes.txt et le contenu suivant :



```
Notes - Bloc-notes
Fichier Edition Format Affichage Aide
$GBD:20
DS:15
BIG DATA:16
Python:20
```



Fichiers

La fonction readlines()

```
>>> fichiers_notes = open ("Notes.txt", "r")
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in <module>
    fichiers_notes = open ("Notes.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'Notes.txt'
>>> fichiers_notes = open ("C:\\Users\\FinKeys Data\\Notes.txt", "r")
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
on 2-3: truncated \UXXXXXXXX escape
```

```
>>> fichiers_notes = open ("C:\\\\Users\\\\FinKeys Data\\\\Notes.txt", "r")
>>> fichiers_notes.readlines()
['SGBD:20\n', 'DS:15\n', 'BIG DATA:16\n', 'Python:20\n']
>>> fichiers_notes.close()
>>> fichiers_notes.readlines()
Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    fichiers_notes.readlines()
ValueError: I/O operation on closed file.
>>> |
```



Fichiers

La fonction readlines()

```
>>> fichiers_notes = open ("Notes.txt", "r")
Traceback (most recent call last):
  File "<pyshell#93>", line 1, in <module>
    fichiers_notes = open ("Notes.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'Notes.txt'
>>> fichiers_notes = open ("C:\\Users\\FinKeys Data\\Notes.txt", "r")
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in
on 2-3: truncated \UXXXXXXXX escape
```

```
>>> fichiers_notes = open ("C:\\\\Users\\\\FinKeys Data\\\\Notes.txt", "r")
>>> fichiers_notes.readlines()
['SGBD:20\n', 'DS:15\n', 'BIG DATA:16\n', 'Python:20\n']
>>> fichiers_notes.close()
>>> fichiers_notes.readlines()
Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    fichiers_notes.readlines()
ValueError: I/O operation on closed file.
>>> |
```



Fichiers

Itérations sur le fichier

```
>>> fichiers_notes = open ("C:\\Users\\FinKeys Data\\Notes.txt", "r")
>>> for ligne in fichiers_notes:
...     print(ligne)
...
...
SGBD:20

DS:15

BIG DATA:16

Python:20
>>> |
```



Fichiers

Itérations sur le fichier

```
>>> with open ("C:\\Users\\FinKeys Data\\Notes.txt", "r") as fichiers_notes
...     for ligne in fichiers_notes:
...         print(ligne)
...
SGBD:20

DS:15

BIG DATA:16

Python:20

>>> |
```



Fichiers

La Méthode read()

```
>>> with open ("C:\\Users\\FinKeys Data\\Notes.txt", "r") as fichiers_notes:
...     fichiers_notes.read()
...
...
...
'SGBD:20\nDS:15\nBIG DATA:16\nPython:20\n'
>>> |
```



La Méthode readline()

[illegible]

Fichiers

Ecrire dans un fichier

```
>>> with open ("C:\\Users\\FinKeys Data\\Notes.txt", "w") as fichiers_notes:
...     for element in notes:
...         fichiers_notes.write(element)
...
...
8
16
6
```

Notes - Bloc-notes

Fichier Edition Format Affichage Aide

PLSQL:12Algorithmique:14TEC:15



Fichiers

Ecrire dans un fichier

```
>>> with open ("C:\\Users\\FinKeys Data\\Notes.txt", "w") as fichiers_notes:
...     for element in notes:
...         fichiers_notes.write(f'{element}\n')
...
...
9
17
7
>>>
```

Notes - Bloc-notes

Fichier Edition Format Affichage Aide

PLSQL:12
Algorithmique:14
TEC:15



Fichiers

Ouvrir deux fichiers avec l'instruction with

```
>>> with open ("C:\\Users\\FinKeys Data\\Notes.txt", "r") as fichiers_notes1, c
...      ("C:\\Users\\FinKeys Data\\Notes1.txt", "w") as fichier_notes2:
...
...     for element in fichiers_notes1:
...
...         fichier_notes2.write(f"## {element}\n")
...
...
13
21
11
>>>
```

Notes1 - Bloc-notes

Fichier Edition Format Affichage Aide

PLSQL:12

Algorithmique:14

TEC:15





Fonctions

Fonctions

Définition

Les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme.
Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques.

Len()

Cos()

Sin()

Append() est une méthode.

Fonction

1. Paramètre,
2. Code principal
3. Résultat

Méthode

1. Paramètre,
2. Code principal



Fonctions

Comment programmer?

```
>>> def somme(x,y):  
...     return(x+y)  
...  
>>> somme(41,55)  
96  
>>> def sous(x,y):  
...     return(x-y)  
...  
>>> sous(47,88)  
-41  
>>> def hello():  
...     print('hi')  
...  
>>> hello()  
hi  
>>> sous(78,88)  
-10  
>>> var=sous(78,88)  
>>> var  
-10
```

```
>>> var2=hello()  
hi  
>>> var2  
.
```



Fonctions

Les arguments & Types

- Le nombre des arguments à choisir est variable selon le programmeur.
- Le type des variables à inclure, python est un langage au « typage dynamique »



Fonctions

Renvoi de résultats

```
>>> def carre_cube(x):  
...     return x**2,x**3  
...  
>>> carre_cube(10)  
(100, 1000)  
>>> def carre_cube(x):  
...     return [x**2,x**3]  
...  
>>>  
>>> carre_cube(10)  
[100, 1000]  
>>> x,y=carre_cube(10)  
>>> x  
100  
>>> y  
1000  
>>> |
```



Fonctions

Arguments positionnels

```
>>> def multiplication(x,y):  
...     return x*y  
...  
>>> multiplication(10)  
Traceback (most recent call last):  
  File "<pyshell#229>", line 1, in <module>  
    multiplication(10)  
TypeError: multiplication() missing 1 required positional argument: 'y'  
>>> |
```



Fonctions

Arguments par mot-clé

```
>>> def fonc(x=7):  
...     return x  
...  
>>> fonc()  
7  
>>> fonc(10)  
10  
>>> |
```

```
>>> def fonc(x=1,y=2,z=4):  
...     return x , y, z  
...  
>>> fonc()  
(1, 2, 4)  
>>> fonc(10,25,77)  
(10, 25, 77)  
>>> fonc(10)  
(10, 2, 4)  
>>> fonc(10,11)  
(10, 11, 4)  
>>> |
```

```
>>> fonc(x=0)  
(0, 2, 4)  
>>> fonc(z=10,y=7,x=78)  
(78, 7, 10)  
>>> |
```



Fonctions

Arguments par mot-clé

```
>>> def fonc(x,y,z=0):  
...     return x,y,z  
...  
>>> fonc(10,20)  
(10, 20, 0)  
>>> fonc(10)  
Traceback (most recent call last):  
  File "<pyshell#249>", line 1, in <module>  
    fonc(10)  
TypeError: fonc() missing 1 required positional argument: 'y'  
>>> |
```

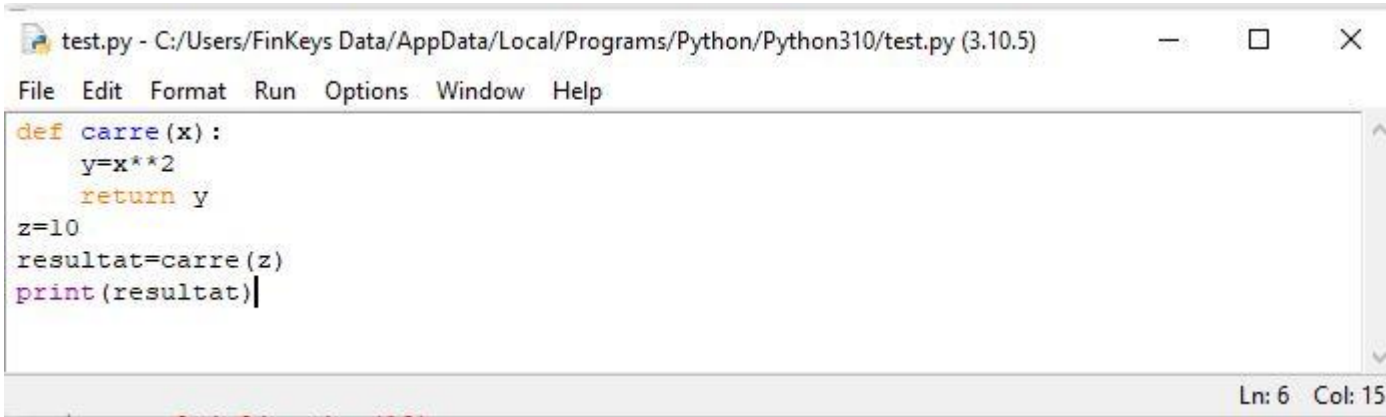


Fonctions

Variables locales et variables globales

Une variable est dite **locale** lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Une variable est dite **globale** lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme



```
test.py - C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/test.py (3.10.5)
File Edit Format Run Options Window Help
def carre(x):
    y=x**2
    return y
z=10
resultat=carre(z)
print(resultat)
```

Ln: 6 Col: 15



Fonctions

Principe DRY

```
>>> z_carre=z**3  
>>> y=10  
>>> y_carre=y**3  
>>> c=12  
>>> c_carre=c**3  
>>> |
```

```
>>> def carre(x):  
...     return x**2  
...  
>>> |
```



Don't Repeat Yourself.





**Plus sur les chaînes
de caractères**

Plus sur les chaines de caractères

Chaine de caractères et listes

```
>>> Cours="Machine Learning"
>>> len(Cours)
16
>>> Cours[4]
'i'
>>> Cours[:10]
'Machine Le'
>>> Cours[4:-3:2]
'ieLan'
>>> Cours[1:5]
'achi'
>>> |
```



Plus sur les chaines de caractères

Limitation : changer une chaine de caractères

```
>>> Cours[10]
'a'
>>> Cours[10]='t'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    Cours[10]='t'
TypeError: 'str' object does not support item assignment
>>> |
```



Créer une nouvelle chaine



Plus sur les chaines de caractères

Caractères spéciaux

```
>>> Description_projet="Prédiction du pouvoir d'achat du client"
>>> Description_projet
"Prédiction du pouvoir d'achat du client"
>>> Description_projet='Prédiction du pouvoir d'achat du client'
SyntaxError: unterminated string literal (detected at line 1)
>>> Description_projet="Prédiction \n du pouvoir \t d'achat du client \"
>>> print(Description_projet)
Prédiction
  du pouvoir      d'achat du client "
>>> Description_projet
'Prédiction \n du pouvoir \t d\'achat du client "'
>>> x = """ ML
...   DL SGBD """
>>> x
' ML\n DL SGBD '
>>> print(x)
ML
DL SGBD
>>> |
```



Plus sur les chaines de caractères

Préfixe de chaîne de caractères

f-strings ➡ « *f* » est un préfixe de chaîne de caractères » ou stringprefix.

Raw string ➡ « *r* » force la non-interprétation des caractères spéciaux

```
>>> chaine=r"voilà la première ligne \n et la deuxieme ligne"
>>> chaine
'voilà la première ligne \\n et la deuxieme ligne'
>>> print(chaine)
voilà la première ligne \n et la deuxieme ligne
>>> chaine="voilà la première ligne \n et la deuxieme ligne"
>>> print(chaine)
voilà la première ligne
    et la deuxieme ligne
>>> |
```



Plus sur les chaînes de caractères

Méthodes associées aux chaînes de caractères

```
>>> chaine="voilà un exemple"
>>> chaine.split()
['voilà', 'un', 'exemple']
>>> |
```

```
>>> chaine = "une:chaine:de::caractères"
```

```
>>> chaine.split(":")
['une', 'chaine', 'de', '', 'caractères']
```



Plus sur les chaînes de caractères

Méthodes associées aux chaînes de caractères

```
>>> chaine: "voilà mon text"
>>> chaine.split(maxsplit=3)
['une:chaine:de::caractères']
>>> chaine="voilà mon text"
>>> chaine.split(maxsplit=3)
['voilà', 'mon', 'text']
>>> chaine.split(maxsplit=2)
['voilà', 'mon', 'text']
>>> |
```

```
>>> chaine.split(maxsplit=1)
['une:chaine:de::caractères']
>>> chaine.split(maxsplit=2)
['une:chaine:de::caractères']
>>> chaine.split(maxsplit=3)
['une:chaine:de::caractères']
>>> chaine: "voilà mon text"
>>> chaine.split(maxsplit=3)
['une:chaine:de::caractères']
```



Plus sur les chaînes de caractères

Méthodes associées aux chaînes de caractères

```
>>> cours="Big data"
...
>>> cours.find("i")
...
1
>>> cours.find("B")
...
0
>>> cours.find('b')
...
-1
>>> cours.find("z")
...
-1
>>> cours.replace("data", "données")
...
'Big données'
>>> cours.capitalize()
...
'Big data'
>>> cours="big data"
...
>>> cours.capitalize()
...
'Big data'
>>> cours.count('a')
...
2
```

```
>>> cours.startswith("data")
...
False
>>> cours.startswith('big')
...
True
>>> |
```



Plus sur les chaînes de caractères

Méthodes associées aux chaînes de caractères

```
>>> chaine= "  comment enlever les espaces supplémentaires"  
...  
>>> chaine.strip()  
...  
'comment enlever les espaces supplémentaires'
```

```
>>> chaine=" \t test \n"  
...  
>>> chaine.strip()  
...  
'test'  
>>>
```



Plus sur les chaines de caractères

Extraction des valeurs numériques à partir une chaine de caractères

```
>>> test="21.5 889.5 des valeurs manquantes"
...
>>> test.split()
...
['21.5', '889.5', 'des', 'valeurs', 'manquantes']
>>> val=test.split()
...
>>> somme = float(val[0])+float(val[1])
...

>>> somme
...
911.0
>>> |
```



Plus sur les chaînes de caractères

Conversion d'une liste de chaînes de caractères en une chaîne de caractères

```
>>> chaine=["c","o","u","r","s"]
...
>>> val="-".join(chaine)
...
>>> val
...
'c-o-u-r-s'
>>> val=" ".join(chaine)
```

```
>>> val
...
'c o u r s'
>>> val="".join(chaine)
...
>>> val
...
'cours'
>>> |
```



Plus sur les chaînes de caractères

Conversion d'une liste de chaînes de caractères en une chaîne de caractères

```
>>> chaine=["Z",3,"Y"]
...
>>> "".join(chaine)
...
Traceback (most recent call last):
  File "<pyshell#89>", line 1, in <module>
    "".join(chaine)
TypeError: sequence item 1: expected str instance, int found
```

```
...
>>> chaine=["Z","3","Y"]
...
>>> "".join(chaine)
...
'Z3Y'
>>> |
```





**Plus sur
les fonctions**

Plus sur les fonctions

Appel d'une fonction dans une fonction

```
>>> def carre(x ):
...     return (x **2)
...
>>> carre(2)
4
>>> def calc(debut,fin):
...     liste=[]
...     for i in range(debut,fin+1):
...         liste.append(carre(i))
...     return liste
...
>>> print(calc(5,9))
[25, 36, 49, 64, 81]
>>> |
```



Plus sur les fonctions

Fonction récurives: Factorielle

```
>>> def factorielle(x):  
...     if x == 1:  
...         return x  
...     else:  
...         return x * factorielle(x-1)  
... 
```

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$



Plus sur les fonctions

Portée sur des variables

```
>>> def f1():
...     x='f1'
...     print(f'on x = {x}')
...
...
>>> f1()
on x = f1

>>> print(x)
Traceback (most recent call last):
  File "<pyshell#157>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined

>>> def f2():
...     print(x)
...
...
>>> x='f2'
>>> f2()
f2
>>> |
```



Plus sur les fonctions

Portée sur des variables

```
>>> def ma_f1():
...     print(x)
...
>>> x=2
>>> ma_f1()
2
>>> def ma_f2():
...     x=x+1
...
...     |
>>> x=3
>>> ma_f2()
Traceback (most recent call last):
  File "<pyshell#172>", line 1, in <module>
    ma_f2()
  File "<pyshell#170>", line 2, in ma_f2
    x=x+1
UnboundLocalError: local variable 'x' referenced before assignment
```



Plus sur les fonctions

Portée sur des variables

```
>>> def ma_f3():  
...     glablal x  
...  
SyntaxError: invalid syntax  
>>> def ma_f3():  
...     global x  
...     x=x+1  
...  
...  
>>> x=2  
>>> ma_f3()  
>>> x  
3  
>>> x=10  
>>> ma_f3()  
>>> x  
11  
>>>
```



Plus sur les fonctions

LGI : Locale, Globale, Interne.

```
>>> def f1():  
...     x=2  
...     x=x+1  
...     print(x)  
...  
...  
>>> x=4  
>>> f1()  
3  
>>> |
```

EVITEZ LES VARIABLES
GLOBALES





Containers, dictionnaires, tuples et sets

Containers, dictionnaires, tuples et sets

Containers : Définition.

Un container est un nom générique pour définir un objet Python qui contient une collection d'autres objets



Containers, dictionnaires, tuples et sets

Containers : Propriétés .

- Capacité des **tests d'appartenance**.
- Capacité à **supporter la fonction len()** renvoyant la longueur du container.
- Un container peut être :
- **Ordonné (ordered en anglais)** : il y a un ordre précis des éléments; cet ordre correspond à celui utilisé lors de la création ou de la modification du container (si cela est permis); ce même ordre est utilisé lorsqu'on itère dessus;
- **Indexable (subscriptable en anglais)** : on peut retrouver un élément par son indice (i.e. sa position dans le container) ou plusieurs éléments avec une tranche; en général, tout container indexable est ordonné;
- **Itérable (iterable en anglais)** : on peut faire une boucle dessus. Certains containers sont appelés objets séquentiels ou séquence.



Containers, dictionnaires, tuples et sets

Dictionnaires: déclaration.

```
>>> dictionnaire_notes={}
>>> dictionnaire_notes['ML']=15
>>> dictionnaire_notes['DL']=17
>>> dictionnaire_notes['Python']=19
>>> dictionnaire_notes
{'ML': 15, 'DL': 17, 'Python': 19}
>>> dictionnaire_notes['English']=20
>>> dictionnaire_notes
{'ML': 15, 'DL': 17, 'Python': 19, 'English': 20}
>>> dictionnaire_notes['ML']
15
>>> |
```



Containers, dictionnaires, tuples et sets

Dictionnaires:

Itération sur les clés pour obtenir les valeurs.

```
>>> for key in dictionnaire_notes:  
...     print(key, dictionnaire_notes[key])  
...  
ML 15  
DL 17  
Python 19  
English 20  
>>> |
```



Containers, dictionnaires, tuples et sets

Dictionnaires:

Méthodes .keys(), .values() et .items()

```
>>> dictionnaire_notes.keys()
dict_keys(['ML', 'DL', 'Python', 'English'])
>>> dictionnaire_notes.values()
dict_values([15, 17, 19, 20])
>>> dictionnaire_notes.items()
dict_items([('ML', 15), ('DL', 17), ('Python', 19), ('English', 20)])
>>> list(dictionnaire_notes.items())
[('ML', 15), ('DL', 17), ('Python', 19), ('English', 20)]
>>> dictionnaire_notes.items()[0]
Traceback (most recent call last):
  File "<pyshell#216>", line 1, in <module>
    dictionnaire_notes.items()[0]
TypeError: 'dict_items' object is not subscriptable
>>> for key, val in dictionnaire_notes.items():
...     print(key, val)
...
...
ML 15
DL 17
Python 19
English 20
```



Containers, dictionnaires, tuples et sets

Dictionnaires: Tri

```
>>> sorted(dictionnaire_notes, key=dictionnaire_notes.get)
['ML', 'DL', 'Python', 'English']
>>> sorted(dictionnaire_notes)
['DL', 'English', 'ML', 'Python']
>>> for key in sorted(dictionnaire_notes):
...     print(key, dictionnaire_notes[key])
...
...
DL 17
English 20
ML 15
Python 19
>>> |
```



Containers, dictionnaires, tuples et sets

Dictionnaires: Tri

```
>>> etud1={'nom':'Jon','note':15}
>>> etud2={'nom':'Ahmed','note':12}
>>> etud=[etud1,etud2]
>>> etud
[{'nom': 'Jon', 'note': 15}, {'nom': 'Ahmed', 'note': 12}]
```

```
>>> for element in etud:
...     print(element['nom'])
...
Jon
Ahmed
>>> dictionnaire_notes
{'ML': 15, 'DL': 17, 'Python': 19, 'English': 20}
>>> dictionnaire_notes['ML']
15
>>>
```



Containers, dictionnaires, tuples et sets

Dictionnaires: Tri

```
>>> Liste_cours = [{" DL ", 2] , [{" ML " , 3]}
>>> dict(Liste_cours)
{' DL ': 2, ' ML ': 3}
>>> |
```



Containers, dictionnaires, tuples et sets

Tuples:Définitions

Les tuples (« n-uplets » en français) sont des objets séquentiels correspondant aux listes (itérables, ordonnés et indexables) mais ils sont toutefois non modifiables.



Containers, dictionnaires, tuples et sets

Tuples:Définitions

```
>>> T=(1,2,3,4,5)
>>> T[0]
1
>>> T[0:3]
(1, 2, 3)
>>> T[1]=3
Traceback (most recent call last):
  File "<pyshell#241>", line 1, in <module>
    T[1]=3
TypeError: 'tuple' object does not support item assignment
>>> |

>>> T=T+(10,)
>>> T
(1, 2, 3, 4, 5, 10)
>>> |

>>> S=1,2,3
>>> S
(1, 2, 3)
>>> |
```



Containers, dictionnaires, tuples et sets

Tuples: Définitions

Les listes, les dictionnaires et les tuples sont des containers, c'est-à-dire qu'il s'agit d'objets qui contiennent une collection d'autres objets. En Python, on peut construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc. Les combinaisons sont infinies !



Containers, dictionnaires, tuples et sets

Tuples: Définitions

```
>>> Liste=[1,2,3]
>>> t=(Liste,'cours')
>>> t[0]
[1, 2, 3]
>>> t[0].append(15)
>>> t
([1, 2, 3, 15], 'cours')
>>> |
```

Tuples
modifiables ??



Containers, dictionnaires, tuples et sets

Sets et frozensets : Définitions

Les objets de type set représentent un autre type de containers qui peut se révéler très pratique. Ils ont la particularité d'être modifiables, non hachables, non ordonnés, non indexables et de ne contenir qu'une seule copie maximum de chaque élément.



Containers, dictionnaires, tuples et sets

Sets et frozensets : Définitions

```
>>> S={1,5,2,8,5}
>>> S
{8, 1, 2, 5}

>>> type(S)
<class 'set'>
>>> |

>>> d={1,5,(8,5)}
>>> d
{1, 5, (8, 5)}
>>> |

>>> set([1,2,3])
{1, 2, 3}

>>> set({'key':'value','key2':'value2'})
{'key', 'key2'}
>>> |
```



Containers, dictionnaires, tuples et sets

Sets et frozensets : Définitions

```
>>> s=set([1,2,3])
>>> s
{1, 2, 3}
>>> type(s)
<class 'set'>
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#267>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
>>> t
([1, 2, 3, 15], 'cours')
>>> t[1]
'cours'
>>> s.add(10)
>>> s
{10, 1, 2, 3}
>>> s.discard(2)
>>> s
{10, 1, 3}
>>>
```



Containers, dictionnaires, tuples et sets

Sets : Utilité

```
>>> import random
>>> s=[random.randint(0,15) for i in range(10)]
>>> s
[7, 7, 15, 11, 4, 6, 0, 2, 10, 12]
>>> set(s)
{0, 2, 4, 6, 7, 10, 11, 12, 15}
>>> list(set(s))
[0, 2, 4, 6, 7, 10, 11, 12, 15]
>>> |
```





Modules

Modules

Définition

Ils contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou libraries).



Modules

Importation des modules

```
--
>>> import random
>>> random.randint(0,12)
5
>>> random.randint(0,12)
3
>>> random.randint(0,12)
4
>>> from random import randint
>>> random.randint(0,12)
6
>>> randint(0,12)
6
>>> |
```

```
--
>>> import math
>>> s=math.cos(math.pi/2)
>>> s
6.123233995736766e-17
>>> |

>>> random.uniform(10,89)
22.779876345396197

x=['a','r','t','y']
random.shuffle(x)
x
['a', 't', 'r', 'y']
random.shuffle(x)
>>> x
['r', 't', 'y', 'a']
>>> |
```



Modules

Module Random : exemple des fonctions

```
>>> random.choice(x)
'r'
>>> random.choice(x)
'y'
>>> random.choice(x,k=3)
Traceback (most recent call last):
  File "<pyshell#189>", line 1, in <module>
    random.choice(x,k=3)
TypeError: Random.choice() got an unexpected keyword argument 'k'
>>> random.choices(x,k=3)
['r', 'a', 'r']
>>> random.choices(x,k=4)
['r', 'r', 'a', 't']
>>> |
```



Modules

Obtenir de l'aide : Help

```
>>> import random as rand
>>> rand.randint(10,78)
43
>>> help(rand)
Help on module random:

NAME
    random - Random variable generators.

MODULE REFERENCE
    https://docs.python.org/3.10/library/random.html

    The following documentation is automatically generated from the Python
    source files. It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations. When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    bytes
    -----
        uniform bytes (values between 0 and 255)

    integers
    -----
        uniform within range

    sequences
    -----
        pick random element
        pick random sample
        pick weighted random sample
        generate random permutation

    distributions on the real line:
    -----
        uniform
        triangular
        normal (Gaussian)
        lognormal
        negative exponential
        gamma
        beta
        pareto
        Weibull

    distributions on the circle (angles 0 to 2pi)
    -----
        circular uniform
        von Mises

    General notes on the underlying Mersenne Twister core generator:

    * The period is 2**19937-1.
    * It is one of the most extensively tested generators in existence.
    * The random() method is implemented in C, executes in a single Python step,
      and is, therefore, threadsafe.

CLASSES
    _random.Random(builtins.object)
```



Modules

Obtenir de l'aide : Help

```
>>> cours=10
>>> help(cours)
Help on int object:

class int(object)
|   int([x]) -> integer
|   int(x, base=10) -> integer
|
|   Convert a number or string to an integer, or return 0 if no arguments
```



Modules

Liste des fonctions disponibles

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'w', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_ONE', '_Sequence', '_spec', '_accumulate', '_acos', '_bisect', '_ceil', '_cos', '_e', '_exp', '_floor', '_index', '_inst', '_isfinite', '_warn', 'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'logn', 'tstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>> |
```



Création des modules

Module : Utilité

Vous vous rappelez c'est quoi l'utilité des fonctions ?

La réutilisation d'une fraction de code plusieurs fois au sein d'un même programme sans avoir à dupliquer ce code



Création des modules

Module : Utilité

La réutilisation d'une fonction ou un ensemble de fonctions dans un autre programme Python.

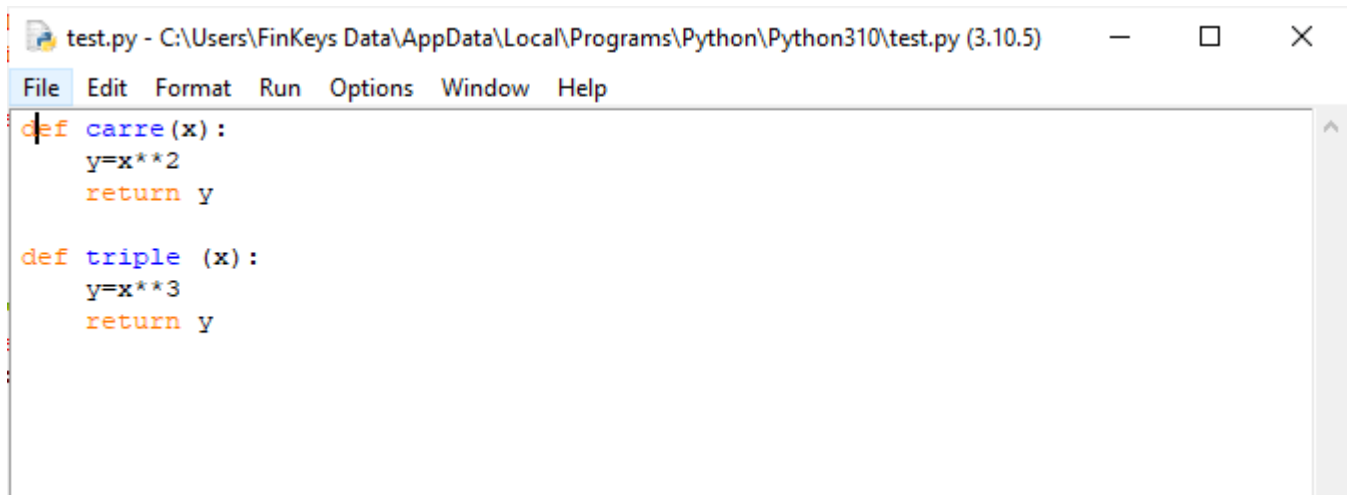
Dans un module, on met un ensemble de fonctions que l'on peut être amené à utiliser souvent. Ces modules sont regroupés autour d'un thème précis.



Création des modules

Création des modules

En Python, la création d'un module est très simple. Il suffit d'écrire un ensemble de fonctions (et/ou de constantes) dans un fichier, puis d'enregistrer ce dernier avec une extension .py (comme n'importe quel script Python).



```
test.py - C:\Users\FinKeys Data\AppData\Local\Programs\Python\Python310\test.py (3.10.5)
File Edit Format Run Options Window Help
def carre(x):
    y=x**2
    return y

def triple (x):
    y=x**3
    return y
```



Création des modules

Utilisation de son propre module

code.py - C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/code.py (3.10.5) — □ ×

File Edit Format Run Options Window Help

```
import test
print(test.carre(4))
print(test.triple(10))
```

```
>>> = RESTART: C:/Users/FinKeys Data/AppData/Local/Programs/Python/Python310/code.py
16
1000
>>> |
```

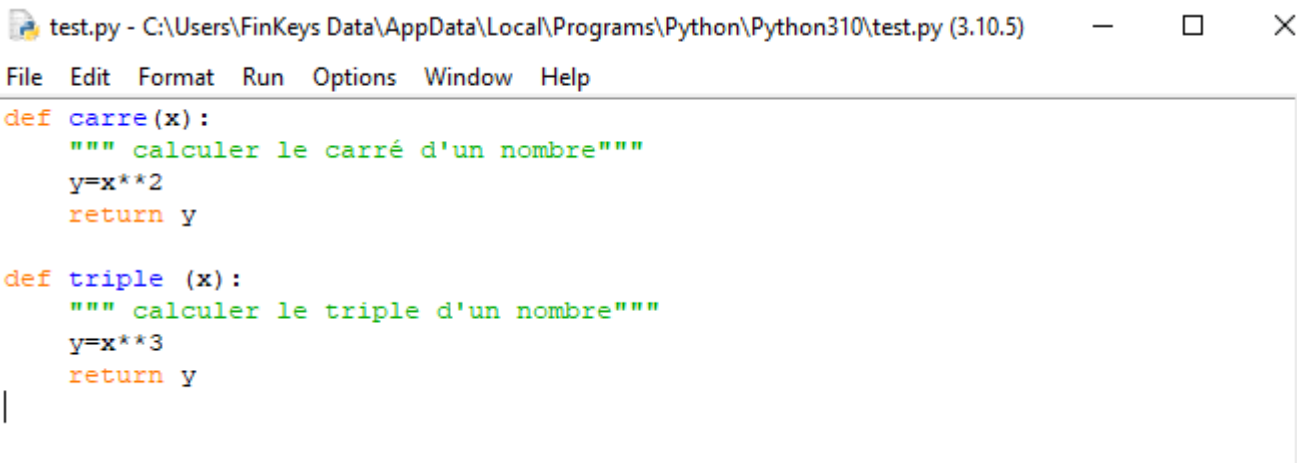
Ln: 39 Col: 0



Création des modules

Docstrings

Les chaînes de caractères entre triple guillemets situées en début du module et de chaque fonction sont là pour expliquer ce que fait le module et comment utiliser chaque fonction, on les appelle docstrings (« chaînes de documentation » en français). Ces docstrings permettent notamment de fournir de l'aide lorsqu'on invoque la commande `help()` :



```
test.py - C:\Users\FinKeys Data\AppData\Local\Programs\Python\Python310\test.py (3.10.5)
File Edit Format Run Options Window Help
def carre(x):
    """ calculer le carré d'un nombre"""
    y=x**2
    return y

def triple(x):
    """ calculer le triple d'un nombre"""
    y=x**3
    return y
```



Création des modules

Docstrings

```
Help on module test:
```

```
NAME
```

```
    test
```

```
FUNCTIONS
```

```
    carre(x)
```

```
        |         calculer le carré d'un nombre
```

```
    triple(x)
```

```
        |         calculer le triple d'un nombre
```

```
FILE
```

```
    c:\users\finkeys data\appdata\local\programs\python\python310\test.py
```



Création des modules

Visibilité des fonctions

La visibilité des fonctions au sein des modules suit des règles simples :

- Les fonctions dans un même module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé. Par exemple, si la commande `import autremodule` est utilisée dans un module, il est possible d'appeler une fonction avec `autremodule.fonction()`.



Création des modules

Module ou script?

Le module *test* ne contient que des fonctions.

Si on l'exécutait comme un script classique, *cela n'afficherait rien*.

Cela s'explique par l'absence de programme principal.

Que se passe-t-il alors si on importe un script en tant que module alors qu'il contient un programme principal avec des lignes de code ? Prenons par exemple le script *test2.py* suivant :

```
def carre(x):  
    """ calculer le carré d'un nombre"""  
    y=x**2  
    return y  
  
def triple (x):  
    """ calculer le triple d'un nombre"""  
    y=x**3  
    return y  
  
#programme principal  
print(f"le carré du nombre 2 = {carre(2)}")
```



Création des modules

Module ou script?

```
>>> = RESTART: C:\Users\FinKeys Data\AppData\Local\Programs\Python\Python310\test.py  
le carré du nombre 2 = 4  
>>> |
```

Ln: 112 Col: 0

Ce n'est pas le comportement voulu pour un module car on n'attend pas d'affichage particulier (par exemple la commande `import math`, n'affiche rien)



Création des modules

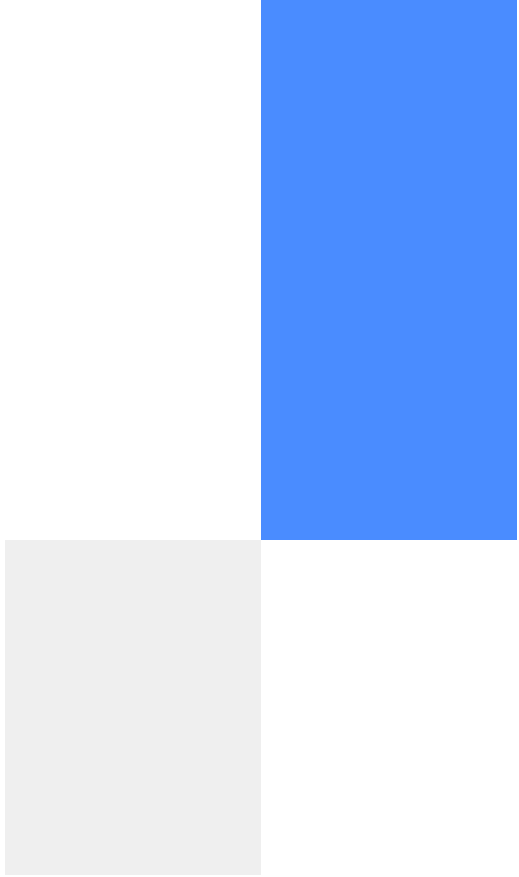
Module ou script?

```
test.py - C:\Users\FinKeys Data\AppData\Local\Programs\Python\Python310\test.py (3.10.5)
File Edit Format Run Options Window Help
def carre(x):
    """ calculer le carré d'un nombre"""
    y=x**2
    return y

def triple (x):
    """ calculer le triple d'un nombre"""
    y=x**3
    return y

#programme principal
if __name__ == "__main__":
    print(f"le carré du nombre 2 = {carre(2)}")
```





Lambda

Lambda

Définition

Une fonction lambda est une petite fonction anonyme.

Une fonction anonyme se réfère à une fonction **déclarée sans nom**. Bien que syntaxiquement elles soient différentes, les fonctions lambda se comportent de la même manière que les fonctions régulières qui sont **déclarées en utilisant le mot-clé def**.

Elle peut prendre un nombre quelconque d'arguments, mais ne peut avoir qu'une seule expression pour définir les mini fonctions.

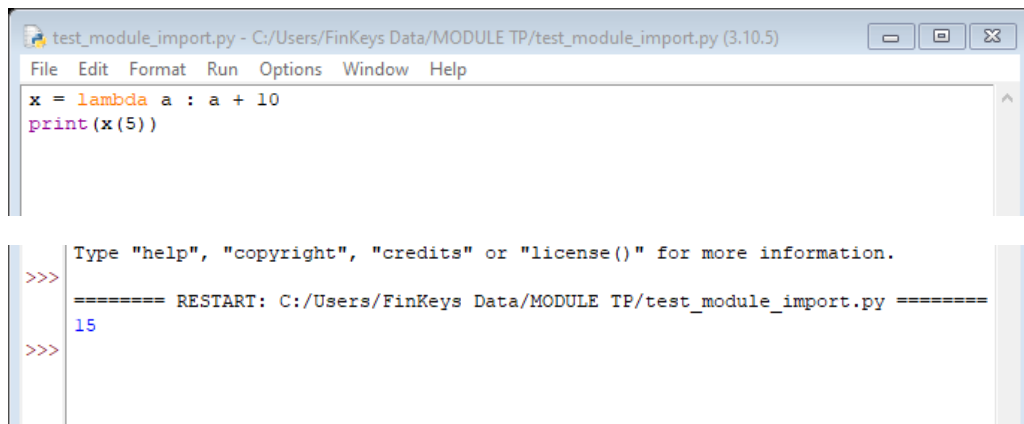
Syntaxe

`lambda arguments : expression`

Lambda

Exemple

Ajouter 10 à l'argument x, et retourner le résultat :



The screenshot shows a Python IDE window titled "test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The editor contains the following code:

```
x = lambda a : a + 10
print(x(5))
```

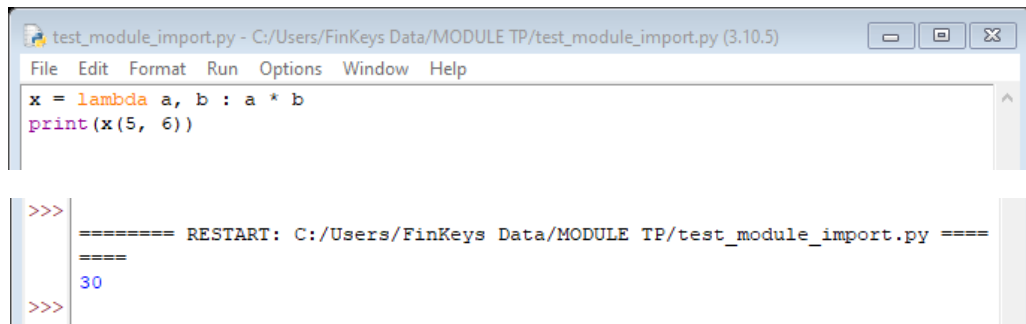
Below the editor, the Python REPL output is shown:

```
>>> Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: C:/Users/FinKeys Data/MODULE TP/test_module_import.py =====
>>> 15
>>>
```

Lambda

Exemple

Les fonctions lambda peuvent prendre un nombre quelconque d'arguments :
Multiplication deux nombre à l'aide de l'expression lambda.



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
x = lambda a, b : a * b
print(x(5, 6))

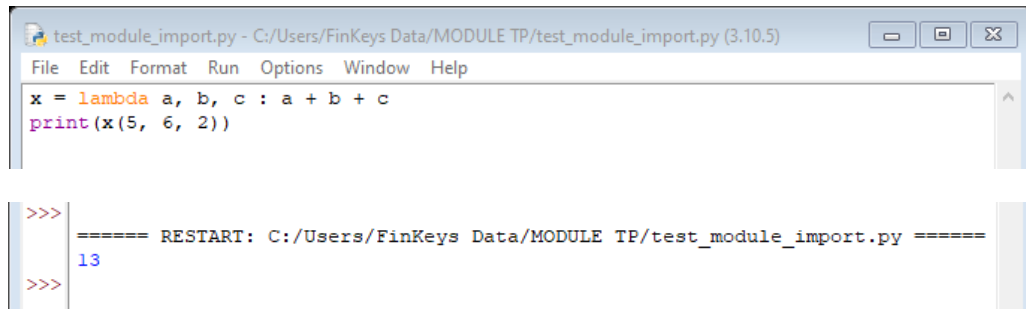
>>> ===== RESTART: C:/Users/FinKeys Data/MODULE TP/test_module_import.py =====
>>> 30
>>>
```

Lambda

Exemple

Les fonctions lambda peuvent prendre un nombre quelconque d'arguments :

Additionner a, b, et c et retourner le résultat :



The screenshot shows a Python IDE window titled "test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python code:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Below the code editor, the Python REPL output is shown:

```
>>> ===== RESTART: C:/Users/FinKeys Data/MODULE TP/test_module_import.py =====
13
>>>
```

Lambda

Exemple

Calculer le carré d'un nombre x:

```
>>> def f(x):  
...     return x*2  
...  
>>> f(3)  
6  
  
1 >>> var=lambda x:x*2  
2 >>> var(5)  
10  
3 >>> (lambda x:x*2)(3)  
6  
4 >>>
```

Fonction normale

Lambda

Lambda

Explication

1

```
>>> var=lambda x:x*2
>>> var(5)
10
```

La fonction lambda fait la même chose que la fonction ordinaire précédente $f(x)$.

La syntaxe :

- ***pas de parenthèses autour de la liste d'arguments***
- Le mot-clé **return est manquant** (*il est implicite, la fonction complète ne pouvant être qu'une seule expression*).
- La fonction **n'a pas de nom**, mais on peut l'appeler à travers la variable *var*.

Lambda

Explication

2

```
>>> (lambda x:x*2) (3)  
6  
>>>
```

Nous pouvons utiliser une fonction lambda sans l'affecter à une variable.

Lambda

Règles

- Une fonction lambda est une fonction **qui prend un nombre quelconque d'arguments** et **retourne la valeur d'une expression unique**.
- Les fonctions lambda **ne peuvent pas contenir de commandes** et **elles ne peuvent contenir plus d'une expression**.
- Si le besoin est de créer quelque chose complexe, il faut déclarer une fonction normale.
- Les fonctions lambda sont **une question de style**. Les utiliser **n'est jamais une nécessité**.

Lambda

Pourquoi utiliser les fonctions lambda ?

- Les fonctions lambda sont **une question de style**. Les utiliser **n'est jamais une nécessité**.
- L'utilisation des fonctions lambda est facultative. Cependant les fonctions lambda sont souvent utiles **lorsqu'on a besoin de lisibilité dans notre code**.

Lambda

Quand utiliser les fonctions lambda ?

Essayons ces deux exemples de code **l'un utilisant une lambda** et l'autre **une fonction classique**.

```
>>> list_ = [15,28,3,8,40,48,3003,3894,3,98]
>>> list_sup = list(filter(lambda x: x > 20, list_)) 1
>>> def list_filter(x):
...     return x > 10 2
...
...
>>> list_ = [1,2,3,87,40,49,303,34,32,98]
>>> list_sup = list(filter(list_filter, list_))
>>>
```

Ln: 36 Col: 0

Lambda

Quand utiliser les fonctions lambda ?

- Les deux exemples de codes font exactement la même tâche : filtrer les valeurs supérieures à 20 dans la liste « list_ » par la fonction python filter() .
- Le premier exemple est **beaucoup plus lisible et compréhensible (Une seule ligne)**.

```
>>> list_ = [15,28,3,8,40,48,3003,3894,3,98]
>>> list_sup = list(filter(lambda x: x > 20, list_))
>>> def list_filter(x):
...     return x > 10
...
...
>>> list_ = [1,2,3,87,40,49,303,34,32,98]
>>> list_sup = list(filter(list_filter, list_))
>>>
```

Ln: 36 Col: 0

- Les fonctions lambda peuvent être utilisé avec pandas via les fonctions apply(), applymap() and map().

Lambda

Lambda en utilisant map()

- la fonction map est utilisée pour appliquer une opération particulière à chaque élément d'une séquence. Comme filter(), elle prend également 2 paramètres :
- Une fonction qui définit l'opération à effectuer sur les éléments.
- Une ou plusieurs séquences

```
>>>
>>> s = [10,2,8,7,5,4,3,11,0, 1]
>>> result_f = map (lambda x: x*x, s)
>>> print(list(result_f))
[100, 4, 64, 49, 25, 16, 9, 121, 0, 1]
```

Ln: 102 Col: 0

Lambda

Lambda en utilisant reduce()

- La fonction `reduce`, comme `map()`, est utilisée pour appliquer une opération à chaque élément d'une séquence. Cependant, elle diffère de `map` dans son fonctionnement. Voici les étapes suivies par la fonction `reduce()` pour calculer une sortie :
 - 1) Effectuer l'opération définie sur les 2 premiers éléments de la séquence.
 - 2) Enregistrer ce résultat.
 - 3) Effectuer l'opération avec le résultat sauvegardé et l'élément suivant de la séquence.
 - 4) Répéter jusqu'à ce qu'il ne reste plus d'éléments.

Lambda

Lambda en utilisant reduce()

Elle prend également deux paramètres :

- Une fonction qui définit l'opération à effectuer.
- Une séquence (tout itérateur comme les listes, les tuples, etc.)

```
>>> from functools import reduce
>>> s = [1,2,3,4,5]
>>> var = reduce (lambda x, y: x*y, s)
>>> print(var)
120
>>>
```

Ln: 41 Col: 0

Lambda

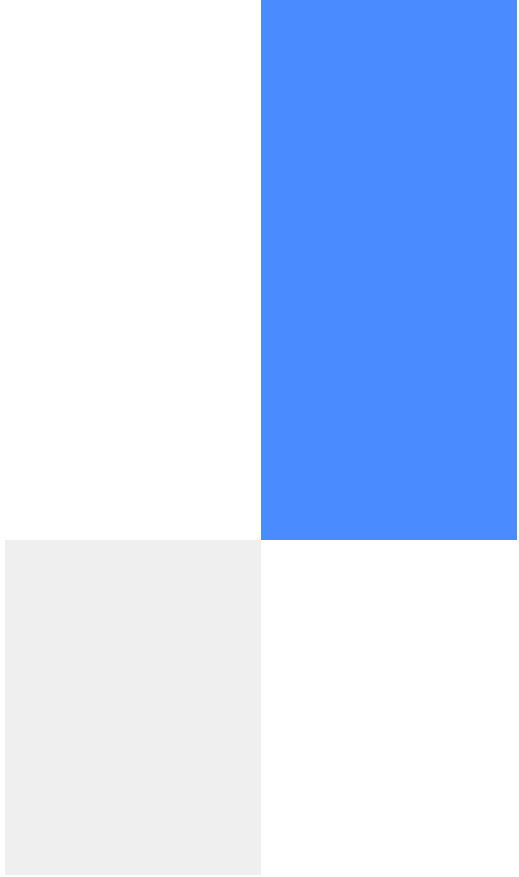
Quand ne pas utiliser les fonctions lambda ?

- Il ne faut pas écrire des fonctions lambda compliquées. Il sera très difficile pour les codeurs qui maintiennent votre code de le décrypter.
- Si vous voulez faire des expressions complexes en une seule ligne, il serait bien plus judicieux de définir une fonction appropriée.
- En tant que meilleure pratique, vous devez vous rappeler qu'un code simple est toujours préférable à un code complexe.

Lambda

Récapitulatif

- Les fonctions normales ne peuvent contenir qu'une seule expression dans leur corps.
- Les fonctions régulières peuvent avoir plusieurs expressions et instructions dans leur corps.
- Les lambdas n'ont pas de nom qui leur est associé. C'est pourquoi elles sont également appelées fonctions anonymes.
- Les fonctions régulières doivent avoir un nom et une signature.
- Les lambdas ne contiennent pas d'instruction de retour car le corps est automatiquement retourné.
- Les fonctions qui doivent retourner une valeur doivent inclure une déclaration `return`.



Map ()
Reduce ()
Filter ()

Map() Reduce() Filter()

Généralités

- Python comprend un certain nombre de **fonctions intégrées prédéfinies** qui peuvent être utilisées par l'utilisateur final **en les appelant simplement**. Ces fonctions facilitent non seulement le travail des programmeurs, mais elles contribuent également à établir **un environnement de codage commun**.
- Les trois piliers de la programmation fonctionnelle (Fonctions pour faire le calcul) sont les fonctions **map, reduce et filter**.

Map() Reduce() Filter()

Généralités

- En programmation fonctionnelle, **les fonctions d'ordre supérieur** sont le principal outil pour définir le calcul. Il s'agit de fonctions qui **prennent une fonction comme paramètre** et renvoient **une fonction comme résultat**.
- **Reduce(), map() et filter()** sont trois des fonctions d'ordre supérieur **les plus utiles de Python**.
- Elles peuvent être utilisées pour effectuer des **opérations complexes** lorsqu'elles sont associées **à des fonctions plus simples**.

Map() Reduce() Filter()

Généralités

- Les fonctions `map()`, `reduce()` et `filter()` peuvent prendre **une autre fonction comme argument**.
- Elles **prennent chacune une fonction et une liste d'éléments**, puis renvoient le résultat de l'application de la fonction **à chaque élément de la liste**.
- Python dispose de fonctions intégrées telles que `map()`, `reduce()` et `filter()`. La fonctionnalité de programmation fonctionnelle de Python est activée par ces fonctions.
- Les seuls facteurs qui déterminent le résultat dans la programmation fonctionnelle sont **les entrées transmises**.
- Ces fonctions **peuvent accepter n'importe quelle autre fonction** comme paramètre et peuvent également être transmises comme paramètres à d'autres fonctions.

Map() Reduce() Filter()

Map() : Définition

- La fonction `map()` est une fonction **d'ordre supérieur**.
- Comme indiqué précédemment, cette fonction **accepte une autre fonction et une séquence d'"itérables" comme paramètres** et fournit une sortie après avoir **appliqué la fonction à chaque itérable de la séquence**.

Syntaxe

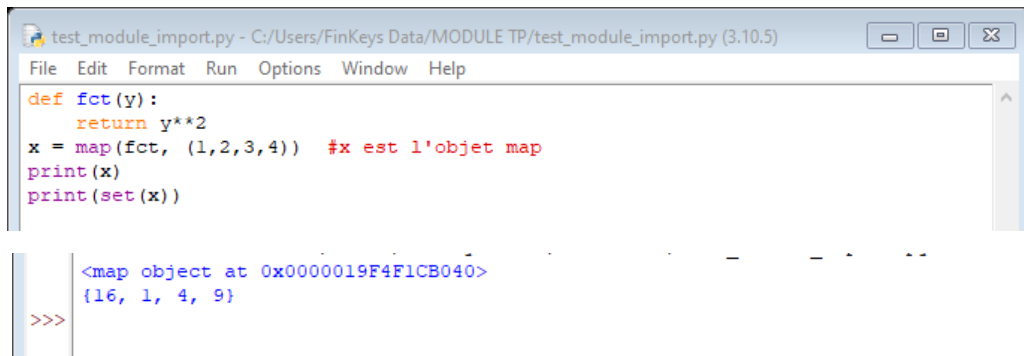
map(function, iterables)

Map() Reduce() Filter()

Map() : Définition

- la fonction est utilisée pour définir une expression qui est ensuite appliquée aux "itérables".
- Les fonctions définies par l'utilisateur et les fonctions lambda peuvent toutes deux être envoyées à la fonction map.

Exemple

A screenshot of a Python IDE window titled 'test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)'. The window has a menu bar with 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The code editor contains the following Python code:

```
def fct(y):  
    return y**2  
x = map(fct, (1,2,3,4)) #x est l'objet map  
print(x)  
print(set(x))
```

The output console at the bottom shows the result of running the code:

```
<map object at 0x0000019F4F1CB040>  
{16, 1, 4, 9}
```

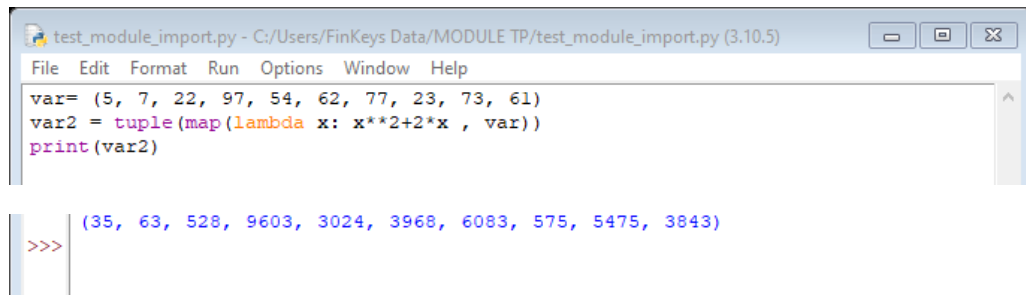
The prompt '>>>' is visible in the console.

Map() Reduce() Filter()

Map() : Définition

- la fonction est utilisée pour définir une expression qui est ensuite appliquée aux "itérables".
- Les fonctions définies par l'utilisateur et les fonctions lambda peuvent toutes deux être envoyées à la fonction map.

Exemple

A screenshot of a Python IDE window titled 'test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)'. The window contains a menu bar (File, Edit, Format, Run, Options, Window, Help) and a code editor. The code in the editor is:

```
var= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)
var2 = tuple(map(lambda x: x**2+2*x , var))
print(var2)
```

Below the code editor, the output of the program is displayed:

```
>>> (35, 63, 528, 9603, 3024, 3968, 6083, 575, 5475, 3843)
```

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
var= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)
var2 = tuple(map(lambda x: x**2+2*x , var))
print(var2)

(35, 63, 528, 9603, 3024, 3968, 6083, 575, 5475, 3843)
>>>
```

Map() Reduce() Filter()

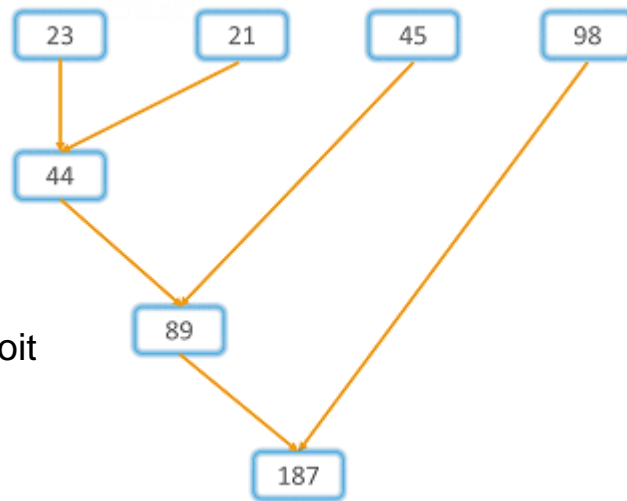
Reduce () : Définition

- La fonction `reduce()` applique une fonction fournie aux "itérables" et renvoie une seule valeur, comme son nom l'indique (reduce : réduire)

Syntaxe

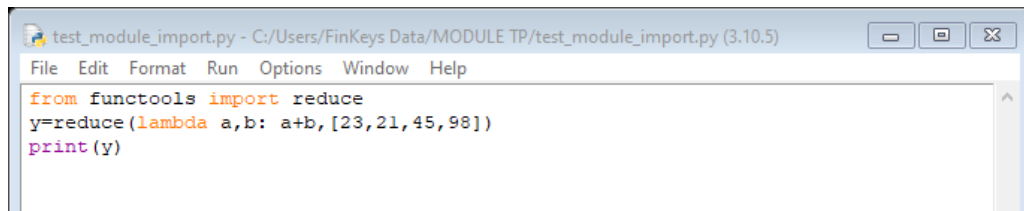
`reduce(function, iterables)`

La fonction indique quelle expression doit être appliquée aux "itérables" dans ce cas. Le module *Function Tools* doit être utilisé pour importer *cette fonction*.

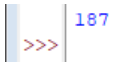


Map() Reduce() Filter()

reduce () : Exemple



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
from functools import reduce
y=reduce(lambda a,b: a+b, [23,21,45,98])
print(y)
```



```
>>>
```

187

Map() Reduce() Filter()

Filter () : Définition

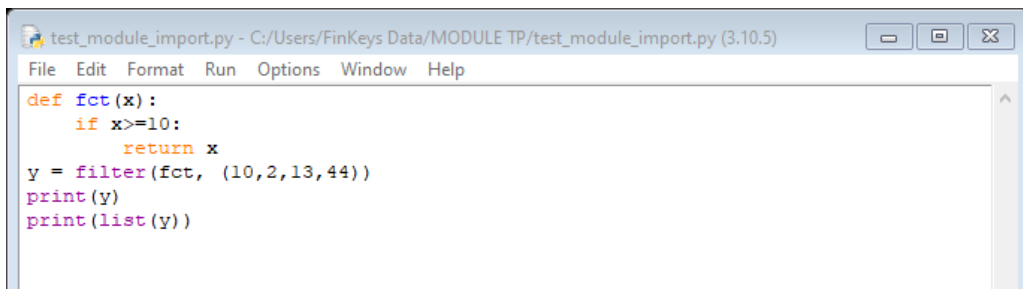
- La fonction filter() est utilisée pour générer une liste de valeurs de sortie qui renvoie vrai lorsque la fonction est appelée.
- Cette fonction, comme map(), peut prendre comme paramètres des fonctions définies par l'utilisateur et des fonctions lambda.

Syntaxe

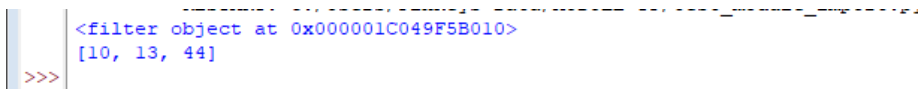
filter (fonction, itérables)

Map() Reduce() Filter()

Filter () : Exemple



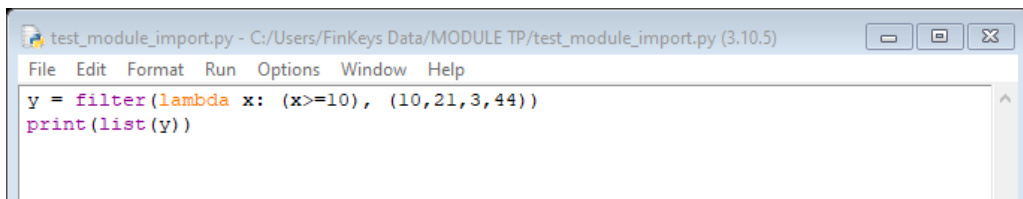
```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
def fct(x):
    if x>=10:
        return x
y = filter(fct, (10,2,13,44))
print(y)
print(list(y))
```



```
<filter object at 0x000001C049F5B010>
[10, 13, 44]
>>>
```

Map() Reduce() Filter()

Filter () : Exemple



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
y = filter(lambda x: (x>=10), (10,21,3,44))
print(list(y))
```



```
>>> [10, 21, 44]
```



Générateurs & Itérateurs

Itérateurs

Définition

- L'**itérateur** en python est un objet qui est **utilisé pour itérer sur des objets itérables** comme **les listes, les tuples, les dicts et les ensembles**.
- L'objet itérateur **est initialisé** à l'aide de la méthode **iter()**. Il utilise la méthode **next()** **pour l'itération**.

`__iter(iterable)` : appelée pour l'initialisation d'un itérateur. Elle renvoie un objet itérateur.

`next (__next__ en Python 3)` : renvoie la valeur suivante pour l'itérable. Lorsque nous utilisons une boucle for pour parcourir un objet itérable, elle utilise en interne la méthode **`iter()`** pour obtenir un objet itérateur qui utilise ensuite la méthode **`next()`** pour l'itérer. Cette méthode lève un ***StopIteration*** pour signaler la fin de l'itération.

Itérateurs

Définition

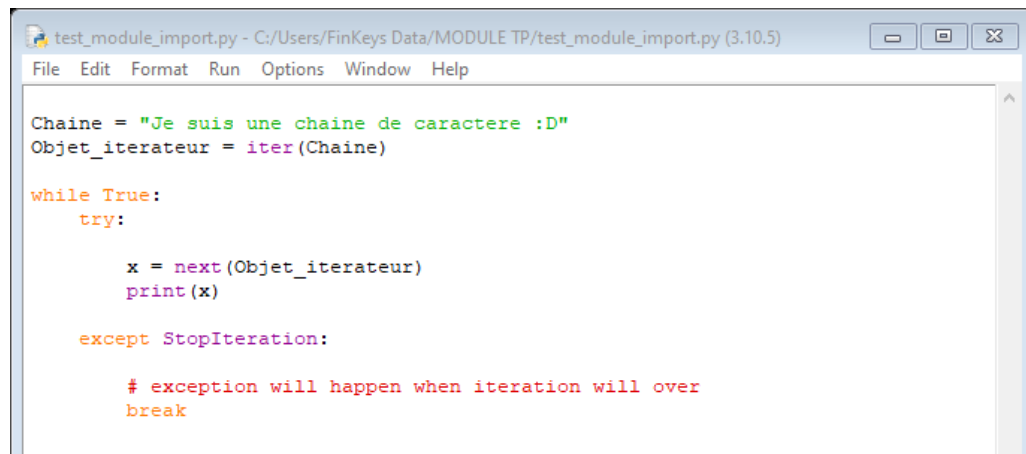
- **L'itérateur** en python est un objet qui est **utilisé pour itérer sur des objets itérables** comme **les listes, les tuples, les dicts et les ensembles**.
- L'objet itérateur **est initialisé** à l'aide de la méthode **iter()**. Il utilise la méthode **next()** **pour l'itération**.

`__iter(iterable)` : appelée pour l'initialisation d'un itérateur. Elle renvoie un objet itérateur.

`next (__next__ en Python 3)` : renvoie la valeur suivante pour l'itérable. Lorsque nous utilisons une boucle for pour parcourir un objet itérable, elle utilise en interne la méthode **`iter()`** pour obtenir un objet itérateur qui utilise ensuite la méthode **`next()`** pour l'itérer. Cette méthode lève un ***StopIteration*** pour signaler la fin de l'itération.

Itérateurs

Exemple



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

Chaine = "Je suis une chaine de caractere :D"
Objet_iterateur = iter(Chaine)

while True:
    try:
        x = next(Objet_iterateur)
        print(x)
    except StopIteration:
        # exception will happen when iteration will over
        break
```

Itérateurs

Exemple

```
>>> ===== RESTART: C:/Users/FinKeys Data/MODULE TP/test_module_import.py =====
J
e
s
u
i
s
u
n
e
c
h
a
i
n
e
d
e
c
a
r
a
c
t
e
r
e
:
D
>>>
```

Ln: 63 Col: 0

Itérateurs

Exemple

```
10
11
12
13
14
15
*****
>>>
```

2

La boucle for utilise en interne (nous ne pouvons pas le voir) un objet itérateur pour parcourir les itérables.

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

class Test:

    # Constructeur
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.x = 10
        return self

    def __next__(self):

        x = self.x

        if x > self.limit:
            raise StopIteration

        self.x = x + 1;
        return x

for i in Test(15):
    print(i)

print('*****')

for i in Test(5):
    print(i)
```

1

Générateurs

Yield

- L'instruction **yield** suspend l'exécution de la fonction et renvoie une valeur à l'appelant, mais **conserve** suffisamment d'état pour permettre à la fonction de **reprendre là où elle s'est arrêtée**. Lorsqu'elle est reprise, la fonction continue **son exécution** immédiatement après la dernière **instruction yield**.
- Cela permet à son code de produire une **série de valeurs au fil du temps**, plutôt que de **les calculer en une fois** et de les renvoyer comme une liste.

Générateurs

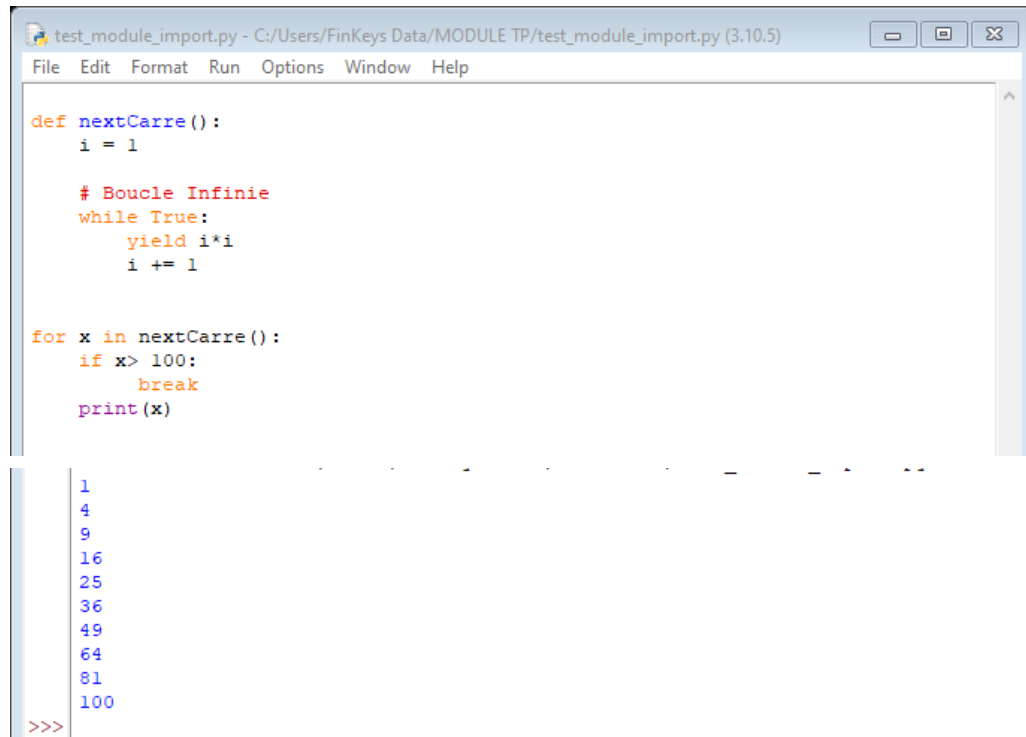
Yield

Return renvoie une valeur spécifiée à son appelant alors que *Yield* peut produire une séquence de valeurs.

Il faut utiliser *yield* lorsque nous voulons *itérer sur une séquence*, mais que nous ne voulons pas stocker la séquence entière en mémoire. Elle est utilisée dans les générateurs Python.

Générateurs

Yield



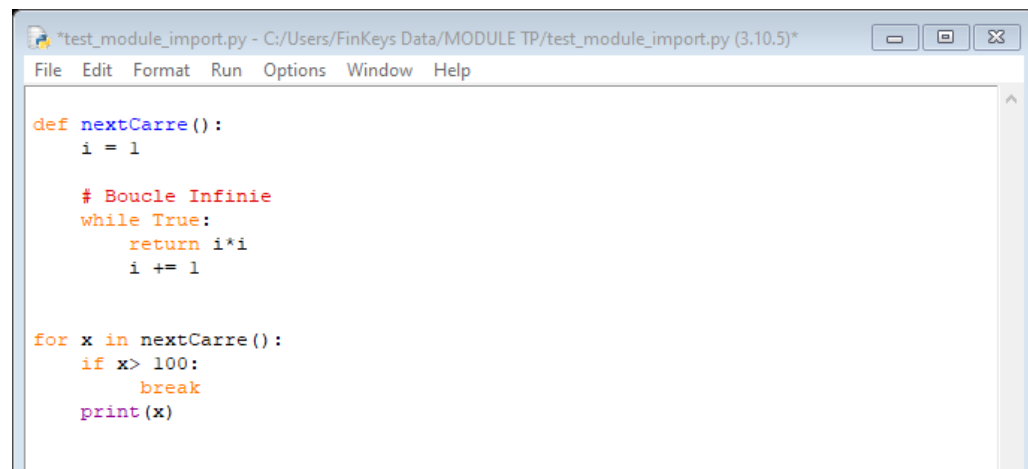
The screenshot shows a Python IDE window titled "test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python code:

```
def nextCarre():  
    i = 1  
  
    # Boucle Infinie  
    while True:  
        yield i*i  
        i += 1  
  
for x in nextCarre():  
    if x > 100:  
        break  
    print(x)
```

Below the code editor, a list of values is displayed, representing the output of the generator function. The values are 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100. The prompt ">>>" is visible at the bottom left of the output area.

Générateurs

Yield

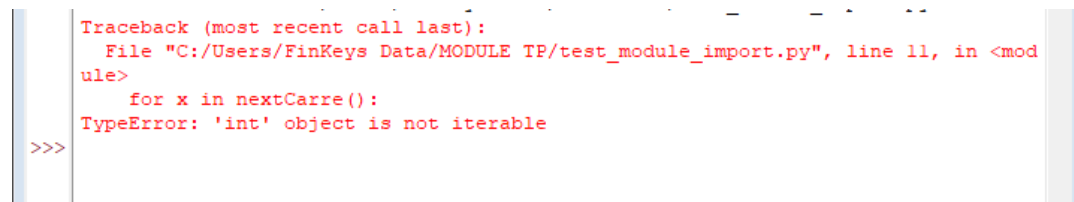


```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)*
File Edit Format Run Options Window Help

def nextCarre():
    i = 1

    # Boucle Infinie
    while True:
        return i*i
        i += 1

for x in nextCarre():
    if x > 100:
        break
    print(x)
```



```
Traceback (most recent call last):
  File "C:/Users/FinKeys Data/MODULE TP/test_module_import.py", line 11, in <module>
    for x in nextCarre():
TypeError: 'int' object is not iterable

>>>
```

Générateurs

Fonction génératrice

Fonction génératrice : Une fonction génératrice est définie comme une fonction normale, mais lorsqu'elle **doit générer une valeur**, elle le fait avec le mot-clé **yield** plutôt que **return**. Si le corps d'une def contient yield, la fonction devient automatiquement **une fonction génératrice**.

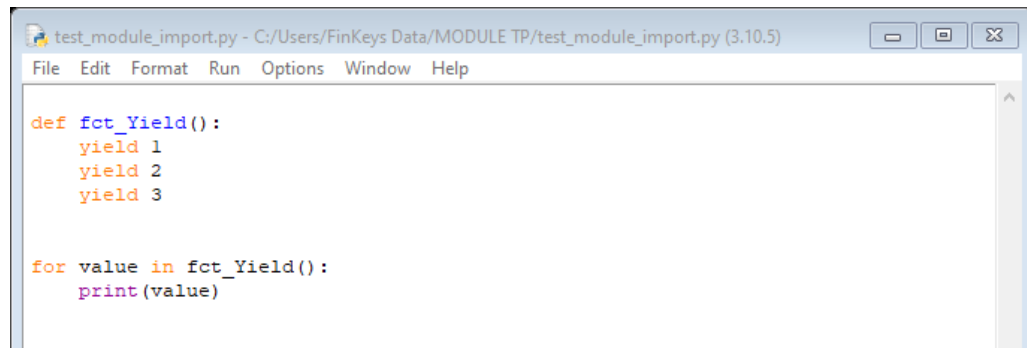
Object générateur

Les fonctions de générateur renvoient un objet générateur.

Les objets générateurs sont utilisés soit en appelant la méthode next sur l'objet générateur, soit en utilisant l'objet générateur dans une boucle "for in"

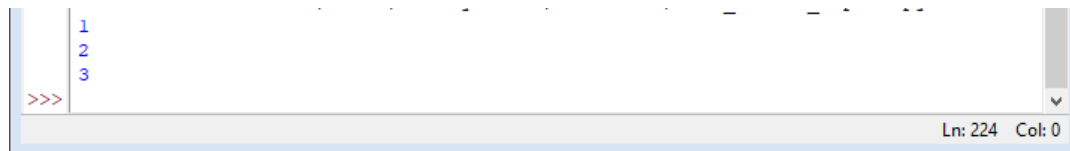
Générateurs

Yield



A screenshot of a Python IDE window titled "test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code editor contains the following Python code:

```
def fct_Yield():  
    yield 1  
    yield 2  
    yield 3  
  
for value in fct_Yield():  
    print(value)
```

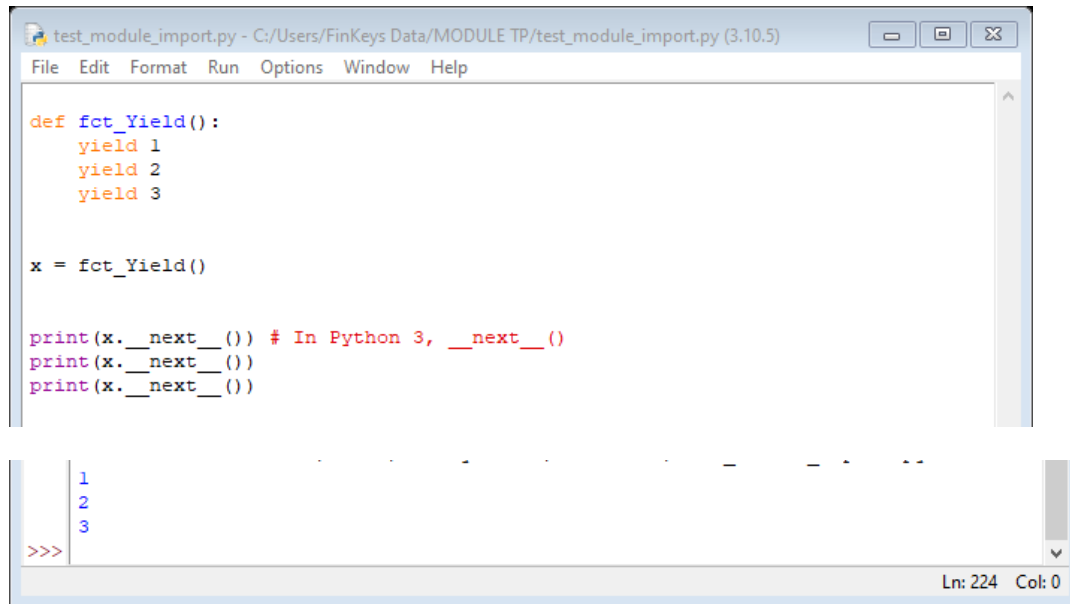


A screenshot of a Python REPL window. The prompt is ">>>". The output shows the values 1, 2, and 3 on separate lines. The status bar at the bottom right indicates "Ln: 224 Col: 0".

```
>>>  
1  
2  
3
```

Générateurs

Fonction génératrice



The screenshot shows a Python IDE window titled 'test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python code:

```
def fct_Yield():  
    yield 1  
    yield 2  
    yield 3  
  
x = fct_Yield()  
  
print(x.__next__()) # In Python 3, __next__()   
print(x.__next__())  
print(x.__next__())
```

Below the code editor is a console window showing the output of the execution:

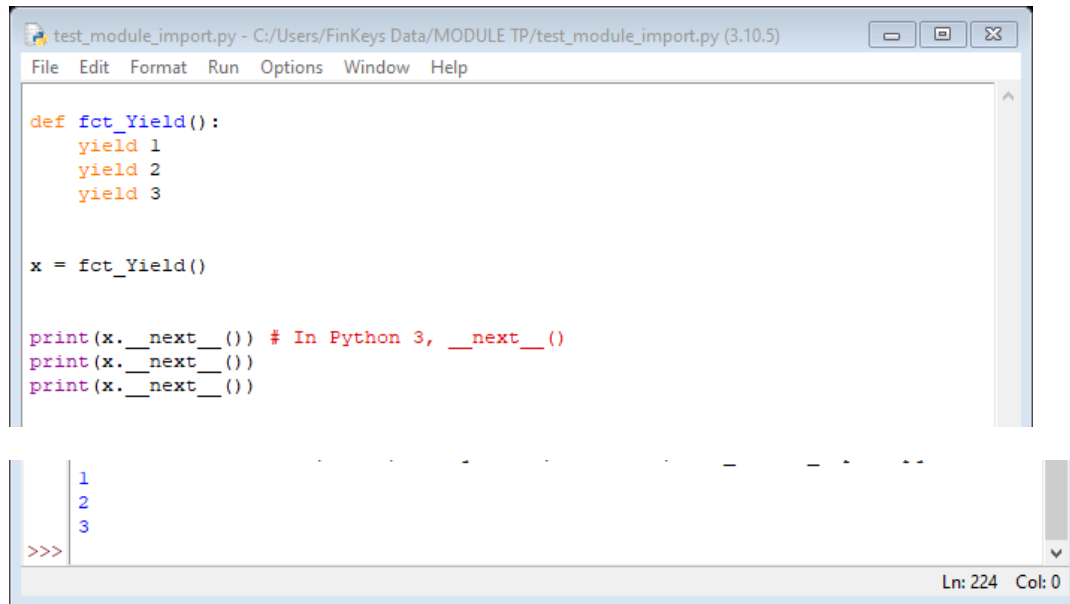
```
>>> 1  
2  
3
```

The status bar at the bottom right indicates 'Ln: 224 Col: 0'.

Une fonction de générateur renvoie donc un objet générateur qui est itérable, c'est-à-dire qu'il peut être utilisé comme un itérateur.

Générateurs

Fonction génératrice



The screenshot shows a Python IDE window titled 'test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python code:

```
def fct_Yield():  
    yield 1  
    yield 2  
    yield 3  
  
x = fct_Yield()  
  
print(x.__next__()) # In Python 3, __next__()   
print(x.__next__())  
print(x.__next__())
```

Below the code editor is a console window showing the output of the execution:

```
>>> 1  
2  
3
```

The status bar at the bottom right indicates 'Ln: 224 Col: 0'.

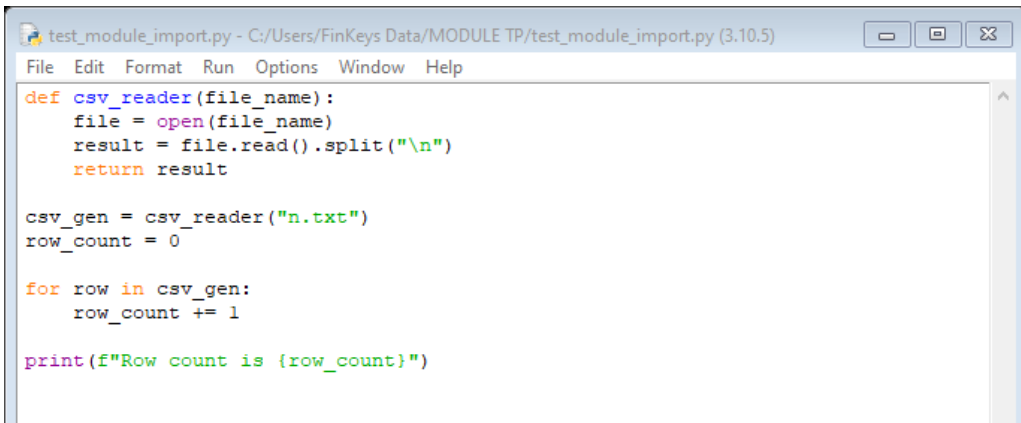
Une fonction de générateur renvoie donc un objet générateur qui est itérable, c'est-à-dire qu'il peut être utilisé comme un itérateur.

Générateurs

Lecture des fichiers larges

Les **générateurs** sont souvent utilisés pour travailler avec **des flux de données ou des fichiers volumineux, comme les fichiers CSV.**

Compter le nombre de lignes dans un fichier CSV



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help
def csv_reader(file_name):
    file = open(file_name)
    result = file.read().split("\n")
    return result

csv_gen = csv_reader("n.txt")
row_count = 0

for row in csv_gen:
    row_count += 1

print(f"Row count is {row_count}")
```

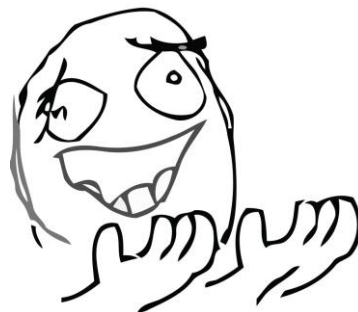
csv_reader() ouvre un fichier et charge son contenu dans csv_gen.
Ensuite, le programme itère sur la liste et incrémente row_count pour chaque ligne.

Générateurs

Lecture des fichiers larges

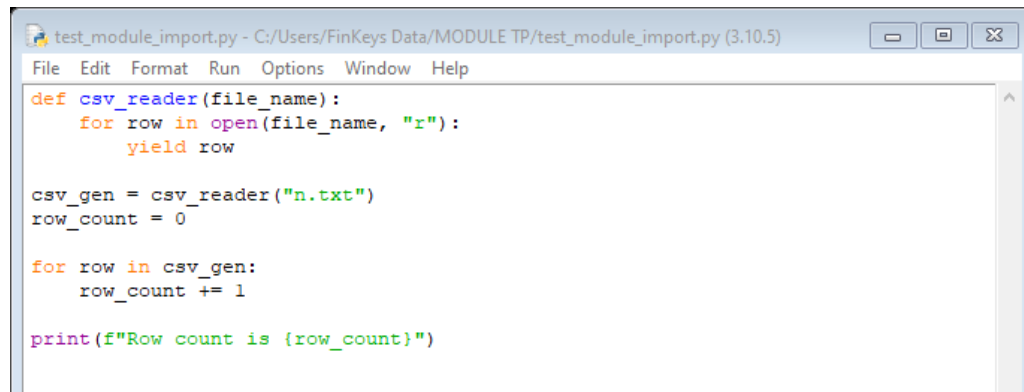
Si le fichier est très grand ? Que se passe-t-il si le fichier est plus grand que la mémoire dont on dispose ?

```
Traceback (most recent call last):
  File "test_module_import.py", line 22, in <module>
    main()
  File "test_module_import.py", line 13, in main
    csv_gen = csv_reader("n.txt")
  File "test_module_import.py.py", line 6, in csv_reader
    result = file.read().split("\n")
MemoryError
```



Générateurs

Lecture des fichiers larges



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row

csv_gen = csv_reader("n.txt")
row_count = 0

for row in csv_gen:
    row_count += 1

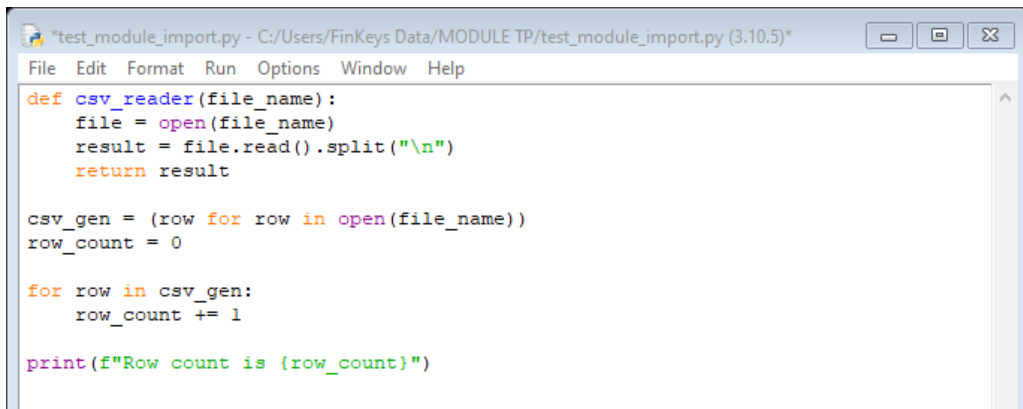
print(f"Row count is {row_count}")
```

Good

Générateurs

Lecture des fichiers larges

Nous pouvons également définir une expression de générateur (également appelée compréhension de générateur). De cette façon, Nous pouvons utiliser le générateur sans appeler une fonction



```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)*
File Edit Format Run Options Window Help

def csv_reader(file_name):
    file = open(file_name)
    result = file.read().split("\n")
    return result

csv_gen = (row for row in open(file_name))
row_count = 0

for row in csv_gen:
    row_count += 1

print(f"Row count is {row_count}")
```