Cours python par l'exemple

Table de matière

Installation de python

Installation de python

Télécharger la dernier version de python, actuellement pour le 15/11/2023 c'est **python-3.12.0-amd64** application d'extension exe,

Note : il faut impérativement cocher le bouton box en bas de la fenêtre d'installation afin d'ajouter au **path** (ajouter le chemin du répertoire à la variable d'environnement Windows).

Installation de pip

- Exécuter l'invite de commande en tant qu'administrateur
- Avant l'installation, il faudrait déterminer la version de python.
- C:\Windows\system32>pip install
  Dans le cas ou la version du package de pip antérieure à la version actuelle de python, il va falloir la mettre à jour en exécutons cette commande :
- C:\Windows\system32>python .exe –m pip install –upgrade pip
  Installation de la bibliothèque « **selenium** »
- C:\Windows\system32>pip install selenium

Installation IDE de PyCharm 2023.3.2(voir la fin du cours)

Comment installer Python sur Windows [Pycharm IDE] (guru99.com)

Les entrées/sorties (en Python)

Les entrées

Fonction d'entrée input()

La fonction input permet la saisie des données à travers une console.

Syntaxe :

input*(prompt*)

Valeurs des paramètres

Parameter	Description
prompt	Une chaîne, représentant un message par défaut avant la saisie

Exemple

IDE Sell

```
>>>x = input('Donner une valeur à x :')
>>>print('La valeur de x= ' + x)
```

Si on veut convertir la chaîne de caractères en un entier, on pourra alors utiliser la fonction **int().**

Exemples d'utilisation

IDE Sell

```
/*Lecture d'une chaîne*/
>>> mot1=input()
Jijel
/* impression*/
>>> print(mot1)
Jijel
```

```
>>> mot2=input("Entrez une ville: ")
Entrez une ville: Constantine
>>> print(mot2)
Constantine
```

```
>>> nombre1=int(input("Entrez un nombre : "))
Entrez un nombre : 12
>>> nombre1+13
25
```

Une manière moins précise consiste à laisser Python choisir le type avec la fonction **eval().**

La fonction eval()

La fonction **eval()** analyse l'argument de l'expression et l'évalue ensuite comme une expression python.

Syntaxe :

eval (expression, [globals()[, locals()]])

Exemple

IDE Sell

```
/*Déclaration de variable*/
>>> x=7
>>>eval("x+5")
```

Résultat

12

Les arguments de la fonction eval()

Les arguments de la fonction **eval()** sont une chaîne et des arguments **optionnels** de **globals** et **locals**.

Globals: il doit s'agir d'un dictionnaire.

IDE Sell

```
x = 10
 # Expression à évaluer
expression = "x + 14"
 # Utiliser eval pour évaluer l'expression
resultat = eval(expression, globals(), None)
# Afficher le résultat
print(resultat)
```

Résulat

24

Note : Dans cet exemple, globals() fournit le contexte global où x est défini, et None est utilisé pour le contexte local, ce qui signifie qu'il n'y a pas de variables locales supplémentaires à considérer.

Locals : peuvent être n'importe quel objet cartographique.

Exemple

IDE Sell

```
from os import cpu_count
>>>eval("[1,cpu_count()]")
```

Résulat

[1,8]

Remarque :

Étant donné que cette fonction évalue chaque chaîne en tant que code, il faut être très prudent lors de son utilisation. Quelqu'un peut l'utiliser pour exécuter du code sur l'ordinateur comme une faille de sécurité.

Par exemple, l'**eval (input())** demandera à l'utilisateur de saisir la chaîne et de l'exécuter en tant que code. Si le module os est importé, on peut effectuer toutes sortes d'actions sur l'appareil, comme la suppression de fichiers et la falsification du système.

Bonnes pratiques lors de l'utilisation de la fonction eval()

- Limiter l'utilisation de eval() aux cas où c'est vraiment nécessaire.
- Valider et filtrer les entrées utilisateur pour éviter les attaques d'injection de code.
- Utiliser des alternatives plus sûres lorsque cela est possible, comme **ast.literal_eval()** pour évaluer des expressions littérales.

Sécurité

La fonction eval() peut présenter des risques de sécurité si elle est utilisée avec des entrées utilisateur non vérifiées. Les attaques d'injection de code sont possibles si des chaînes malveillantes sont évaluées sans validation adéquate.

On peut voir la **Configuration dynamique** , **Interprétation de formules dynamiques** et **Construction dynamique de fonctions** das la suite du cours.

Les sorties

Fonction de sortie print()
La fonction "**print()**" en Python est utilisée pour imprimer un message spécifié à l'écran. La commande d'impression en Python imprime des chaînes ou des objets qui sont convertis en chaîne lors de l'impression sur un écran.

Syntaxe :

print*(object(s)*, sep=*separator*, end=*end*, file=*file*, flush=*flush*)

Valeurs des paramères

Paramètres	Description
object(s)	C'est un objet quelconque qui sera converti en chaîne avant impression
sep='séparateur'	Facultatif, qui permet de spécifié comment séparer les objets. La valeur par défaut est ' '
end='fin'	Facultatif. Qui spécifié ce qu'il faut imprimer à la fin. La valeur par défaut est '\n' (sut de ligne)
File(fichier)	Facultatif. Permet par une méthode d'écriture, d'afficher l'objet. La valeur par défaut est sys.stdout
flush	Facultatif. C'est un booléen, spécifiant si la sortie est vidée (True) ou mise en mémoire tampon (False). La valeur par défaut est False

Exemple

IDE Sell

```
/* Imprimons 5 lignes vierges */
>>>print(5*"\n ")
```

Résultat
On obtient comme résultat 5 lignes vierges

```
>>> print("\n\n\n\n\n ")
```

Résultat
On obtient comme résultat 5 lignes vierges→

```
>>> print('Sortie→', 45, [-3,0,5,10])
```

Résultat

```
Sortie→45 [-3,0,5,10]
>>> print(1+13)
14
>>>x=1+13
>>>print(x)
14
```

IDE Sell

```
/* Afficher plusieurs  valeurs */
>>>print(5,-7,3,4)
```

Résultat

```
5-734
```

```
>>>x=20.5
>>> print('x=', x)
```

Résultat

```
X=20.5
```

```
>>>liste = [1, 2, 3, 4, 5]
>>>print(liste[0], *liste[1:-1], liste[-1])
# Affichera 1 2 3 4 5
# Explications :
# liste[0] = 1 (le premier élément de la liste)
# liste[1:-1] = [2, 3, 4] (les éléments en partant du 1er non-inclus, jusqu'au dernier non-inclus)
# L'opérateur * permet de déstructurer la liste [2, 3, 4] en 3 nombres 2, 3 et 4
# liste[-1] = 5 (le dernier élément de la liste)
```

Commande de fin d'impression

Par défaut, la fonction d'impression en Python se termine par une nouvelle ligne. Cette fonction est livrée avec un paramètre appelé « fin ». La valeur par défaut de ce paramètre est '\n', c'est-à-dire le caractère de nouvelle ligne. Vous pouvez terminer une instruction d'impression avec n'importe quel caractère ou chaîne en utilisant ce paramètre. Ceci est disponible uniquement en Python 3+

Exemple

Edit

```
print ("Bienvenue", end = ' ')
print ("à JIJEL code, postale 18000", end = '!')
```

Le résultat est affiché sur ID Shell

IDE Sell

```
Bienvenue à JIJEL, code postale 180000!
```

Edit

```
n = input("Choisissez un nombre, n'importe quel nombre : ")
print (" Saviez-vous que " , str(n) ," au carré est : "  , str(n*n))
```

Le résultat est affiché sur ID Shell

IDE Sell

```
Choisissez un nombre, n'importe quel nombre : 5
Saviez-vous que 5 au carré est : 25
```

Le paramètre 'sep' pour la fonction 'print' en Python

Le séparateur par défaut entre les arguments pour la fonction '**print**' en Python est un espace : **' '**. On peut passer une valeur au paramètre '**sep**' à la fonction '**print()**' pour remplacer cet espace par n'importe quelle autre chaîne de caractères en Python 3.

Voici quelques exemples d'utilisation du paramètre sep pour la fonction print en Python.

IDE Sell

```
# Cas de base sans l'utilisation du paramètre sep :
print(1, 2, 3)
```

Résultat

```
1 2 3
```

```
# Utiliser des virgules pour séparer des variables :
print(1, 2, 3, sep=',')
/* impression*/
 1,2,3

# Enlever l'espace mis par défaut :
print(1, 2, 3, sep='')
/* impression*/
123
```

IDE Sell

```
>>>x=20.5
>>> print('x=', x)
/* impression*/
x=20.5
```

```
>>>liste = [1, 2, 3, 4, 5]
>>>print(liste[0], *liste[1:-1], liste[-1])
# Affichera 1 2 3 4 5
# Explications :
# liste[0] = 1 (le premier élément de la liste)
# liste[1:-1] = [2, 3, 4] (les éléments en partant du 1er non-inclus, jusqu'au dernier non-inclus)
# L'opérateur * permet de déstructurer la liste [2, 3, 4] en 3 nombres 2, 3 et 4
# liste[-1] = 5 (le dernier élément de la liste)
```

Encore une fois, il est plus clair d'écrire cette conversion en utilisant l'instruction "**repr()**".

A noter que l'instruction "**print**" provoque un saut de ligne (comportement par défaut). Pour écrire une suite de chaînes de caractères, on pourra utiliser la concaténation de chaînes en utilisant l'opérateur "+".

IDE Sell

```
>>> mot1="L'Algérie"
>>> print(mot1)
/* Impression*/
Algérie
```

```
>>> mot2="pays"
>>> print(mot2)
/* Impression*/
Pays
```

```
>>> print(mot1 + " est  un "+ mot2+" Africain. " )
# Impression
L'Algérie est un pays Africain
```

```
>>> mot2=input("Entrez un pays : ")
Entrez un pays : Palistine
>>> print(mot2)
#Impression
Palistine
```

```
>>> nombre1=int(input("Entrez un nombre : "))
/* Impression*/
20
```

```
>>> nombre2=nombre1+13
>>> print(nombre2)
/* Impression*/
25
```

```
>>> print("----------------------")
```

```
>>mot2= " demain"
>>print("{} de {}".format(mot1,mot2) )
# Impression
L'Algérie et de demain
```

Formatage de sortie

La méthode format() formate la ou les valeurs spécifiées et les insère dans l'espace réservé de la chaîne.

L'espace réservé est défini à l'aide d'accolades : {}. En savoir plus sur les espaces réservés dans la section Espace réservé ci-dessous.

La méthode format() renvoie la chaîne formatée

Syntaxe :

string.format(*value1, value2...*)

Valeurs des paramètres

Paramètre	Déscription
value1, value2...	Requis. Une ou plusieurs valeurs qui doivent être formatées et insérées dans la chaîne. Les valeurs sont soit une liste de valeurs séparées par des virgules, une liste clé=valeur ou une combinaison des deux. Les valeurs peuvent être de n'importe quel type de données.

Les espaces réservés

Les espaces réservés peuvent être identifiés à l'aide d'index nommés {price}, d'index numérotés {0}ou même d'espaces réservés vides {}.

"({0},{1},{2}) an".format(self.x, self.y, self.z)

Exemple

Utilisation de différentes valeurs d'espace réservé :

IDE Sell

```
>>> txt1 = "Mon nom est {fname},  j'ai {age} an".format(fname = "Omar", age = 36)
>>> print(txt1)
```

Résultat

Mon nom est Omar, j'ai 36

```
>>>txt2 = "Mon nom est {0}, j'ai {1} an".format("Omar",36)
```

Résultat

Mon nom est Omar, j'ai 36 an

```
>>>txt3 = "Mon nom est {}, j'ai {} an".format("Omar",36)
```

Résultat

Mon nom est Omar, j'ai 36 an

IDE Sell

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1 #Chaque ligne contient un nombre, son carré et son cube
 2   4    8  #{0:2d} : La valeur de x, formatée pour occuper au moins 2 caractères.
 3   9   27 #{1:3d} : Le carré de x (x * x), formaté pour occuper au moins 3 caractères.
 4  16   64 #{2:4d} : Le cube de x (x * x * x), formaté pour occuper au moins 4 caractères.
 5  25  125 #Le format permet de substituer les valeurs de x, x * x et x * x * x dans la chaîne
 6  36  216 #de caractères, en respectant les largeurs spécifiées (2, 3 et 4 caractères
 7  49  343  « respectivement).
 8  64  512
 9  81  729
10 100 1000
```

(Notez que dans ce premier exemple, un espace entre chaque colonne a été ajouté par la façon dont fonctionne print(), qui ajoute toujours des espaces entre ses paramètres.)

Types de formatage

À l'intérieur des espaces réservés, vous pouvez ajouter un type de formatage pour formater le résultat :

:<	Alignement à gauche du résultat (dans l'espace disponible)
:>	Alignement à droite du résultat (dans l'espace disponible)
:^	Alignement centré du résultat (dans l'espace disponible)
:=	Place le panneau à l'extrême gauche
:+	Utilisez le signe plus pour indiquer si le résultat est positif ou négatif
:-	Utilisez un signe moins pour les valeurs négatives uniquement
:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Utiliser un trait de soulignement comme séparateur de milliers
:b	Format binaire
:c	Convertit la valeur en caractère Unicode correspondant
:d	Format décimal
:e	Format scientifique, avec un e minuscule
:E	Format scientifique, avec un E majuscule
:f	Format du numéro de point fixe
:F	Format du numéro de point fixe, au format majuscule (afficher **inf** et **nan** comme **INF** et **NAN**)
:g	Format générale
:G	Format général (utilisant un E majuscule pour les notations scientifiques)
:o	Format octal
:x	Format hexadécimal, minuscule
:X	Format hexadécimal, majuscules
:n	Format pour le nombre
:%	Le format pourcentage

L’utilisation de base de la méthode str.format() ressemble à ceci :

IDE Sell

```
>>> print('Nous sommes les {} qui disent "{}!"'.format('Chevaliers', 'Non'))
```
Résultat
Nous sommes les chevaliers qui disent "Non!"

Les accolades et les caractères à l’intérieur (appelés les champs de formatage) sont remplacés par les objets passés en paramètres à la méthode str.format(). Un nombre entre accolades se réfère à la position de l’objet passé à la méthode str.format().

IDE Sell

```
>>> print('{0} et {1}'.format('spam', 'Oeufs'))
```
Résultat
spam et Oeufs
```
>>> print('{1} et {0}'.format('spam', 'Oeufs'))
```
Résultat
Oeufs et spam

Si des arguments nommés sont utilisés dans la méthode str.format(), leurs valeurs sont utilisées en se basant sur le nom des arguments :

IDE Sell

```
>>> print('Cette {nourriture} est { adjective }'.format(
...       nourriture ='spam', adjective='absolument horrible'))
```
Résultat
Ce spam est absolument horrible.

Les arguments positionnés et nommés peuvent être combinés arbitrairement :

IDE Sell

```
>>> print('L histoire de {0}, {1}, et {autre}.'.format('Omar', 'Malek', autre='Driss'))
```
Résultat
L'histoire d’Oma, Malek et Driss.

Note :
'!a' (appliquer ascii()), '!s' (appliquer str()) et '!r' (appliquer repr()) peuvent être utilisées pour convertir les valeurs avant leur formatage :

IDE Sell

```
>>> contenu = 'anguilles'
>>> print(' Mon aéroglisseur est plein d {}.'.format(contenu))
```

Résultat

Mon aéroglisseur est plein d'anguilles.

```
>>> print('My hovercraft is full of {!r}.'.format(contenu))
```

Résultat

Mon aéroglisseur est plein d'anguilles.

Un caractère ':' suivi d'une spécification de formatage peuvent suivre le nom du champ. Ceci offre un niveau de contrôle plus fin sur la façon dont les valeurs sont formatées. L'exemple suivant arrondit Pi à trois chiffres après la virgule (NdT : qui, en notation anglo-saxonne, est un point).

IDE Sell

```
>>> import math
>>> print('La valeur de PI est d\'environ {0:.3f}.'.format(math.pi))
```

Résultat

La valeur de PI est d'environ 3,142.

Indiquer un entier après le ':' indique la largeur minimale de ce champ en nombre de caractères. C'est utile pour faire de jolis tableaux :

IDE Sell

```
>>> table = {'Omar': '0612325612', 'Malek': ' 0772325612', 'Ali': '0555325612'}
>>> for nom, telephone in table.items():
   print('{0:10} ==> {1:10}'.format(nom, telephone))

Omar       ==>       0612325612
Malek    ==>       0772325612
Ali        ==>       0555325612
```

Si vous avez une chaîne de formatage vraiment longue que vous ne voulez pas découper, il est possible de référencer les variables à formater par leur nom plutôt que par leur position. Utilisez simplement un dictionnaire et la notation entre crochets '[]' pour accéder aux clés :

IDE Sell

```
>>> table = {'Omar': '0612325612', 'Malek': ' 0772325612', 'Ali': '0555325612'}
>>> print('Malek: {0[Malek]:s}; Omar: {0[Omar]:s}; '
     'Ali: {0[Ali]:s}'.format(table))
```

Résultat

Malek: 0772325612; Omar: 0612325612; Ali: 0555325612

Vous pouvez obtenir le même résultat en passant le tableau comme des arguments nommés en utilisant la notation **

IDE Sell

```
>>> table = {'Omar': '0612325612', 'Malek': ' 0772325612', 'Ali': '0555325612'}
>>> print('Malek: {Malek:s}; Omar: {Omar:s}; Ali: {Ali:s}'.format(**table))
```

Résultat

Malek: 0772325612; Omar: 0612325612; Ali: 0555325612

C'est particulièrement utile en combinaison avec la fonction native vars() qui renvoie un dictionnaire contenant toutes les variables locales.

Pour avoir une description complète du formatage des chaînes de caractères avec la méthode str.format(), lisez : Syntaxe de formatage de chaîne.

Mais une question demeure, bien sûr : comment convertir des valeurs en chaînes de caractères ? Heureusement, Python fournit plusieurs moyens de convertir n'importe quelle valeur en chaîne :

Les fonctions repr() et str().

La fonction str() : est destinée à représenter les valeurs sous une forme lisible.

Syntaxe :

str(*object*, encoding=*encoding*, errors=*errors*)

Valeurs des paramètres

Paramètre	**Description**
object	Spécifie l'objet à convertir en chaîne
encoding	L'encodage de l'objet. La valeur par défaut est UTF-8
errors	Si le décodage échoue erreurs nous spécifie comme résoudre le problème

La fonction repr() :est destinée à générer des représentations qui puissent être lues par l'interpréteur (ou qui lèvera une Syntax Error s'il n'existe aucune syntaxe équivalente).

Syntaxe :

repr(*object*)

Valeurs des paramètres

Paramètre	**Description**
object	L'objet dont la représentation imprimable doit être renvoyée

IDE Sell

```
#Quel est la signification de spam ?
#Un spam désigne toute communication non sollicitée envoyée en masse.

>>> x =32.5 ; y =40000
>>> # L'argument de repr() peut être n'importe quel objet Python :
>>> repr((x, y, ('spam', 'Oeufs')))
```

Résultat

```
"(32.5, 40000, ('spam', 'Oeufs'))"
```

```
>>> for x in range(1, 11):
    print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
    # Notez l'utilisation de "end" sur la ligne précédente
    print(repr(x*x*x).rjust(4))
```

Résultat

```
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Il existe une autre méthode, **str.zfill(),** qui comble une chaîne numérique à gauche avec des zéros. Elle comprend les signes plus et moins :

IDE Sell

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Anciennes méthodes de formatage de chaînes

L'opérateur % peut aussi être utilisé pour formater des chaînes. Il interprète l'argument de gauche pratiquement comme une chaîne de formatage de la fonction printf() à appliquer à l'argument de droite, et il renvoie la chaîne résultant de cette opération de formatage.

IDE Sell

```
>>> import math
>>> print(' La valeur de PI est d\'environ %5.3f.' % math.pi)
```

Résultat

```
La valeur de PI est d'environ 3.142.
```

IDE Sell

```
>>>x=5
>>>print('%d'%x)
```
Résultat
```
5
>>>carct='x'
>>>print('%c'%carct)
```
Résultat
```
x
>>>mot='Mustapha'
>>>print('%s'%mot)
```
Résultat
```
Mustapha
```

Les opérateurs Python

Python dispose de nombreux opérateurs qui peuvent être classés selon les catégories suivantes :

- Les opérateurs arithmétiques ;
- Les opérateurs de chaines ;
- Les opérateurs d'affectation ou d'assignation ;
- Les opérateurs de comparaison ;
- Les opérateurs logiques ;
- Les opérateurs d'identité ;
- Les opérateurs d'appartenance;
- Les opérateurs binaires.

Les opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés pour effectuer des opérations mathématiques.

Python reconnait et accepte les opérateurs arithmétiques suivants :

Opérateur	Déscription
+	Addition
–	Soustraction
*	Multiplication
/	Division
%	Modulo
**	Puissance
//	Division entière

Remarque

Le modulo '%' correspond au reste d'une division Euclidienne (division entière) tandis que l'opérateur // permet d'obtenir le résultat entier d'une division (ou la partie entière de ce résultat pour être tout à fait exact.

IDE Sell

#La division
>>>13/4
Résultat
3.25

#partie entière du résultat d'une division
>>>13//4
Résultat
3

#Reste d'une division entière
>>>13%4
Résultat
1

#elévatation de 2 à la puissance 4 (2*2*2*2) :
>>>2**4
Résultat
16

Les opérateurs de chaines

Les opérateurs de chaines vont nous permettre de manipuler des données de type `str` (chaines de caractères) et par extension des variables stockant des données de ce type.
Python met à notre disposition deux opérateurs de chaine : l'opérateur de concaténation `+` et l'opérateur de répétition `*`.

IDE Sell

#Concaténation de deux variables
>>>Prenom="Moad"
>>>Nom="Benkinouar"
>>>print(Nom+" " +Prenom)
Résultat
Benkinouar Moad

#Affichage 4 fois Moad
>>>4*Prenom
Résultat
'MoadMoadMoadMoad'

#Affichage 4 fois Moad séparé entre eux par un espace
>>>4*(Prenom + " ")
Résultat
'Moad Moad Moad Moad'

Les opérateurs d'affectation ou d'assignation

Python reconnait également des opérateurs d'affectation qu'on appelle "composés" et qui vont nous permettre d'effectuer deux opérations à la suite : une première opération de calcul suivie immédiatement d'une opération d'affectation.

Ces opérateurs vont donc nous permettre de réduire la taille de notre code en nous offrant une écriture simplifiée. Voici la liste des opérateurs d'affectation supportés par Python et leur équivalent en "version longue" :

Opérateur	Exemple	Equivalent à	Description
=	**x = 1**	**x = 1**	Affecte 1 à la variable x
+=	**x += 1**	**x = x + 1**	Ajoute 1 à la dernière valeur connue de x et affecte la nouvelle valeur (l'ancienne + 1) à x
-=	**x -= 1**	**x = x – 1**	Enlève 1 à la dernière valeur connue de x et affecte la nouvelle valeur à x
*=	**x *= 2**	**x = x * 2**	Mutliplie par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
/=	**x /= 2**	**x = x / 2**	Divise par 2 la dernière valeur connue de x et affecte la nouvelle valeur à x
%=	**x %= 2**	**x = x % 2**	Calcule le reste de la division entière de x par 2 et affecte ce reste à x
//=	**x //= 2**	**x = x // 2**	Calcule le résultat entier de la division de x par 2 et affecte ce résultat à x
**=	**x **= 4**	**x = x ** 4**	Elève x à la puissance 4 et affecte la nouvelle valeur dans x

Vous pouvez également déjà noter qu'il existe également les opérateurs d'affectation combinés &=, !=, ^=, >>= et <<= qui vont permettre d'effectuer des opérations dont nous discuterons plus tard et d'affecter le résultat de ces opérations directement à une variable.

IDE Sell

```
>>>x=5
>>>y=3
#Equivalence entre x=x+1  et x=y
>>>x+=y
>>>x
```

Résultat

8

```
#Equivalence entre x=x/yet x/=y
>>>x/=y
>>>x
```

Résultat

4

Les opérateurs de comparaison

Opérateur	Opérateur en Python	Description
=	==	Égal à
≠	!= (ou <>)	Différent de
>	>	Supérieur à
≥	>=	& Supérieur ou égal à
<	<	Inférieur à
≤	<=	Inférieur ou égal à
∈	In	Dans
∉	not in	Exclu

Les opérateurs booléens

Les expressions avec un opérateur booléen sont évaluées à « true » ou « false »

Opérateur	Description
X or Y : Ou booléen	Si X est évalué à True, alors : L'expression est True et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
X and Y : Et booléen	Si X est évalué à False, alors : L'expression est False et Y n'est pas évalué. Sinon, l'expression est évaluée à la valeur booléenne de Y.
not X : NON booléen	Evalué à la valeur booléenne opposée de X.

IDE Sell

```
>>>not "      # Chaîne vide
```

Résultat

true

```
>>>not 'abc'   # Chaîne non vide
```

Résultat

False

```
>>>not 'Famse' #chaîne  non vide
```

Résultat

False

```
>>>not 'True' #chaîne non vide
```

Résultat

False

IDE Sell

```
>>>0 or 1
```

Résultat

1 #true

```
>>>0 and 1
```

Résultat

0 #false

On peut également effectuer la comparaison entre deux chaînes de caractères. La comparaison s'effectue en fonction de l'ordre lexicographique :

IDE Sell

```
>>> m_1 = "mange"
>>> m_2 = "manger"
>>> m_3 = "boire"
>>> print(m_1 < m_2) # mange avant manger
```

Résultat

true

```
>>> print(m_3 > m_1) # boire avant de manger
```

Résultat

false

Lorsque l'on compare deux listes entre-elles, Python fonctionne pas à pas. Regardons à travers un exemple comment cette comparaison est effectuée.

Chainer les comparateurs

Il est également possible de chainer les comparateurs:

IDE Sell

```
>>> a,b,c=1, 10, 100
>>> a< b <c
```

Résultat

true

```
>>> a> b <c
```

Résultat

false

Inclusion et exclusion

Les tests d'inclusions s'effectuent à l'aide de l'opérateur « **in** ».

IDE Sell

```
>>> print(3 in [1, 2, 3])
```

Résultat

true

Pour tester si un élément est exclu d'une liste, d'un n-uplet, dictionnaire, etc., on utilise not in :

IDE Sell

```
>>> print(4 not in [1,2, 3])
#Impression
true
>>> print(4 not in [1,2, 3, 4])
#Impression
```

Avec un dictionnaire

IDE Sell

```
>>> dictionnaire = {"nom": "Rockwell", "prenom": "Criquette"}
>>> "age" not in dictionnaire.keys()
Résultat
true
```

Opérations sur les bits

Les expressions avec un opérateur logique sont évaluées à 0 ou 1 pour chaque bit des opérandes entiers. Pour chaque bit des opérandes X et Y :

X | Y : OU.	Si le bit de X ou le bit de Y est évalué à 1, alors : Le bit résultant sera 1. Sinon, le bit résultant sera 0.
X ^ Y : OU exclusif.	Si le bit de X est différent du bit de Y, alors : Lle bit résultant sera 1. Sinon, le bit résultant sera 0.
X & Y : ET.	Si le bit de X et le bit de Y sont tous les deux évalués à 1, alors : Le bit résultant sera 1. Sinon, le bit résultant sera 0.
~ X : NON logique.	Le bit résultant est la valeur opposée du bit de X.
X << Y	Décalage à gauche
X >> Y	Décalage à droite

Quelques constantes

La librairie math propose quelques constantes :

Quelques constantes intégrées dans Python	
Fonction	**Description**
math.pi	Le nombre Pi (π)
math.e	La constante e
math.tau	La constante τ, égale à 2π
math.inf	L'infini (∞)
-math.inf	Moins l'infini (−∞)
math.nan	Nombre à virgule flotante *not a number*

Variables en python

Les bases sur les variables :

- le nom d'une variable peut commencer par n'importe lettre minuscule ou majuscule ou un '_', puis des lettres, des chiffres.
- une variable ne peut pas être utilisée avant d'avoir été définie (utilisation d'une variable dans une expression avant de lui affecter une valeur produit une erreur).
- par contre, pas besoin de déclarer ou typer explicitement une variable avant de lui affecter une valeur. Il suffit de faire : myVar = 'Bonjour' pour définir myVar.
- Une variable sans valeur est définie par : myVar = None (None est l'équivalent de null dans d'autres langages).

Visibilité des variables

Par défaut, une variable est toujours locale, donc disponible seulement dans la fonction dans laquelle est elle définie.

IDE Sell

```
>>>x=0
def myFunc():
    x = 1
myFunc()
  print(x)
```

Résultat

0

Exécution

Imprime 0, car la variable x définie dans la fonction est différente, mais donc qu'on sort de la fonction, c'est la variable en dehors de la fonction qui est disponible.

- Une variable définie en dehors de toutes les fonctions est globale.
- On peut accéder à l'intérieur d'une fonction à une variable globale, mais par défaut, on ne peut pas la modifier (lecture seule).
- Par contre, si on déclare la variable dans la fonction avec global, on peut la modifier :

IDE Sell

```
>>>x=7
def myFunc():
   global x
    print(x)
   x += 1
myFunc()
  print(x)
```

Résultat

Imprime 7, puis 8

- et par contre, on peut bien sûr toujours modifier le contenu d'une variable si celui-ci est mutable :

IDE Sell

```
>>>x=['a']
def myFunc():
    x.append('b')
myFunc()
  print(x)
```

Résultat

Donne ['a', 'b'] et pas besoin d'instruction global

Identifiant d'une variable

- id(myVar) : donne l'id interne de la structure de données pointée par la variable (c'est l'id décimal que l'on peut voir aussi en hexadécimal quand la variable est un objet et qu'on l'imprime).
- cet id est unique et constant pendant toute la durée de vie de la structure (mais un id peut être réutilisé dès que cette structure a disparu).

IDE Sell

```
>>> a = 'Bonjour'; b = a; print(id(a), "  ",id(b))
#Impression
2309365064656    2309365064656
```

Imprime deux fois le même id car les variables a et b pointent vers la même structure de données.

Les underscores '_'

- un seul underscore au début : élément privé qui n'est pas importé lors d'un **from myModule import *.**
- un seul underscore à la fin : pour éviter les conflits avec un mot clef réservé.
- deux underscores au début : le nom est changé en le préfixant avec _nomClasse.
- deux underscores au début et à la fin : noms pour fonctions spéciales comme **__init__,** ne pas en définir des nouveaux.

Types des variables

Une variable sans valeur est définie par : **myVar = None** (None est l'équivalent de **null** dans d'autres langages).

Valeurs qui sont considérées comme fausses : False, None, 0, '', [], (), {}

Types primitifs

- **bool :** booléen (**True ou False**)
- **int :** entier.
- **float :** nombre flottant qui a la précision d'un double.
- **str :** chaîne de caractère (string).

Pour avoir l'entier de taille maximum : **sys.maxsize**

```
import sys
print("Pour avoir l'entier de taille maximum : {0}".format(sys.maxsize))
#Impression
Pour avoir l'entier de taille maximum : 9223372036854775807
```

Conversion de n'importe quoi en string : avec `` ou repr() :

```
x = 10
s= 'valeur ={}'.format(x)
```

Utilisation f-strings –Python 3.6+)

```
s = f'valeur = {x}'
s = 'valeur = ' + repr(x)
```

Conversion d'une string en int ou float

- x = int('2')

Déclenche une **ValueError** si ce n'est pas le bon type.

Conversions :

float(3) : convertit l'int en float.
int(3.0) : convertit le float en int.
int(3.6) : convertit le float en int, en donnant ici 3.

Examen du type d'une variable

- **type(var) :** renvoie le type de la variable, par exemple <type 'int'> ou <type 'list'>
- pour tester le type d'une variable, on peut faire : type(var) == list (ou str ou int ou float)
- mais pour tester le type d'une variable, le mieux est isinstance(var, list).
- insistance donne True si on teste si un objet contre sa classe, mais aussi contre ses classes de base.

Détruire une variable

del x

Dimension d'un objet occcupée en mémoire

- sys.getsizeof(Objet)
- ca s'appelle Objet.__sizeof__ en rajoutant éventuellement un overhead du au garbage collector.
- attention, ca ne compte que la mémoire utilisée par l'objet, pas celle utilisée par les objets qui sont référencés dedans.

Pour avoir le nombre de références qui pointe vers une variable :

- **sys.getrefcount(['abc'])** : renvoie 1
- **x = ['abc']; print(sys.getrefcount(x)) :** renvoie 2, car il y a aussi la référence reçue par l'appel de la fonction elle-même.
- avec les constantes (notamment entier ou chaînes), on a souvent beaucoup de références car python stocke des références internes vers certaines valeurs.

Les instructions conditionnelles

Les instructions conditionnelles if

L'instruction conditionnelle la plus simple que l'on peut rencontrer est if. Si et seulement si une expression est évaluée à True, alors une instruction sera évaluée.

Syntaxe :

```
if expression :
    Instructions
```

Les lignes après les deux points (:) doivent être placées dans un bloc, en utilisant un taquet de tabulation

Editeur

```
>>>x=2
if  x = =2 :
   Print("Bonjour")
```

Résultat : sur IDE shell

IDE Sell

Résultat

Bonjour

Editeur

```
>>>x=2
if  x = =2 :
   y="Bonjour"
    print(y + " , vaut : " + str(x))
```

Résultat : sur IDE shell

IDE Sell

Résultat

Bonjour, x vaut : 2

Les instructions conditionnelles if-else

Si la condition n'est pas vérifiée, on peut proposer des instructions à effectuer, à l'aide des instructions if-else.

Syntaxe :

```
if expression :
    Instructions 1
else :
    Instructions 2
```

Editeur

```
temperature = 26
chaleur = ""
if temperature > 28 :
    chaleur = "chaud"
else :
    chaleur = "froid"
print("Il fait " + chaleur)
```

Résultat : sur IDE shell

IDE Sell

```
#Impression
Il fait froid
```

Si la température est à présent de 32 degrés C :

Editeur

```
temperature = int(input(‘ Donner un valu pour la température T= ‘))
chaleur = ""
if temperature > 28 :
    chaleur = "chaud"
else :
    chaleur = "froid"
print("Il fait " + chaleur)
```

Résultat : sur IDE shell

IDE Sell

```
Résultat
Il fait chaud
```

Exemple :

On peut utiliser une **condition IF & ELSE** pour affecter une valeur à la variable b en fonction de la valeur de la variable a.

Editeur

```
a = 10
b=10
b=1 if a<20 else 2
print(b)
```

Résultat : sur IDE shell

IDE Sell

Résultat

```
1
```

Exemple :

Choisir quelle variable Python va afficher en fonction de la valeur de ces variables.

Editeur

```
a = 0
b= 5
print( a if a!=0 else b
```

Résultat : sur IDE shell

IDE Sell

Résultat

```
5
```

Les instructions conditionnelles if-elif

Si la condition n'est pas vérifiée, on peut en tester une autre et alors évaluer d'autres instructions si cette seconde est vérifiée. Sinon, on peut en tester encore une autre, et ainsi de suite. On peut aussi proposer des instructions si aucune des conditions n'a été évaluée à True. Pour ce faire, on peut utiliser des instructions conditionnelles if-elif.

Syntaxe

```
if expression:
    instructions
elif expression_2:
    instructions_2
elif expression_3:
    instructions_3
else:
    Autres_instruction
```

L'exemple précédent manque un peu de sens commun. Peut-on dire que lorsqu'il fait 28 degrés C ou moins il fait froid ? Ajoutons quelques nuances :

Editeur

```
temperature = int(input('donner une tempérture'))
chaleur = ""
if temperature > 28:
   chaleur = "chaude"
elif temperature <= 28 and temperature > 15:
   chaleur = "tempérée"
elif temperature <= 15 and temperature > 0:
   chaleur = "froide"
else:
   chaleur = "très froide"
print("La température est " + chaleur)
```

Résultat : sur IDE shell

IDE Sell

Résultat

La température est très froide

Remarque :

L'avantage d'utiliser des instructions conditionnelles if-elif par rapport à écrire plusieurs instructions conditionnelles if à la suite est qu'avec la première manière de faire, les comparaisons s'arrêtent dès qu'une est remplie, ce qui est plus efficace.

Editeur

```
a = 115
b = 115
if b > a:
   print("b est plus grand que a")
elif a == b:
   print("a et b sont égaux")

print("La température est " + chaleur)
```

Exemple

Un exemple d'usage de elif

Imaginons un concert de rock pour lequel, l'organisateur a établi une fourchette de tarifs. Le prix des places dépend de l'âge du spectateur. Les jeunes et très jeunes, mais aussi les personnes proches de l'âge de la retraite bénéficient de tarifs avantageux. Les autres doivent débourser le prix fort.

Nous pourrions avoir un bloc de code tel que celui-ci:

Editeur

```
age=70
if age < 10:
    prix_de_la_place = 8
elif age < 18:
    prix_de_la_place = 20
elif age > 60:
    prix_de_la_place = 15
else:
    prix_de_la_place = 35
```

Editeur

```
from random import randint

value = randint(0, 9)
print(f'Pour information, value = {value}")
if value == 0:
    print("Zéro")
elif value == 1:
    print("Un")
elif value == 2:
    print("Deux")
elif value == 3:
    print("Trois")
elif value == 4:
    print("Quatre")
elif value == 5:
    print("Cinq")
elif value == 6:
    print("Six")
elif value == 7:
    print("Sept")
elif value == 8:
    print("Huit")
elif value == 9:
    print("Neuf")
else:
    print("Ce n'est pas un chiffre mais un nombre")
```

if et and

Le mot-clé et est un opérateur logique, et est utilisé pour combiner des instructions conditionnelles. Tester si a est supérieur à b, ET si c est supérieur à a :

Editeur

```
if a > b and c > a:
   print("Les 2 conditions sont vraies")
```

if et or

Le ou mot-clé est un opérateur logique, et est utilisé pour combiner des instructions conditionnelles. Vérifier si a est supérieur à b, OU si a est supérieur à c :

Editeur

```
if a > b or a > c:
   print("Au moins une condition est Vraie")
```

Les instructions match-case

Les instructions match-case sont arrivées avec la version 3.10 de Python et PEP 634. C'est l'équivalent du switch-case des autres langages de programmation. Cette instruction de "Filtrage par motif structurel" permet de vérifier qu'une variable correspond à une des valeurs définies**.**

match-case se présente comme ça en Python :

Syntaxe :

```
match valeur:
  case pattern_1:
    expression_1
  case pattern_2:
    expression_2
  case pattern_3:
    expression_3
       …
       ...
       ...
  case _:
    expression_par_defaut
```

Editeur

```
def jour_de_la_semaine(jour):
    match jour:
        case 1:
            return 'Lundi'
        case 2:
            return 'Mardi'
        case 3:
            return 'Mercredi'
        case 4:
            return 'Jeudi'
        case 5:
            return 'Vendredi'
        case 6:
            return 'Samedi'
        case 7:
            return 'Dimanche'
        case _:
            return 'Pas un jour de la semaine'
```

Résultat : sur IDE shell

IDE Sell

```
Jour = 3
print(jour_de_la_semaine(Jour)
```

Résultat

Mercredi

Comment vérifier le code statut d'une requête en Python ?

On peut facilement vérifier le statut d'une réponse HTTP avec un match case en Python.

Prenons l'exemple ci-dessous où on a un code réponse (response_code) qu'on test contre plusieurs valeurs.

Ce code nous permettra de renvoyer de meilleurs messages à nos utilisateurs.

Editeur

```
response_code = 200
match response_code:
  case 200:
    print("Tout s'est bien passé ")
  case 300 | 301 | 302:
    print("Vous allez être redirigé ")
  case 400 | 404:
    print("Cette page ne semble pas exister ")
  case 500:
    print("Oops, quelque chose s'est mal passé ")
```

Notre requête simple qui s'est bien passée avec un code 200 affichera Tout s'est bien passé sur la sortie standard.

Cet exemple est très simple, il faudrait ajouter une meilleure gestion d'erreur avec plus de code d'erreur et plus de personnalisation.

Notez aussi l'utilisation des opérateurs | quand on veut passer plusieurs valeurs contre lesquelles tester notre variable !

Comment vérifier l'intégrité d'une structure en Python ?

Il est possible de vérifier la structure d'un objet en Python avec l'instruction switch case (match-case) en Python.

Prenons une liste de n éléments, on va print une phrase différente en fonction de la structure de cette liste.

Editeur

```
liste = ['un', 'deux', 'trois', 'quatre', 'cinq', 'six']

match liste:
  case [a]:
    print(f'Une seule valeur : {a}')
  case [a, b]:
    print(f'Deux valeurs : {a} et {b}')
  case [a, b, c]:
    print(f'Trois valeurs : {a}, {b} et {c}')
  case [a, b, c, d]:
    print(f'Quatre valeurs : {a}, {b}, {c} et {d}')
  case [a, b, c, d, *reste]:
    print(f'Quatre valeurs {a}, {b}, {c}, {d} mais aussi {", ".join(reste)}')
```

Qui affichera la chaîne de caractères Quatre valeurs un, deux, trois, quatre mais aussi cinq, six sur la sortie standard.

Comment gérer les commandes en argument en Python ?

On peut aussi utiliser l'opérateur pipe (|) pour tester un ou plusieurs éléments de la liste.
Imaginons un script qui prend des commandes pour ajouter ou soustraire un nombre à un nombre total.
En Python, on peut utiliser les valeurs passé en argument depuis la console avec la variable sys.argv.
sys.argv[0] est le nom du fichier donc on commence à sys.argv[1].

Editeur

```
import sys

match sys.argv[1:]:

  case ['ajouter', valeur_a_ajouter, valeur_de_depart]:
    print(f'On a ajouté {valeur_a_ajouter} à {valeur_de_depart}, ce qui donne
{int(valeur_de_depart) + int(valeur_a_ajouter)}')

  case ['soustraire', valeur_a_soustraire, valeur_de_depart]:
    print(f'On a soustrait {valeur_a_soustraire} à {valeur_de_depart}, ce qui donne
{int(valeur_de_depart) - int(valeur_a_soustraire)}')

  case ['aide']:
    print('Ajouter un message d\'aide qui explique à l\'utilisateur comment utiliser le script')
```

Si on lance le script python comme ça python NOM_DU_SCRIPT.py ajouter 5 4, on aura On a ajouté 5 à 4, ce qui donne 9 en sortie standard.

Autre facon d'écrire ce programme avecune fonction appeler main:

Editeur

```
import sys

def main():
    match sys.argv[1:]:
        case ['ajouter', valeur_a_ajouter, valeur_de_depart]:
            print(f'On a ajouté {valeur_a_ajouter} à {valeur_de_depart}, ce qui donne
{int(valeur_de_depart) + int(valeur_a_ajouter)}')
        case ['soustraire', valeur_a_soustraire, valeur_de_depart]:
            print(f'On a soustrait {valeur_a_soustraire} de {valeur_de_depart}, ce qui donne
{int(valeur_de_depart) - int(valeur_a_soustraire)}')
        case ['aide']:
            print("Utilisation du script:")
            print("  ajouter <valeur_a_ajouter> <valeur_de_depart> - Ajouter une valeur à une
autre")
            print("  soustraire <valeur_a_soustraire> <valeur_de_depart> - Soustraire une valeur
d'une autre")
        case _:
            print("Erreur: Commande non reconnue. Utilisez 'aide' pour les instructions.")

if __name__ == "__main__":
    main()
```

Résultat : sur IDE shell

IDE Sell

```
python Nmscript.py ajouter 5 10
```

Résultat

```
On a ajouté 5 à 10, ce qui donne 15
```

Ou bien

IDE Sell

```
python Nmscript.py soustraire 3 8
```

Résultat

```
On a soustrait 3 de 8, ce qui donne 5 strait 3 de 8, ce qui donne 5
```

Et pour afficher le message d'aide :

IDE Sell

python Nmscript.py aide

Résultat

Utilisation du script:

ajouter <valeur_a_ajouter> <valeur_de_depart> - Ajouter une valeur à une autre

soustraire <valeur_a_soustraire> <valeur_de_depart> - Soustraire une valeur d'une autre t 3 de 8, ce qui donne 5

Les boucles

La boucle while

Le but de la boucle *while* est de répéter certaines instructions tant qu'une condition est respectée.

Syntaxe :

Syntaxe
instruction normale 1
instruction normale 2
instruction normale 3
...
while condition :
    instruction de la boucle 4
    instruction de la boucle 5
    instruction de la boucle 6
    ...
instruction normale 7
instruction normale 8
instruction normale 9
...

Editeur

```
print("*** Avant la boucle ***")
entree="bonjour"
while entree != " " :
  print("------ dans la boucle ------")
  entree=input("Taper un mot : ")
print("*** Après la boucle ***")
input("Appuyer sur ENTER pour terminer le programme. ")
```

Editeur

```
nb_repetitions = 3

i = 1
while i <= nb_repetitions :
   print("Et "+str(i)+"!")
   i = i+1
print("Zéro!")
input("Appuyer sur ENTER pour terminer le programme. ")
```

L'instruction break et while

L'instruction break sert, non pas à interrompre le programme, mais à sortir de la boucle.

Editeur

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

While imbriqué

Editeur

```
n = 3
while n < 10:
    x = 2
    while x < n:
        if (n % x) == 0:
            print(n, 'égal à', x, '*', n // x)
            break
        x += 1
    else:
        print(n, 'est un nombre premier')
    n += 1
```

Résultat : sur IDE shell

IDE Sell

Résultat

```
2 est un nombre premier
3 est un nombre premier
4 égal à 2 * 2
5 est un nombre premier
6 égal à 3 * 2
7 est un nombre premier
8 égal à 4 * 2
9 égal à 3 * 3
```

Instruction continue

Avec l'instruction continue nous pouvons arrêter l'itération en cours, et continuer avec la suivante.

Passer à l'itération suivante si i est égal à 3 :

Editeur

```
i = 0
while i < 6:
   i += 1
   if i == 3:
      continue
   print(i)
```

L'instruction « pass »

L'instruction « pass » ne fait rien. Son rôle est juste de fournir une instruction lorsque celle-ci est nécessaire pour fournir une syntaxe correcte, mais qu'aucune action n'est utile. Par exemple :

Editeur

```
while True:
   pass                    # Juste pour attendre le ctrl-c
```

Un autre cas d'utilisation du « pass » est de réserver un espace en phase de développement pour une fonction ou un traitement conditionnel, permettant ainsi au développeur de construire son code à un niveau plus abstrait. L'instruction « pass » est alors ignorée silencieusement :

Syntaxe :

```
if expression:
   pass                    # Sera implémenté plus tard
else:
   Autre action
```

L' instruction « do…while »

L'instruction do…while utilisée dans d'autres langages n'existe pas en Python, car sa syntaxe souple et ses outils intégrés permettent de programmer facilement son équivalent.

Équivalent d'un do…while en langage C (action et répéter tant que la condition est vraie)…

Syntaxe :

```
do {
   action();
} while (condition);

while True:
   action()
   if not condition: break
```

La boucle for

Une boucle for est généralement utilisée pour parcourir des séquences (c'est-à-dire une liste, un tuple, un dictionnaire, un ensemble ou une chaîne).

Syntaxe :

```
instruction normale
instruction normale
instruction normale
instruction normale
...
for variable in liste_valeurs :
    instruction de la boucle
    instruction de la boucle
    instruction de la boucle
    ...
instruction normale
instruction normale
instruction normale
...
```

Exemple

Editeur

```
fruits = ["poire", "mangue", "kiwi"]
for x in fruits:
    print(x)
```

Exemple

Editeur

```
for i in range(1,4) :
    print("Et "+str(i)+"!")

print("Zéro!")
input("Appuyer sur ENTER pour terminer le programme. ")
```

Instruction break et for

Avec l'instruction break, nous pouvons arrêter la boucle avant qu'elle n'ait parcouru tous les éléments.

Editeur

```
fruits = ["poire", "mangue", "kiwi"]
for x in fruits:
   print(x)
   if x == "mangue":
      break
```

Instruction continue et for

Avec l'instruction continue nous pouvons arrêter l'itération courante de la boucle, et continuer avec la suivante.

Exemple

Editeur

```
fruits = ["poire", "mangue", "kiwi"]
for x in fruits:
   if x == "mangue":
      continue
   print(x)
```

Fonction range()

Pour parcourir un ensemble un nombre de fois défini, nous pouvons utiliser la fonction range(). Elle retourne une séquence de nombres, à partir de 0 par défaut, et par incréments de 1 (par défaut), et se termine à un nombre spécifié.

Utilisation de la fonction range() :

Exemple

Editeur

```
for x in range(6):
   print(x)
```

Notez que range(6) ne définit pas les valeurs de 1 à 6, mais les valeurs de 0 à 5.

La fonction range() prend par défaut 0 comme valeur de départ, cependant il est possible de spécifier la valeur de départ en ajoutant un paramètre : range(2, 6), qui signifie des valeurs de 2 à 6 (mais pas 6).

Exemple

Editeur

```
for x in range(2,6):
  print(x)
```

La fonction range() incrémente par défaut la séquence de 1, mais il est possible de spécifier la valeur d'incrémentation en ajoutant un troisième paramètre : range(2, 30, 3).

Exemple

Editeur

```
for x in range(2,30,3):
  print(x)
```

Boucle Else in For

L'instruction « else » spécifie un bloc de code à exécuter lorsque la boucle est terminée.

Afficher tous les chiffres de 0 à 5, et afficher un message lorsque la boucle est terminée.

Exemple

Editeur

```
for x in range(6):
  print(x)
else:
  print("Enfin terminé !")
```

Boucles for imbriquées

Une boucle imbriquée est une boucle à l'intérieur d'une boucle.

La "boucle intérieure" sera exécutée une fois pour chaque itération de la "boucle extérieure".

Afficher chaque adjectif pour chaque fruit :

Exemple

Editeur

```
adj = ["croquant", "gros", "petit"]
fruits = ["poire", "mangue", "kiwi"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Exemple

Editeur

```
# Affiche un rectangle rempli de X
for loop in range(5):
  for loop in range(10):
    print("X", end = "")  # pas de retour à la ligne ici
  print()
```

Exemple

- calcul de la somme des dix premiers entiers au carré
- comptage
- conversion d'un vecteur en une matrice
- conversion d'une chaîne de caractère en datetime
- conversion d'une chaîne de caractère en matrice
- conversion d'une matrice en chaîne de caractères
- conversion d'une matrice en un vecteur

Calcul de la somme des dix premiers entiers au carré
Ce calcul simple peut s'écrire de différentes manières.

Editeur

```
s = 0
for i in range(1, 11):
  s += i**2
```

D'une façon abrégée:

Editeur

```
s = sum ( [ i**2 for i in range(1, 11) ] )
```

Comptage
Exemple suivant compte les mots d'une liste de mots.

Editeur

```
li = ["un", "deux", "un", "trois"]
d = {}
for l in li:
  if l not in d:
    d[l] = 1
  else:
    d[l] += 1
print(d)
```

Résultat

IDLE Shell

```
{'un': 2, 'deux': 1, 'trois': 1}
```

La structure la plus appropriée ici est un dictionnaire puisqu'on cherche à associer une valeur à un élément d'une liste qui peut être de tout type. Si la liste contient des éléments de type modifiable comme une liste, il faudrait convertir ceux-ci en un type immuable comme une chaîne de caractères. L'exemple suivant illustre ce cas en comptant les occurrences des lignes d'une matrice.

Editeur

```
mat = [[1, 1, 1], [2, 2, 2], [1, 1, 1]]
d = {}
for l in mat:
    k = str(l)    # k = tuple (l) lorsque cela est possible
    if k not in d:
        d[k] = 1
    else:
        d[k] += 1
print(d)
```

Résultat

IDLE Shell

```
{'[1, 1, 1]': 2, '[2, 2, 2]': 1}
```

Pourquoi les listes ne peuvent pas être des clés de dictionnaire : Why Lists Can't Be Dictionary Keys.

On peut également vouloir non pas compter le nombre d'occurrence mais mémoriser les positions des éléments tous identiques. On doit utiliser un dictionnaire de listes :

Editeur

```
li = ["un", "deux", "un", "trois"]
d = {}
for i, v in enumerate(li):
    if v not in d:
        d[v] = [i]
    else:
        d[v].append(i)
print(d)
```

Résultat

IDLE Shell

```
{'un': [0, 2], 'deux': [1], 'trois': [3]}
```

S'il suffit juste de compter, l'écriture la plus simple est :

Editeur

```
r = {}
li = ["un", "deux", "un", "trois"]
for x in li:
    r[x] = r.get(x, 0) + 1
print(r)
```

Résultat

IDLE Shell

```
{'un': 2, 'deux': 1, 'trois': 1}
```

Conversion d'un vecteur en une matrice

Dans un langage comme le C++, il arrive fréquemment qu'une matrice ne soit pas représentée par une liste de listes mais par une seule liste car cette représentation est plus efficace. Il faut donc convertir un indice en deux indices ligne et colonne. Il faut bien sûr que le nombre de colonnes sur chaque ligne soit constant. Le premier programme convertit une liste de listes en une seule liste.

Editor

```
ncol = 2
vect = [0, 1, 2, 3, 4, 5]
mat = [vect[i * ncol: (i + 1) * ncol] for i in range(0, len(vect) // ncol)]
print(mat)
```

Résultat

IDLE Shell

```
[[0, 1], [2, 3], [4, 5]]
```

Conversion d'une chaîne de caractère en datetime

C'est le genre de fonction qu'on n'utilise pas souvent mais qu'on peine à retrouver lorsqu'on en a besoin. Il faut utiliser la fonction strftime.

Editeur

```
import datetime
dt = datetime.datetime.strptime("07/04/2026", "%d/%m/%Y")
print(dt)
#autre façon pour affichr la date +temps
datetime.datetime(2026, 4, 7, 0, 0)
#Afficher seulement la date
date_only = dt.date()
print(date_only)
#formatag de date
formatted = dt.strftime("%Y-%m-%d")
print(formatted)
```

Conversion d'une chaîne de caractère en matrice

Les quelques lignes qui suivent permettent de décomposer une chaîne de caractères en matrice. Chaque ligne et chaque colonne sont séparées par des séparateurs différents. Ce procédé intervient souvent lorsqu'on récupère des informations depuis un fichier texte lui-même provenant d'un tableur.

Editeur

```
s = "case11;case12;case13|case21;case22;case23"
# décomposition en matrice
ligne = s.split("|")                    # lignes
mat = [l.split(";") for l in ligne]  # colonnes
print(mat)
```

Résultat

IDLE Shell

```
[['case11', 'case12', 'case13'], ['case21', 'case22', 'case23']]
```

Comme cette opération est très fréquente lorsqu'on travaille avec les données, on ne l'implémente plus soi-même. On préfère utiliser un module comme pandas qui est plus robuste et considère plus de cas. Pour écrire, utilise la méthode to_csv, pour lire, la fonction read_csv. On peut également directement enregistrer au format Excel read_excel et écrire dans ce même format to_excel.

Conversion d'une matrice en chaîne de caractères

Editeur

```
mat = [['case11', 'case12', 'case13'], ['case21', 'case22', 'case23']]
ligne = [";".join(l) for l in mat]     # colonnes
s = "|".join(ligne)                    # lignes

print(s)
```

Résultat

IDLE Shell

```
case11;case12;case13|case21;case22;case23
```

Conversion d'une matrice en un vecteur

Dans un langage comme le C++, il arrive fréquemment qu'une matrice ne soit pas représentée par une liste de listes mais par une seule liste car cette représentation est plus efficace. Il faut donc convertir un indice en deux indices ligne et colonne. Il faut bien sûr que le nombre de colonnes sur chaque ligne soit constant. Le premier programme convertit une liste de listes en une seule liste.

Editeur

```
mat = [[0, 1, 2], [3, 4, 5]]
lin = [i * len(mat[i]) + j
      for i in range(0, len(mat))
      for j in range(0, len(mat[i]))]
print(lin)
```

Résultat

IDLE Shell

```
[0, 1, 2, 3, 4, 5]
```

Editeur

```
mat = [[0, 1, 2], [3, 4, 5]]
#1. Avec compréhension de liste (recommandé)
lin = [elem for row in mat for elem in row]
print(lin)
#2. Avec sum (astuce simple)
lin = sum(mat, [])
print(lin)
#3. Avec itertools
import itertools
lin = list(itertools.chain.from_iterable(mat))
print(lin)
```

Vous pouvez aussi utiliser des fonctions telles que reduce.

Editeur

```
from functools import reduce
mat = [[0, 1, 2], [3, 4, 5]]
lin = reduce(lambda x, y: x + y, mat)
print(lin)
```

Résultat

IDLE Shell

```
[0, 1, 2, 3, 4, 5]
```

Types séquentiels — list, tuple, range

Il existe trois types séquentiels basiques: les *lists*, *tuples* et les *range*. D'autres types séquentiels spécifiques au traitement de données binaires et chaînes de caractères sont décrits dans des sections dédiées.

Définition d'une liste (les éléments peuvent être q uelconques) :

Editeur

```
# Déclaration d'une liste vide
myList = []
Ou  bien
myList = list()
```

Editeur

```
# Exemple de liste
myList = [2, 4, 6]
# Une liste peut avoir des éléments de type hétérogènes
myList = ['ab', 2, 'cd', 5]
# Liste contenant des listes.
myList = ['a', ['b', 'c'], 'd']
```

Editeur

```
# Liste calculée à partir d'un autre
myList = [2, 4, 5]; myList2 = [3 * x for x in myList]
#Imprission
[6, 12, 15]
```

```
# Liste calculée à partir d'un autre, mais en prenant certains éléments seulement
myList = [2, 4, 5]; myList3 = [3 * x for x in myList if x > 3]
#Impression
[12, 15]
```

Concaténation entre deux listes
Lorsque l'on utilise le symbole + entre deux listes, Python les concatène en une seule :

Editeur

```
list = [1, "Pomme", 5, 7]
liste = [9, 11]
print(list + liste)
```

Résultat

```
[1, 'Pomme', 5, 7, 9, 11]
```

```
# Idem avec des n-uplets
A= (1, "Pomme", 5, 7)
 B= (9, 11)
print(A + B)
```

Résultat

```
(1, 'Pomme', 5, 7, 9, 11)
```

Multiplication d'une liste par un scalaire n
En "multipliant" une liste par un scalaire n, Python répète n fois cette liste :

Editeur

```
list = [1, "Pomme", 5, 7]
print(3*list)
```

Résultat

```
[1, 'Pomme', 5, 7, 1, 'Pomme', 5, 7, 1, 'Pomme', 5, 7]
```

```
# Idem avec des n-uplets
A= (1, "Pomme", 5, 7)
print(3*A)
```

Résultat

```
(1, 'Pomme', 5, 7, 1, 'Pomme', 5, 7, 1, 'Pomme', 5, 7)
```

Opérations sur les listes

Console

```
>>>myList = ['a', 'b', 'c', 'd']
>>>myList[2]
```

Résultat

'c' #i(indices commencent à 0).

```
# Affichage du dernier élément 'd'
>>>A=myList[-1]
>>>print(3*A)
```

Résultat

'd'

```
#Affichage de 'b' de deux manières différentes
>>>myList[-3]
```

Résultat

'b'

```
>>>myList[1]
```

Résultat

'b'

Sous liste d'une liste

Console

```
# Sous liste commençant à l'indice 0 et terminant à l'indice précédent 2
>>>myList = ['a', 'b', 'c', 'd']
>>> myList[0:2]
```

Résultat

['a', 'b']

```
# Tous les éléments sauf les deux derniers
>>> myList[0:-2]
```

Résultat

['a', 'b']

```
# Sous liste commençant à l'indice 0 et terminant à l'indice précédent 4, et par pas #de 2, donc
>>>myList[0:4:2]
```

Résultat

['a', 'c']

```
>>> myList[-3:len(myList)] # les 3 derniers éléments
```

Résultat

['b', 'c', 'd']

Remplacer un élément par un autre élément

Console

```
>>>myList[1] = 'x'
```
Résultat
```
['a', 'x', 'c', 'd']#ici on écrase la valeur de b
# Insertion d'un élément à la position 1
>>> myList[1:1] = ['x']
```
Résultat
```
['a', 'x', 'b', 'c',  'd']#ici on n'écrase pas la valeur de b
# Insertion des éléments x et y en début de liste
>>> myList[1:1] = 'xy'
```
Résultat
```
['a', 'x', 'y', 'b', 'c',  'd']
#Remplacement d'une sous-liste par une autre liste
>>> myList[0:2] = ['x', 'y']
```
Résultat
```
[ 'x', 'y', 'c',  'd']
#Remplacement  une sous-liste par une liste qui n'a pas la même taille
>>> myList[0:2] = ['x', 'y', 'z']
```
Résultat
```
['x', 'y', 'z', 'c', 'd']
```

Elimination d'un ou d'une partie d'une liste

Console

```
>>>myList = ['a', 'b', 'c', 'd']

#Elimination des deux premiers éléments de la liste
>>> myList[0:2] = []
```
Résultat
```
['c', 'd']
# Détruire l'élément d'indice 2
>>> del myList[2]
```
Résultat
```
['a', 'b', 'd']
# Détruire la slice de 0 à 2 - 1
>>> del myList[0:2]
```
Résultat
```
['c', 'd']
#Détruire toute la liste
>>> del myList
```

Longueur d'une liste

len(myList) : longueur de la liste.

IDLE Shell

```
#Longueur de la liste
>>> len(myList)
```

Inverse d'une liste

IDLE Shell

```
#Inverser une liste
>>> myList[::-1]
Résultat
['d', 'c', 'b', 'a']
# Adressage d'une liste ['a', 'b', 'c', 'd'] hors limites
>>> A=myList[4]
Résultat
Erreur
# Par contre l'écriture suivante
>>> myList[2:5]
Résultat
['c', 'd']
# Renvoie d'une liste vide
>>> myList[4:5]
```

Copie de liste

l2 = l1 : l2 et l1 référencent la même liste.

l2 = l1[:] : l2 est une copie de la liste l1.

l2 = list(l1) : l2 est une copie de la liste l1.

Console

```
l1=['a', 'b', 'c', 'd']
>>> l2 = l1  #: l211 référence la même liste
>>>print(l2)
>>> l2 = l1[:]  # l2 est une copie de la liste l1
>>>print(l2)
>>>l2 = l1(4) # l2 est une copie de la liste l1
>>>print(l2)
```

Opérateur de répétition

Répétition d'une liste : ['a'] * 5 ou [None] * 100

Exemple :

IDLE Shell

```
>>>dicts = [{}] * 3 #Attention, ce n'est pas bon, c'est 3 fois la même référence au  dictionnaire !
#Pareil avec une liste.
>>>dicts = [{} for i in range(3)] # C'est bon, c'est 3 références différente
```

La commande Range

IDLE Shell

```
>>>range(3) : #donne un itérateur sur la liste [0, 1, 2].
>>>print(range(3))
>>>range(2, 7)  #donne un itérateur sur la liste [2, 3, 4, 5, 6].
>>>range(1, 7, 2)  #range un itérateur de 1 à 7 exclus, mais par pas de 2 (sur la liste [1, 3, 5])
>>>range(5, -1, -1) #range de 5 à -1 exclus, mais en décroissant (par pas de -1), donc donne
#un itérateur sur la liste [5, 4, 3, 2, 1, 0], très pratique pour boucler sur une liste dans l'ordre
#décroissant.
>>>range(4, 4) #donne un littérateur sur la liste vide [].
```

Exemple

Création des listes avec la commande range

IDLE Shell

```
>>> list(range(10))
```

Résultat

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
```

Résultat

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
```

Résultat

```
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
```

Résultat

```
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
```

Résultat

```
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
```

Résultat

```
[]
>>> list(range(1, 0))
```

Résultat

```
[]
```

Opération	Résultat
s[i] = x	élément *i* de *s* est remplacé par *x*
s[i:j] = t	tranche de *s* de *i* à *j* est remplacée par le contenu de l'itérable *t*
del s[i:j]	identique à s[i:j] = []
s[i:j:k] = t	les éléments de s[i:j:k] sont remplacés par ceux de *t*
del s[i:j:k]	supprime les éléments de s[i:j:k] de la liste
s.append(x)	ajoute *x* à la fin de la séquence (identique à s[len(s):len(s)] = [x])
s.clear()	supprime tous les éléments de s (identique à del s[:])
s.copy()	crée une copie superficielle de s (identique à s[:])
s.extend(t) or s += t	étend *s* avec le contenu de *t* (proche de s[len(s):len(s)] = t)
s *= n	met à jour *s* avec son contenu répété *n* fois
s.insert(i, x)	insère *x* dans *s* à l'index donné par *i* (identique à s[i:i] = [x])
s.pop([i])	récupère l'élément à *i* et le supprime de *s*
s.remove(x)	supprime le premier élément de *s* pour qui s[i] == x
s.reverse()	inverse sur place les éléments de *s*

Note :
« t » doit avoir la même longueur que la tranche qu'il remplace.

L'argument optionnel **i** vaut **-1** par défaut, afin que, par défaut, le dernier élément soit retiré et renvoyé.

« remove » lève une exception ValueError si **x** ne se trouve pas dans **s**.

La méthode « **reverse() »** modifie les séquence sur place pour économiser de l'espace lors du traitement de grandes séquences. Pour rappeler aux utilisateurs qu'elle a un effet de bord, elle ne renvoie pas la séquence inversée.

« clear() » et « **copy() »** sont inclues pour la compatibilité avec les interfaces des conteneurs muables qui ne supportent pas les opérations de découpage (comme dict et set)

Remarque : Nouveau à partir de la version 3.3: méthodes **clear()** et **copy().**

La valeur n est un entier, ou un objet implémentant « **__index__() »**. Zéro et les valeurs négatives de n permettent d'effacer la séquence. Les éléments dans la séquence ne sont pas copiés ; ils sont référencés plusieurs fois, comme expliqué pour « **s * n »** dans Opérations communes sur les séquences.

Fonctions préétablies sur les listes

Fonction	Description
l.append(x)	Ajoute un élément x
l.extend(l2)	Ajoute la liste l2 à la fin.
l.insert(i, x)	Insert l'élément i à la position x, et décale le reste (comme l[i:i] = x).
l.remove(x)	Enlève le premier élément identique à x (erreur si non trouvé).
l.pop()	Enlève le dernier élément et le renvoie.
l.pop(0)	Enlève le premier élément et le renvoie.
l.pop(i)	Enlève l'élément à la position i et le renvoie.
On peut aussi faire : **sorted(l, key = lambda x: ...)**	(comme avec sort)
l.index(x)	Renvoie l'index du 1er élément identique à x (erreur si non trouvé).
l.count(x)	Compte combien de fois l'élément x est dans la liste.
l.sort()	Trie en place les éléments de la liste.
l.sort(key = lambda x: x[1])	Effectue le tri sur les valeurs calculées en appliquant la fonction donnée chaque élément (exemple de tri case-insensitif : **l.sort(key = str.lower)**). Pour faire du tri sur une clef multiple, on peut utiliser un tuple
l.sort(inverse = True**)**	Effectue le tri dans l'autre sens.
sorted(l)	Renvoie la liste triée, mais sans affecter la liste de départ (sorted peut prendre les mêmes options key et reverse que sort()).
sorted(l, myOrder**)**	**Q**uand on veut utiliser une fonction de tri personnalisée (comme avec sort) **sort** et **sorted** font des tris stables (ordre de clefs identiques est conservé).
l.reverse()	**R**enverse la liste en place.
reversed(l)	**R**envoie un itérateur qui permet de boucler sur les éléments de la liste en sens inverse.

Pour tester si une chaîne est dans une liste : s = 'cd'; s in ['ab', 'cd', 'ef'] : donne True.

Listes de compréhension

IDLE Shell

```
# liste de tous les carrés
>>>[x ** 2 for x in [1, 2, 3, 4]]
```

Résultat :

```
[1, 4, 9, 16]
```

On peut rajouter une condition

IDLE Shell

```
# liste de tous les carrés
>>>print([x ** 2 for x in [1, 2, 3, 4] if x != 2 and x != 3] )
```

Résultat :

```
[1, 16]
```

On peut utiliser 2 boucles, la 2ème boucle étant interne :

IDLE Shell

```
>>> l1 = ['a', 'b']; l2 = [1, 2]
>>>print([(x1, x2) for x1 in l1 for x2 in l2])
```

Résultat :

```
[('a', 1), ('a', 2), ('b', 1), ('b', 2)]
```

Les deux "for" sont dans le même ordre que si c'était 2 boucles imbriquées.

Exemple :

IDLE Shell

```
>>>[(x,y) for x in ('a', 'b', 'c') for y in ('A', 'B')]
```

Résultat :

```
[('a', 'A'), ('a', 'B'), ('b', 'A'), ('b', 'B'), ('c', 'A'), ('c', 'B')]
```

Exemple :

Trouver tous les index d'une liste ayant une certaine valeur : l = ['a', 'b', 'c', 'a'];

IDLE Shell

```
>>> l = ['a', 'b', 'c', 'a']
>>>[i for i in range (len(l)) if l[i] == 'a']
```

Résultat :

```
[0, 3]
```

Concaténation de plusieurs listes en une liste

Exemple :

```
myList = [['a', 'b'], ['c'], [], ['d', 'e', 'f']]
```

1ère possibilité :

IDLE Shell

```
>>> myList = [['a', 'b'], ['c'], [], ['d', 'e', 'f']]
>>> sum(myList, [])
```

Résultat :

```
['a', 'b', 'c', 'd', 'e', 'f']
```

2ième possibilité : [élément pour la sous-liste dans myListe pour l'élément dans la sous-liste] : renvoie aussi ['a', 'b', 'c', 'd', 'e', 'f']).

On peut récupérer les valeurs d'un appel de fonction avec une liste de variables : [x, y, z] = s.split(','), mais il faut être sur d'avoir le bon nombre de valeurs retournées, sinon une exception levée !

-min et max
min(4, 2, 7) : renvoie 2

IDLE Shell

```
>>> min(4, 2, 7)
```
Résultat :
2

Action sur une liste

IDLE Shell

```
>>> min([4, 2, 7])
```
Résultat :
2

Remarque :
On ne peut pas mettre à jour les éléments d'une liste dans une boucle :

Editeur

```
l = [1, 2, 3]
for x in l:
    x += 1
```

N'affecte pas la liste de départ.
Si on veut modifier la liste de départ, il faut faire :

Editeur

```
l = [1, 2, 3]
for in range(len(l)):
    l[i] += 1
```

Remarque :
On peut tester l'égalité de 2 listes l1 == l2 : deux listes sont égales ssi elles ont même nombre d'éléments et tous les éléments sont identiques.

Tri d'une liste selon l'ordre d'une autre liste
On veut trier l2 selon l'ordre de tri de l1, avec l1 = ['a', 'd', 'e', 'b', 'c']; l2 = [1, 4, 5, 2, 3].

Première solution :

IDLE Shell

```
>>> l1 = ['a', 'd', 'e', 'b', 'c']; l2 = [1, 4, 5, 2, 3]
>>> map(lambda x: x[1], sorted(zip(l1, l2), key = lambda x: x[0]))
```

Résultat :

```
[1, 2, 3, 4, 5]
```

Deuxième solution : en utilisant une liste intermédiaire qui donne l'ordre (utile si plusieurs listes à ranger toujours dans le même ordre) :

IDLE Shell

```
>>> listOrder = sorted(range(len(l1)), key = lambda x: l1[x]);
```

Résultat:

```
[1, 2, 3, 4, 5]
>>>[l2[x] for x in listOrder]
```

Résultat:

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

Conversion d'une liste en itérateur

IDLE Shell

```
>>> iter(myList)
```

IDLE Shell

```
>>> next(iter(myList), 'a')
# Renvoie la première valeur de la liste, et si elle est vide, renvoie 'a'
```

Les opérations **in** et **not in** ont les mêmes priorités que les opérations de comparaison. Les opérations + (concaténation) et * (répétition) ont la même priorité que les opérations numériques correspondantes. [3]

Opération	**Résultat**
x in s	True si un élément de *s* est égal à *x*, sinon False
x not in s	False si un élément de *s* est égal à *x*, sinon True
s + t	la concaténation de *s* et *t*

Opération	Résultat
s * n or n * s	équivalent à ajouter *s n* fois à lui même
s[i]	i^e élément de *s* en commençant par 0
s[i:j]	tranche (*slice*) de *s* de *i* à *j*
s[i:j:k]	tranche (*slice*) de *s* de *i* à *j* avec un pas de *k*
len(s)	longueur de *s*
min(s)	plus petit élément de *s*
max(s)	plus grand élément de *s*
s.index(x[, i[, j]])	indice de la première occurrence de *x* dans *s* (à ou après l'indice *i* et avant indice *j*)
s.count(x)	nombre total d'occurrences de *x* dans *s*

Les séquences du même type supportent également la comparaison. En particulier, les *tuples* et les listes sont comparés **lexico_graphiquement** en comparant les éléments correspondants. Cela signifie que pour être égales, chaque élément doit être égal deux à deux et les deux séquences doivent être du même type et de la même longueur

La comparaison entre deux listes

Python va commencer par comparer les premiers éléments de chaque liste (ici, c'est possible, les deux éléments sont comparables ; dans le cas contraire, une erreur serait retournée) :

IDLE Shell

```
>>> x = [1, 3, 5, 7]
>>> y = [9, 11]
>>> print(x < y)
```

Résultat

true

Comme 1<9, Python retourne True.

- Maintenant changeons x pour que le premier élément soit supérieur au premier de y

IDLE Shell

```
>>> x = [10, 3, 5, 7]
>>> y = [9, 11]
>>> print(x < y)
Résultat
false
```

Cette fois, comme 10>910>9, Python retourne False.

- Changeons à présent le premier élément de x pour qu'ils soient égal à celui de y :

IDLE Shell

```
>>> x = [10, 3, 5, 7]
>>> y = [10, 11]
>>> print(x < y)
Résultat
true
```

Cette fois, Python compare le premier élément de x avec celui de y, comme les deux sont identiques, les seconds éléments sont comparés.

- On peut s'en convaincre en évaluant le code suivant :

IDLE Shell

```
>>> x = [10, 12, 5, 7]
>>> y = [10, 11]
>>> print(x < y)
Résultat
true
```

Tuples

Un tuple, c'est comme une liste, sauf que les éléments ne peuvent pas être changés (non mutable)

Exemple

IDLE Shell

```
>>> t = ('a', 'b')
>>> t[0] = 'c'
Résultat
Erreur
```

Intérêt des tuples : ils peuvent être utilisés comme clefs dans les dictionnaires, alors que les listes ne le peuvent pas. Les tuples prennent aussi un peu moins de mémoire que les listes.

Définition d'un tuple :

IDLE Shell

```
>>> t = ('a', 'b',2)
#tuple à 0 élément :
Résultat :
t = ()
#tuple à 1 élément :
Résultat :
t = ('a',) ou t = 'a', #avec la virgule dans les 2 cas !
#tuple à 2 éléments :
Résultat
t = ('a', 'b') ou t = 'a', 'b'
```

On accède au 2ème élément par t[1] (comme pour les listes).

Sequence packing et unpacking

IDLE Shell

```
# Sequence packing
>>> t = x, y, z
# Sequence unpacking
>>> x, y, z = t
Résultat
t doit avoir 3 éléments dans ce cas r
#On peut faire (x, y, z) = (1, 2, 3) pour affecter à x, y et z les valeurs.
```

Unpacking de tuples ou de listes avec des étoiles

IDLE Shell

```
#On fixe
>>> t = ('a', 'b', 'c', 'd')
# mais cela pourrait aussi être une liste ou même une string, donc n'importe quelle séquence
>>> (x, y, z, u) = t # affecte à chaque variable la valeur
Résultat
x = 'a', y = 'b', z = 'c', u = 'd'
>>> (x, y, *z) = t
Résultat :
x = 'a', y = 'b', z =[ 'c',  'd']#ici z est une liste, pas un tuple
# On peut faire l'affectation dans l'autre sens
>>> (*x, y, z) = t
Résultat :
x = ['a', 'b'], y = 'c', z = 'd'
```

On peut aussi faire (x, *y, z) = t. L'important, c'est de ne pas avoir 2 étoiles à la fois, sinon erreur.

Sur une liste de paires (par exemple), on peut faire :
Exemple

Editor

```
myList = [(1, 2), (3, 4), (5, 6)]
for (x, y) in myList:
   print(x)
   print(y)
```

Une fonction peut renvoyer plusieurs valeurs
Il suffit qu'elle renvoie un tuple, par exemple return a, b, on peut alors l'appeler en stockant le résultat directement dans un tuple avec le bon nombre de variables : x, y = myFunc()

Tuples de compréhension
On peut aussi faire :

IDLE Shell

```
>>>(x ** 2 for x in [1, 2, 3, 4])
#Attention, le résultat de cette instruction ne retourne pas Un tuple, ça retourne un générateur !

# Pour avoir un tuple, il suffit de faire
>>> tuple(x ** 2 for x in [1, 2, 3, 4])
```

Résultat :

```
(1, 4, 9, 16)
# On peut aussi faire
>>>tuple([x ** 2 for x in [1, 2, 3, 4])
```

Unpacking en python3 pour les retours de fonction
si une fonction f renvoie (1, 2, 3) :
*x, = f() donne x = [1, 2, 3]
x, *y = f() donne x = 1 et y = [2, 3]
*x, y = f() donne x = [1, 2] et y = 3

Les Séquences en général
En python, une séquence est : un string (chaîne), une list (liste) ou un tuple.

-in, not in : opérateurs de condition pour savoir si une valeur est présente dans une séquence.
Exemple

IDLE Shell

```
>>>s = 'ACGANGAC'
>>>'N' in s
Résultat :
True
>>>l = [1, 2, 4]
>>> 3 in l
Résultat :
False
```

is, is not : pour savoir si 2 objets mutables (liste) sont réellement les mêmes.

Conversion des séquences
Exemple

IDLE Shell

```
# Convertit une chaîne en liste
>>> list('abc')
Résultat :
['a', 'b', 'c']
# Convertit une chaîne en tuple
Résultat :
('a', 'b', 'c')
# Convertit un tuple en liste
>>> list(('a', 'b', 'c'))
Résultat :
['a', 'b', 'c']
# Convertit une liste en tuple
>>> tuple(['a', 'b', 'c'])
Résultat :
('a', 'b', 'c')
```

Affectation : on peut faire les choses suivantes, du moment que les séquences ont le même nombre d'éléments :
(x, y, z) = (1, 2, 3)
[x, y, z] = [1, 2, 3]
Mais aussi (x, y, z) = [1, 2, 3] et [x, y, z] = (1, 2, 3) et même : (x, y, z) = 'abc' ou [x, y, z] = 'abc'

Convertion d'une liste de tuples en dictionnaire

Exemple

IDLE Shell

```
>>>dict([('a', 2), ('b', 3)])
```

Résultat :

```
{'a': 2, 'b': 3}
```

Fonctions sur les séquences :

filter(myFunc, mySeq)	renvoie les éléments x de la sequence mySeq pour lesquels myFunc(x) est vraie. Le type renvoyé est le même que mySeq
map(myFunc, mySeq)	renvoie la liste myFunc(x) ou x est element de mySeq
map(myFunc, l1, l2)	renvoie la liste des myFunc(x1, x2) ou x1 est dans l1 et x2 est dans l2 pour chaque position (valeur passée à la fin est None si une des listes est plus petite).
zip(['a', 'b', 'c'], ['A', 'B', 'C'])	renvoie une liste de tuples, avec un tuple pour les premiers éléments, puis un tuple pour les seconds ... Ici, [('a', 'A'), ('b', 'B'), ('c', 'C')]. Ca marche aussi avec plus que 2 listes

Attention : si les 2 listes n'ont pas le même nombre d'éléments, zip s'arrête au dernier élément de la liste la plus courte (éléments suivants de la liste la plus longue sont ignorés).
Quand on veut boucler sur 2 itérateurs en même temps, même si ce sont des générateurs, on peut faire :

IDLE Shell

```
>>for x, y in zip(myIterator1, myIterator2# (utile pour lire 2 fichiers en même temps dont les éléments se correspondent).
```

reduce(myFunc, l)	appelle myFunc sur les 2 premiers éléments de l, puis sur le résultat et le 3eme, etc ... et renvoie le dernier résultat.

map et filter fonctionnent aussi sur les sets.

Application : faire à partir d'un dictionnaire de liste, une liste de dictionnaires ayant les mêmes clefs et toutes les combinaisons de valeurs possibles (produit cartésien) :

IDLE Shell

```
>>>d = {'a': [4, 6], 'b': [7, 8, 9]}
>>>keys, values = zip(*d.items())
>>>dictList = [dict(zip(keys, v)) for v in itertools.product(*values)]
```

Résultat:

```
[{'a': 4, 'b': 7}, {'a': 4, 'b': 8}, {'a': 4, 'b': 9}, {'a': 6, 'b': 7}, {'a': 6, 'b': 8}, {'a': 6, 'b': 9}]
```

On peut utiliser map sur plusieurs séquences avec une fonction qui prend plusieurs arguments

IDLE Shell

```
>>> map(lambda x,y: x + y, ['a', 'b', 'c', 'd'], ['1', '2', '3', '4'])
```
Résultat:
['a1', 'b2', 'c3', 'd4']

On peut même faire

IDLE Shell

```
>>> map(lambda x,y: x + y, ['a', 'b', 'c', 'd'], '1234')
```
Résultat:
['a1', 'b2', 'c3', 'd4']

-enumerate pour boucler sur les éléments d'une séquence et avoir les index en même temps :
-renvoie un itérateur qui renvoie des tuples (numéro d'index, élément).

IDLE Shell

```
>>> enumerate(['a', 'b', 'c']) # renvoie un itérateur qui va renvoyer les couples suivants
```
Résultat:
(0, 'a'), (1, 'b'), (2, 'c')

Exemple :

```
for (ind, elt) in enumerate(['a', 'b', 'c']):
    print(ind, elt)
```

Editeur

```
for(ind,elt) in enumerate(['a', 'b', 'c'])
print(ind, elt)
```

IDLE Shell

```
>>> list(enumerate(['a', 'b', 'c']))
```
Résultat:
[(0, 'a'), (1, 'b'), (2, 'c')]

On peut utiliser enumerate sur n'importe quelle séquence :

IDLE Shell

```
>>> list(enumerate('abc'))
```
Résultat:
[(0, 'a'), (1, 'b'), (2, 'c')]

On peut changer l'index de départ :

IDLE Shell

```
>>> list(enumerate(['a', 'b', 'c'], start = 1))
#Ou simplement
>>> list(enumerate(['a', 'b', 'c'], 1))
```

Résultat :

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

min, max : on peut choisir l'ordre à appliquer : max(myList, key = lambda x: ...)

Dictionnaires

Dictionnaires : les clefs peuvent être n'importe quel type non mutable :

- Un tuple dont les éléments sont non mutables.
- Un namedtuple (dont les éléments sont non mutables).
- Un frozenset pas une liste.
- Mais pas un tuple dont certains éléments sont une liste.
- Un dictionnaire peut avoir une clef à None (mais bien sûr, une seule !).

Manipulations élémentaires

IDLE Shell

```
>>> d = {} #Initialisation à vide
# On peut aussi faire
>>> d = dict()
>>> d = {'toto': 1, 'titi': 2} # initialisation.
```

On peut aussi définir un dictionnaire avec une liste de paires (tuples à 2 valeurs) :

IDLE Shell

```
>>> d = dict([('a', 1), ('b', 2)])
# Affectation d'une valeur
>>> d['toto'] = 5
```

Les clefs d'un dictionnaire peuvent être des tuples (de constantes) ! (mais pas des listes)

IDLE Shell

```
>>> d = dict([('a', 1), ('b', 2)])
# Affectation d'une valeur
>>> d['toto'] = 5
#Destruction d'une clef
>>> del d['toto']
>>> d.keys() #renvoie un itérateur sur les clefs
# Si on veut la liste des clefs, faire
>>> list(d.keys())
>>>'toto' in d  #test de l'existence d'une clef
# Pour tester la présence d'une clef
>>> key in d
#   ou
>>> key not in d
>>> len(d) #renvoie le nombre d'items dans le dictionnaire
```

Pour boucler sur les clefs d'un dictionnaire :

Editor

```
for x in d:
  print(x)
```

Pour boucler sur un dictionnaire en récupérant en même temps les clefs et les valeurs :

Editor

```
for k, v in d.items():
  print(repr(k) + ' : ' + repr(v))
```

Autres fonctions sur les dictionnaires

d.clear()	efface tout.
d.copy()	Copie superficielle (shallow copie) fait référence à une copie d'objet qui ne duplique que les références aux variables de l'instance, sans créer de nouveaux objets.
d.get(key)	renvoie la valeur si clef présente, sinon None
d.get(key, defaultVal)	renvoie la valeur si clef présente, sinon defaultVal
d.items()	renvoie un itérateur sur le dictionniare renvoyant les paires (key, value).
d.pop(key)	enlève la clef et renvoie la valeur (KeyError si clef n'existe pas).
d.pop(key, val)	enlève la clef si elle existe et renvoie la valeur. Renvoie val si clef n'existe pas (sans lever une exception).
d.popitem()	renvoie une paire (key, value) au hasard et la retire du dictionnaire.
d.values()	renvoie un itérateur sur les valeurs.
d.setdefault(key, defaultVal)	si la clef existe, renvoie sa valeur, si la clef n'existe pas, insère la clef avec la valeur defaultVal et renvoie sa valeur.
Pour avoir la première clef d'un dictionnaire	**next(iter(myDict.keys()))**
d2 = dict(d)	fait une copie indépendante du dictionnaire.
Pour rajouter 2 dictionnaires l'un à l'autre :	
d1.update(d2).	modifie d1 en rajoutant les clefs/valeurs de d2 (si clefs communes, c'est la valeur de d2 qui est présente à la fin
{k:v for d in (d1, d2) for k,v in d.items()}	renvoie le dictionnaire avec les clefs de d1 et celles de d2 (si clefs communes, ce sont les valeurs de d2)
dict(list(d1.items()) + list(d2.items()))	autre possibilité.

Dictionnaire de compréhension

IDLE Shell

```
>>>d = {x: 2 * x for x in range(4)}
Résultat:
{0: 0, 1: 2, 2: 4, 3: 6}
```

Inversion des clefs/valeurs d'un dictionnaire

- Quand les valeurs sont uniques :

Editeur

```
invD = {value:key for key, value in d.items()}
```

- Quand les valeurs ne sont pas uniques :

Editeur

```
invD = {}
for k, v in d.items():
   invD.setdefault(v, []).append(k)
```

Pour supprimer les doublons d'une liste sans changer l'ordre :

Editeur

```
collections.OrderedDict.fromkeys(myList).keys()
```

Sets

Un set est un ensemble de clefs non ordonnées et non redondant où l'on peut savoir si un élément est présent sans avoir à parcourir toute la liste (une sorte de dictionnaire où les valeurs seraient ignorées, seules les clefs comptent).
Un set peut avoir comme élément None.

Set :

console

```
#Définition d'un set
>>> s = set(['a', 'b', 'c'])
# Initialisation d'un set avec aucun élément
>>> s1= set()
# Définition d'un set non m,



+
```

Set de compréhension :

Editeur

```
{x for x in range(5)}
```

Test si un élément appartient au set :

IDLE Shell

```
>>'a' in s
#on peut aussi faire
>>> 'a' not in s
```

On peut aussi faire un set de lettres à partir d'un string :

Console

```
>> s = set('wissal bounar')
>> s2 = set('14/08/2004')
#Ajout d'un élément
>>>s.add('d')
# rajoute les éléments à s2 et modifie s
>>> s.update(s2)
# On peut aussi donner un argument une liste
>>>s.update(myList)
```

IDLE Shell

```
#Retrait d'un élément
>>>s.remove('d') # donne une exception KeyError si non présent
#Retrait d'un élément s'il existe
>>>s.discard('d').
#Pour enlever un élément au hasard et le renvoyer
>>>s.pop()
#Pour récupérer un élément au hasard sans l'enlever
>>> next(iter(s))#utiliser a prtir e la version de python
#Pour effacer tous les éléments au hasard et le renvoyer
>>>s.clear()
Pour boucler sur les éléments d'un set
>>>for x in s:
#Nombre d'éléments du set
>>>len(s)
#Autres fonctions des set
>>>s.isdisjoint(s2)
>>>s.issubset(s2)
>>>s.issuperset(s2)
# Inclusion
>>>s <= s2  #pareil avec s >= s2
# inclusion stricte
>>>s < s2  #pareil avec s > s2
>>>s.union(s2) #n'affecte pas les sets de départ (on peut aussi faire s | s2).
>>>set.union(s1, s2, s3) #renvoie la réunion de différents sets (utile quand plus que 2).
>>>s.intersection(s2) #on peut aussi faire s & s2
>>>set.intersection(s1, s2, s3) #renvoie l'intersection de différents sets (utile quand plus que 2)
>>>s1 & s2 & s3 #permet aussi de faire l'intersection de multiples sets.
>>>s1 | s2 | s3 #permet de faire le réunion de multiples sets.
>>>s.difference(s2) #on peut aussi faire s - s2
>>>s.symmetric_difference(s2) #on peut aussi faire s ^ s2
>>>s.copy() #renvoie une shallow copy.

#union, intersection, difference, symmetric_difference, issubset, issuperset acceptent des
listes #en arguments, mais pas leur équivalents avec un symbole.

>>>s.intersection_update(s2) #fait l'intersection et modifie s (on peut aussi faire s &= s2).
>>>s.difference_update(s2) #fait la différence et modifie s (on peut aussi faire s -= s2).
#Différence symétrique et modification de  s (on peut aussi faire s ^= s2).
>>>s.symmetric_difference_update(s2)
Pour convertir les clefs d'un dictionnaire en set
>>>mySet = set(myDict.keys())
#Ou même
>>>mySet = set(myDict)
```

Arrays

Arrays :

C’est une séquence qui permet de représenter de manière compacte une liste de valeurs toutes du même type (élémentaire). Sa taille n'est pas fixe contrairement aux arrays numpy.
Pour l'utiliser, il faut importer array comme ça :

import array as arr

Définition d'une array :

Définition d’un array de type double

IDLE Shell

```
import array

# Création d’un vecteur d’entiers (type 'i' = integer)
vecteur = array.array('c', [‘1’,’ 2’, ‘3’, ‘4’, ‘5’])

print(vecteur)
```

IDLE Shell

```
import numpy as np

# Création d’un vecteur d’entiers (type 'i' = integer)
vect = np.array('i', [1, 2, 3, 4, 5])

print(vect)
```

Les différents types sont

Type	Désignation
c	Caractère
b	Entiers signés sur 1 octet.
B	Entiers non signés sur 1 octet.
i	Entiers signés sur 2 octets.
I	Entiers non signés sur 2 octets.
f	float sur 4 octets.
d	double sur 8 octets.

IDLE Shell

```
Pour définir une array de caractères, donner une chaîne
>>>a = array.array('b', 'abcde')
#Pour avoir la place occupée par un élément :
>>>print(a.itemsize)
#Pour avoir le type (lettre ci-dessus)
>>>print(a.typecode)
>>>a.tolist() #renvoie une liste normale.
#Lectures et modifications d'une array :
>>>a[0],a[1:3] #et toutes les opérations sur les séquences sont valables.
>>>a.append(5.3) #rajoute une valeur à la fin.
>>>a.extend([2.1, 8.3]) #rajoute à la fin la liste ou l'array (qui doit alors avoir le même type).
>>>a.count(3.2) #renvoie le nombre de valeurs qui sont à 3.2 dans l'array.
>>>a.index(3.2) #renvoie le plus petit index (origine à 0) dont la valeur est 3.2.
>>>a.insert(2, 7.7) #insère la valeur à la position 2 (origine à 0) et décale les valeurs après.
>>>a.insert(-1, 7.7) #insère la valeur à l'avant-dernière position.
>>>a.remove(7.7) #enlève la première valeur qui vaut 7.7 (et décale le reste).
>>>a.pop(2) #enlève la valeur à la position 2 (origine à 0) et la renvoie.
>>>a.pop() #enlève la dernière valeur et la renvoie (équivalent à a.pop(-1).
>>>a.reverse() #renverse la liste en place.
```

Creation d'un tableau avec array

Editor

```
import array as arr
import math# utilisé la biblithoèque mathématique
a = arr.array( [2, 4, 6, 8])

print("Premier élement:", a[0])
print("Deuxième élement:", a[1])
print("Dernier élement:", a[-1])
```

Commande «Tyoe » pour connaitre le type de tableau

IDLE Shell

```
>> type(a)
```

Tableau linéaire avec la bibliothèque numpy

Installation de la bibliothèque numpy

- C:\Windows\system32>pip install numpy
  Il va télécharger et installer le package de numpy.

Application

Editeur

```
import numpy as np
import os
os.system("cls")# clear screen avecwndows
os.system("clear")# car sur linux
#Convertit une liste contenant les éléments 5, 2, 8, 17, 6, 14 en un tableau  numpy contenant
# les mêmes éléments et dans le même ordre.
T = np.array([5,2,8,17,6,14])
print(T) #impression du résultat

#Crée un tableau contenant 15 valeurs entières allant de 0 à 14
T1 = np.arange(15)
print(T1) #impression du résultat

#Crée un tableau contenant des valeurs séparées de 0.5 comprises dans l'intervalle [0.9;8.1].
#Le premier paramètre précise la valeur initiale du tableau.
#Les valeurs du tableau seront comprises dans l'intervalle [premier paramètre, second
paramètre].
#Le dernier paramètre indique l'intervalle entre deux valeurs successives du tableau.
#En langage Matlab T2=0.9:0.5:8.1 ou bien  T2 = (0.9:0.5;8.1)
T2 = np.arange(0.9,8.1,0.5)
print(T2) #impression du résultat

#Crée un tableau contenant 16 valeurs (de 0 à 1/4=0,25).
#Le premier paramètre précise la valeur initiale du tableau.
#Le second paramètre précise la valeur finale du tableau.
#Le dernier paramètre indique le nombre total de valeurs.
#En Matlab  y1 = linspace (0:1/4:16) 16 nombre de valeur  0 et 1/4
T3 = np.linspace(0,1/4,16)
print(T3) #impression du résultat
```

Editeur

```
# Plusieurs manières de récupérer les éléments d'un tableau numpy :
# Applications

print(T)
print(T[0])
print(T[2])
print(T[-1])
print(T[-2])
print(T[1:3])
print(T[3:])
print(len(T))
T4 = np.array([])
print(T4)

# PLusieurs fonctions pour ajouter une valeur ou supprimer des valeurs
# d'un tableau numpy
print(T)
T1=np.append(T,18)     # Crée un tableau T1 à partir du tableau T
print(T1)              # en ajoutant l'élément 18 à la fin du
print(T)               # tableau T. Ne modifie pas le tableau T.

T=np.append(T,18)      # Ajoute l'élément 18 à la fin du tableau T.
print(T)

T=np.insert(T,2,20)    # Insère l'élément 20 à la position d'indice 2
print(T)               # (donc en troisième position) du tableau T.


T=np.delete(T,2)       # Supprime l'élément d'indice 2 (ici l'élément 20)
print(T)               # du tableau T.
T=np.delete(T,-1)      # Supprime l'élément d'indice -1 (donc le dernier
print(T)               # élément) du tableau T.
```

Fonction

Les fonctions natives (préinstallée)
Une fonction native en Python est une fonction qui fait partie de la bibliothèque de base de Python.

Exemple de fonctions natives en Python :

Fonction abs(A) : retourne une valeur absolue de A

IDLE Shell

```
>> abs(-2)
```
Résultat :
1

Fonction all(iterable) : Retourne True si tous les éléments d'un élément itérable sont True

IDLE Shell

```
>>> liste = [True,True,True,1]
>>> all(liste)
```
Résultat :
True

Fonction any(iterable) : Retourne True si au moins un élément d'un élément itérable est True

IDLE Shell

```
>>> liste = [True,False, True]
>>> any(liste)
```
Résultat :
True

Fonction bin(x) : convertit un integer en chaine de caractères binaires.

IDLE Shell

```
>> bin(101)
```
Résultat :
'0b1100101'

Fonction callable(object) : détermine si un objet est callable.

IDLE Shell

```
>>> callable("A")
```
Résultat :
False
```
>>> callable(int)
```
Résultat :
True

Fonction str.capitalize() : La méthode capitalize permet de mettre une chaine de caractères au format Xxxxx

IDLE Shell

```
>>> liste = [True,False, True]
>>> any(liste)
```
Résultat :
True
```
>>> "bEnkinouAR".capitalize()
'Benkinouar'
```

Fonction choice([]) : Retourne une valeur d'une liste aléatoirement.

IDLE Shell

```
>>> import random
>>> random.choice([1,2,3,4,5])
```
Résultat :
3
```
>>> random.choice([1,2,3,4,5])
```
Résultat :
2

Fonction str.count(string)
La méthode count compte le nombre d'occurences de la recherche demandée.

IDLE Shell

```
>>> "benkinouar".count("n")
```
Résultat :
2

Fonction dir(object)
Indique les noms de la structure de l'objet.

IDLE Shell

```
>>> dir(int)
```
Résultat :
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'is_integer', 'numerator', 'real', 'to_bytes']

Fonction str.endswith(str)
La méthode endswith test si une chaine de caractères se termine par la chaine demandée

IDLE Shell

```
>>> a = "benkinouar"
>>> a.endswith("r")
Résultat :
True
>>> a.endswith("ar")
Résultat :
True
>>> a.endswith("a")
Résultat :
False
```

Fonction eval(expression, **globals**=None, **locals**=None**)**
Execute une chaine de caractères.

IDLE Shell

```
>>> v = 101
>>> eval('v+1')
Résultat :
102
```

Fonction str.find(string)
La méthode find trouve la première occurence de la recherche demandée.

IDLE Shell

```
>>> "benkinouar".find("n")
Résultat :
2
```

Fonction help(element)
Cette fonction vous retourne des informations sur l'utilisation de l'élément qui vous intéresse.

IDLE Shell

```
>>>help(int)
```

Fonction hex
Convertit un nombre en valeur hexadécimale.

IDLE Shell

```
>>>hex(16)
Résultat :
'0x10'
```

Fonction str.isalnum()
Retoune True si tous les caractères sont alphanumériques et qu'il y a au moins un caractère. Sinon False.

IDLE Shell

```
>>> "25".isalnum()
Résultat :
True
>>> "25b".isalnum()
Résultat :
True
>>> "25bé".isalnum()
Résultat :
True
>>> "25bé@".isalnum()
Résultat :
False
>>> "-".isalnum()
Résultat :
False
>>> "_".isalnum()
Résultat :
False
>>> "".isalnum()
Résultat :
False
>>> "25".isalnum()
Résultat :
True
>>> "25b".isalnum()
Résultat :
True
>>> "25bé".isalnum()
Résultat :
True
>>> "25bé@".isalnum()
Résultat :
False
>>> "-".isalnum()
Résultat :
False
>>> "_".isalnum()
Résultat :
False
>>> "".isalnum()
Résultat :
False
```

Fonction str.isalpha()
Retourne True si tous les caractères sont des lettres et qu'il y a au moins un caractère. Sinon False

IDLE Shell

```
>>> "x".isalpha()
Résultat :
True
>>> "-".isalpha()
Résultat :
False
>>> "12".isalpha()
Résultat :
False
>>> "jean-claude".isalpha()
Résultat :
False
>>> "jean claude".isalpha()
Résultat :
False
>>> "élise".isalpha()
Résultat :
True
```

Fonction str.isdigit()
Retourne True si tous les caractères sont numériques et qu'il y a au moins un caractère. Sinon False.

IDLE Shell

```
>>> "1".isdigit()
Résultat :
True
>>> "1.5".isdigit()
Résultat :
False
>>> "1,5".isdigit()
Résultat :
False
>>> "3b".isdigit()
Résultat :
False
>>> " ".isdigit()
Résultat :
False
```

Fonction str.islower()
Retoune True si tous les caractères sont en minuscule.

IDLE Shell

```
>>> "azzeddine".islower()
Résultat :
True
>>> " Azzeddine ".islower()
Résultat :
False
```

Fonction str.isspace()
Retoune True si il n'y a que des espaces et au moins un caractère.

IDLE Shell

```
>>> " ".isspace()
Résultat :
True
>>> "Fezani Azzeddine".isspace()
Résultat :
False
>>> "    ".isspace()
Résultat :
True
```

Fonction str.istitle()
Retourne True si la chaine a un format titre.

IDLE Shell

```
>>> "Titre".istitle()
Résultat :
True
>>> "TitrE".istitle()
Résultat :
False
>>> "Titre de mon site".istitle()
Résultat :
False
>>> "Titre De Mon Site".istitle()
Résultat :
True
```

Fonction str.isupper()
Retourne True si tous les caractères sont en majucule et qu'il y a au moins un caractère.

IDLE Shell

```
>>> "BENKINOUAR".isupper()
Résultat :
True
>>> "Benkinouar".isupper()
Résultat :
False
>>> "BenkinouaR".isupper()
Résultat :
False
```

Fonction str.join(liste)
La méthode join transforme une liste en chaine de caractères.

IDLE Shell

```
>>> ":".join(["Somme", "m"])
Résultat :
'Somme:m'
```

Fonction len(s)
Retourne le nombre d'items d'un objet.

IDLE Shell

```
>>> len([1,2,3])
Résultat :
3
>>> len("olivier")
Résultat :
7
```

Fonction locals()
Retounr un dictionnaire avec les valeurs des variables en cours.

IDLE Shell

```
>>> locals()
Résultat :
{'a': 12, '__builtins__': , '__package__': None, 'i': 20, 'v': 101, 'liste': [True, False, True], '__name__': '__main__', '__doc__': None}
```

la fonction str.lower()
La méthode lower permet de mettre en minuscule une chaine de caractères.

IDLE Shell

```
>>> "BENKINOUAR".lower()
Résultat :
'benkinouar'
```

Function map(function, [])
Exécute une fonction sur chaque item d'un élément itérable.

IDLE Shell

```
>>> def add_one(x):
...     return x + 1
...
>>> map(add_one, [1,2,3])
[2, 3, 4]
Résultat :
3
```

Fonction max() / min()
Retourne la valeur la plus élevée pour max() et la plus basse pour min()

IDLE Shell

```
>>> max([1,3,2,6,99,1])
Résultat :
99
>>> max(1,4,6,12,1)
Résultat :
12
```

Fonction randint()
Retourne un int aléatoire.

IDLE Shell

```
>>> import random
>>> random.randint(1,11)
Résultat :
5
```

Fonction random()
Retourne une valeur aléatoire.

IDLE Shell

```
>>> import random
>>> random.random()
Résultat :
0.9563522652738929
```

Fonction str.replace(string, string)
La méthode replace remplace un segment d'une chaine de caractères par une autre:

IDLE Shell

```
>>> "Amine".replace("e", "a")
Résultat :
'Amina'
```

Fonction reverse()
La méthode reverse inverse l'ordre d'une liste.

IDLE Shell

```
>>> x = [1,4,7]
>>> x.reverse()
>>> x
Résultat :
[7, 4, 1]
```

Fonction reversed([])
Retourne un itérateur inversé.

IDLE Shell

```
>>> list(reversed([1,2,3,4]))
Résultat :
[4, 3, 2, 1]
```

Fonction round(number)
Arrondi un nombre.

IDLE Shell

```
>>> round(1)
Résultat :
1.0
>>> round(1.2)
Résultat :
1.0
>>> round(1.5)
Résultat :
2.0
>>> round(1.7)
Résultat :
2.0
>>> round(-1.7)
Résultat :
-2.0
>>> round(-1.2)
Résultat :
-1.0
```

Fonction shuffle([])
Mélange aléatoirement une liste.

IDLE Shell

```
>>> import random
>>> x = [1,2,3,4,5]
>>> random.shuffle(x)
>>> x
Résultat :
[2, 5, 4, 1, 3]
```

Fonction str.startswith(prefix[, start[, end]])
Retourne True si la chaine commence par le préfix indiqué. Ce préfix peut être un tuple. Les paramètres start et end (optionnel) test la chaine à la position indiquée. Le test est sensible à la case.

IDLE Shell

```
>>> "olivier".startswith("ol")
Résultat :
True
>>> "olivier".startswith(("ol", "eng"))
Résultat :
True
>>> "olivier".startswith(("xxx", "eng"))
Résultat :
False
>>> "olivier".startswith("OL")
Résultat :
False
>>> "olivier".startswith("ol")
Résultat :
True
```

Fonction list.sort()
La méthode sort permet de trier une liste.

IDLE Shell

```
>>> l = [5,1,4,2,10]
>>> l.sort()
>>> l
Résultat :
[1, 2, 4, 5, 10]
```

Fonction sorted(iterable)
Tri un élément itérable.

IDLE Shell

```
>>> sorted([3,2,12,1])
Résultat :
[1, 2, 3, 12]
```

Fonction str.split(séparateur)
La méthode split transforme une chaine de caractères en liste.

IDLE Shell

```
>>> "Benkinouar:Moad".split(":")
Résultat :
['Benkinouar', 'Moad']
```

Fonction str.splitlines([keepends])
Retourne une liste des lignes de la chaine. Cette méthode utilise le saut à la ligne universel, le retour à la ligne n'est pas inclue, à moins de renseigner le paramètre keepends à True.

IDLE Shell

```
>>> "benkinouar\n\n\moad\n\ndéveloppeur".splitlines()
Résultat :
['benkinouar', '', '\\moad', '', 'développeur']
>>> "benkinouar\nmoad\ndéveloppeur".splitlines()
Résultat :
['benkinouar', 'moad', 'développeur']
>>> "benkinouar\n\rmoad\n\rdéveloppeur".splitlines()
Résultat :
['benkinouar', '', 'moad', '', 'développeur']
>>> "benkinouar\r\nmoad\r\ndéveloppeur".splitlines()
Résultat :
['benkinouar', 'moad', 'développeur']
>>> "benkinouar\r\nmoad\r\n\r\ndéveloppeur".splitlines()
Résultat :
['benkinouar', 'moad', '', 'développeur']
>>> "benkinouar\r\nmoad\r\n\r\ndéveloppeur".splitlines(True)
Résultat :
['benkinouar\r\n', 'moad\r\n', '\r\n', 'développeur']
```

Fonction sum(iterable [,start])
Additionne les valeurs d'un élément itérable.

IDLE Shell

```
>>> sum([1,2,3])
Résultat :
6
```

Fonction str.title()
Transforme la chaine dans un format title.

IDLE Shell

```
>>> "Ceci est un titre".title()
Résultat :
'Ceci Est Un Titre'
```

Fonction upper()
La méthode upper permet de mettre en majuscule une chaine de caractères.

IDLE Shell

```
>>> "benkinouar".upper()
Résultat :
'BENKINOUAR'
```

Fonction zip(*iterables)
Permet de regrouper sous la forme d'un tuple les items de listes.

IDLE Shell

```
>>> a = ["Benkinouar", "BenSouilah", "Fezani"]
>>> b = ["Moad", "Bachir", "Azzeddine"]
>>> result = list(zip(a, b))
>>>print(result)
Résultat :
[('Benkinouar', 'Moad'), (' BenSouilah', ' Bachir'), (' Fezani', ' Azzeddine')]
```

Quelques fonctions
Python dispose de nombreuses fonctions utiles pour manipuler les structures et données. Le tableau suivant en répertorie quelques-unes. Certaines nécessitent le chargement de la librairie math, d'autres la librairie statistics.

Quelques fonctions numériques	
Fonction	**Description**
math.ceil(x)	Plus petits entier supérieur ou égal à x
math.copysign(x, y)	Valeur absolue de x mais avec le signe de y
math.floor(x)	Plus petits entier inférieur ou égal à x
math.round(x, ndigits)	Arrondi de x à ndigits décimales près
math.fabs(x)	Valeur absolue de x
math.exp(x)	Exponentielle de x

Quelques fonctions numériques	
Fonction	**Description**
math.log(x)	Logarithme naturel de x (en base e)
math.log(x, b)	Logarithme en base b de x
math.log10(x)	Logarithme en base 10 de x
math.pow(x,y)	x élevé à la puissance y
math.sqrt(x)	Racine carrée de x
math.fsum()	Somme des valeurs de x
math.sin(x)	Sinus de x
math.cos(x)	Cosinus de x
math.tan(x)	Tangente de x
math.asin(x)	Arc-sinus de x
math.acos(x)	Arc-cosinus de x
math.atan(x)	Arc-tangente de x
math.sinh(x)	Sinus hyperbolique de x
math.cosh(x)	Cosinus hyperbolique de x
math.tanh(x)	Tangente hyperbolique de x
math.asinh(x)	Arc-sinus hyperbolique de x
math.acosh(x)	Arc-cosinus hyperbolique de x
math.atanh(x)	Arc-tangente hyperbolique de x
math.degree(x)	Conversion de x de radians en degrés
math.radians(x)	Conversion de x de degrés en radians
math.factorial()	Factorielle de x
math.gcd(x, y)	Plus grand commun diviseur de x et y
math.isclose(x, y, rel_tol=1e-09, abs_tol=0.0)	Compare x et y et retourne s'ils sont proches au reard de la tolérance rel_tol (abs_tol est la tolérance minimum absolue)
math.isfinite(x)	Retourne True si x est soit l'infini, soir NaN
math.isinf(x)	Retourne True si x est l'infini, False sinon
math.isnan(x)	Retourne True si x est NaN, False sinon

Quelques fonctions numériques	
Fonction	**Description**
statistics.mean(x)	Moyenne de x
statistics.median(x)	Médiane de x
statistics.mode(x)	Mode de x
statistics.stdev(x)	Écart-type de x
statistics.variance(x)	Variance de x

Quelques constantes

La librairie math propose quelques constantes :

Quelques constantes intégrées dans Python	
Fonction	**Description**
math.pi	Le nombre Pi (π)
math.e	La constante e
math.tau	La constante τ, égale à 2π
math.inf	L'infini (∞)
-math.inf	Moins l'infini (−∞)
math.nan	Nombre à virgule flottante *not a number*

Définition d'une fonction

Syntaxe

```
def nom_fonction(e1, e2,...,en):% e1, e2,...,en sont les paramètre d'entrées
    bloc d'instructions
#Appel de la fonction
my_function()
```

Exemple

Editeur

```
def my_function():
  print("Hello from a function")
  return 1
my_function()
```

Documentation d'une fonction : mettre la chaîne de documentation en première ligne du corps :

Editeur

```
def myFunc(n):
   """Chaine de documentation."""
   print(n)
   return 1
myFunc(3)
```

Les paramètres sont passés par valeur (mais c'est la référence qui est passée par valeur quand c'est une variable de type liste ou dictionnaire ou tuple).

Par défaut, les variables d'une fonction sont locales

On ne peut pas accéder à la valeur d'une variable locale avant de l'avoir initialisée (n'est pas initialisée à None par défaut !).

On ne peut pas modifier une variable globale, mais on peut par contre la lire (dans une fonction, la variable est cherchée successivement dans la table des symboles locale, puis globale, puis built-in)

Valeur de retour d'une fonction :

- Une fonction peut renvoyer un argument avec return.

Editeur

```
def multiplyNum(num1) :
   return num1 * 8

résultat = multiplyNum(8)
print(résultat)
```

Résultat

IDLE Shell

•
64

- Une fonction qui n'a pas de return ou un return sans valeur renvoie None.
- Une fonction peut renvoyer un tuple qui est récupéré dans plusieurs variables à condition que le nombre de variables matche le nombre de valeurs du tuple :

Exemple

Editeur

```
def myFunc():
    return (1, 'a')

(x, y) = myFunc() # valide
rint(x,y)
#(x, y, z) = myFunc() # invalide
```

Convention lorsqu'on veut récupérer seulement certains arguments de retour d'une fonction qui renvoie une liste ou un tuple : on utilise '_' pour ceux qu'on ne veut pas récupérer (mais c'est seulement une convention).

Exemple :

Editeur

```
def myFunc():
    return (1, 'a')

x_, y_ = myFunc()
```

On peut définir une fonction avec des valeurs par défaut :

```
def myFunc(a, b = 4):
    return (1, 'a')

(x, y )= myFunc(0, 1)# ou myFunc(0)
```

L'instruction return peut contenir une expression à exécuter une fois la fonction appelée.

Attention aux valeurs par défaut mutables !

La valeur par défaut n'est évaluée qu'une seule fois !

Editeur

```
def f(x, y = []):
    y.append('a')
    print(y)
f(1); f(2); f(3)
```

Résultat

IDLE Shell

```
['a']
['a', 'a']
['a', 'a', 'a']
```

Si on ne veut pas ce comportement, mettre une valeur par défaut à None, et fixer la valeur dans la fonction si la valeur vaut None.
En bref, éviter les objets mutables comme valeurs par défaut dans la définition d'une fonction !
On peut appeler une fonction avec des arguments précisés par nom :

Editeur

```
def myFunc(a, b = 4):
    print(a,b)
myFunc(a = 1, b = 5)
myFunc(a = 2)
```

Les arguments par défaut sont évalués dans le scope de la définition :

Editeur

```
a= 3#variable globale
def myFunc(x = a):
   print(x)
myFunc(x=a)
a = 4
```

Résultat

IDLE Shell

```
3 # et non 4
```

Calcul d'une somme méthode classique pour une liste

Editeur

```
li = [0, 434, 43, 6456]
s = 0                  # initialisation
for j in li:           # boucle
   s += j              # addition
print(s)
```

Résultat

IDLE Shell

```
6933
```

Editeur

```
def fonction(x):
    return x

li = [0, 434, 43, 6456]
s = 0
for j in li:
    s += fonction(j)
print(s)
```

Résultat

IDLE Shell

```
6933
```

Et ces deux lignes pourraient être résumées en une seule grâce à l'une de ces instructions :

Editeur

```
def fonction(x):
    return x

li = [0, 434, 43, 6456]
s1 = sum([fonction(J) for J in li])
s2 = sum(fonction(J) for J in li)
s3 = sum(map(fonction, li))
print(s1, s2, s3)
```

Résultat

IDLE Shell

```
6933 6933 6933
```

Remarque : L'avantage des deux dernières instructions est qu'elles évitent la création d'une liste intermédiaire, c'est un point à prendre en compte si la liste sur laquelle opère la somme est volumineuse

Fonction comme paramètre

Une fonction peut aussi recevoir en paramètre une autre fonction. L'exemple suivant inclut la fonction calcul_n_valeur qui prend comme paramètres l et f. Cette fonction calcule pour toutes les valeurs x de la liste l la valeur f(x). fonction_carre ou fonction_cube sont passées en paramètres à la fonction calcul_n_valeur qui les exécute.

Editeur

```
def fonction_carre(x):
    return x * x

def fonction_cube(x):
    return x * x * x

def calcul_n_valeur(list, f):
    res = [f(i) for i in list]
    return res

L = [0, 1, 2, 3]
print(L)

L1 = calcul_n_valeur(L, fonction_carre)
print(L1)

L2 = calcul_n_valeur(L, fonction_cube)
print(L2)
```

Résultat

IDLE Shell

```
[0, 1, 2, 3] # Résultat  de l'apopel de print(L)
[0, 1, 4, 9] # Résultat  de l'apopel de print(L1)   avec L1 = calcul_n_valeur(L, fonction_carre)
[0, 1, 8, 27] # Résultat  de l'apopel de print(L2) avec L2 = calcul_n_valeur(L, fonction_cube)
```

Une fonction peut être stockée dans une variable avant d'être utilisée :

Editeur

```
def myFunc(x):
    return x + 1

f = myFunc#stockage d'une fonction dans une variable « f »
print(f(3)) #ici on utilise la variable « f » au lieu d'utiliser le nom de la fonction « myFunc ».
```

Résultat

IDLE Shell

```
4
```

On peut définir une fonction à l'intérieur d'une fonction, pour que la fonction englobante retourne une fonction :

Editeur

```
def  buildMyFunc(a):
    def myFunc(x):
        return a * x
    return myFunc

f = buildMyFunc(2) #stockage d'une fonction dans une variable « f »
f(3) # On utilise la variable « f » au lieu d'utiliser le nom de la fonction «buildMyFunc».
```

Résultat

IDLE Shell

```
6
```

Minimum avec position

La fonction min retourne le minium d'un tableau mais pas sa position. Le premier réflexe est alors de recoder le parcours de la liste tout en conservant la position du minimum.

Editeur

```
li = [0, 434, 43, 6436, 5]
m = 0
for i in range(0, len(li)):
    if li[m] < li[i]:
        m = i
print(m)
```

Résultat

IDLE Shell

```
3
```

Mais il existe une astuce pour obtenir la position sans avoir à le reprogrammer.

Editeur

```
li = [0, 434, 43, 6436, 5]
k = [(v, i) for i, v in enumerate(li)]
m = min(k)
print(m)
```

Résultat

IDLE Shell

```
(0, 0)
```

La fonction min choisit l'élément minimum d'un tableau dont les éléments sont des couples (élément du premier tableau, sa position).

Recherche avec index

Lorsqu'on cherche un élément dans un tableau, on cherche plus souvent sa position que le fait que le tableau contient cet élément.

Editeur

```
def recherche(li, c):
    for i, v in enumerate(li):
        if v == c:
            return i
    return -1
li = [45, 32, 43, 56]
print(recherche(li, 43))  # affiche 2
```

Résultat

IDLE Shell

```
2
```

En python, il existe un fonction simple qui permet de faire ça :

Editeur

```
li = [45, 32, 43, 56]
print(li.index(43))  # affiche 2
```

Résultat

IDLE Shell

```
2
```

Lorsque l'élément n'y est pas, on retourne souvent la position -1 qui ne peut être prise par aucun élément :

Editeur

```
if c in li:
    return li.index(c)
else:
    return -1
```

Même si ce bout de code parcourt deux fois le tableau (une fois déterminer sa présence, une seconde fois pour sa position), ce code est souvent plus rapide que la première version et la probabilité d'y faire une erreur plus faible.

Recherche dichotomique

La recherche dichotomique est plus rapide qu'une recherche classique mais elle suppose que celle-ci s'effectue dans un ensemble trié. L'idée est de couper en deux l'intervalle de recherche à chaque itération. Comme l'ensemble est trié, en comparant l'élément cherché à l'élément central, on peut éliminer une partie de l'ensemble : la moitié inférieure ou supérieure.

Editeur

```
def recherche_dichotomique(li, c):
    a, b = 0, len(li) - 1
    while a <= b:
        m = (a + b) // 2
        if c == li[m]:
            return m
        elif c < li[m]:
            b = m - 1
        else:
            a = m + 1
    return -1

print(recherche_dichotomique([0, 2, 5, 7, 8], 7))
```

Résultat

IDLE Shell

```
3
```

Tri, garder les positions initiales

Le tri est une opération fréquente. On n'a pas toujours le temps de programmer le tri le plus efficace comme un tri quicksort et un tri plus simple suffit la plupart du temps. Le tri suivant consiste à recherche le plus petit élément puis à échanger sa place avec le premier élément du tableau du tableau. On recommence la même procédure à partir de la seconde position, puis la troisième et ainsi de suite jusqu'à la fin du tableau.

Editeur

```
li = [5, 6, 4, 3, 8, 2]

for i in range(0, len(li)):
    # recherche du minimum entre i et len (li) exclu
    pos = i
    for j in range(i + 1, len(li)):
        if li[j] < li[pos]:
            pos = j
    # échange
    ech = li[pos]
    li[pos] = li[i]
    li[i] = ech

print(li)
```

Résultat

IDLE Shell

```
[2, 3, 4, 5, 6, 8]
```

La fonction sorted trie également une liste mais selon un algorithme plus efficace que celui-ci (voir Timsort). On est parfois amené à reprogrammer un tri parce qu'on veut conserver la position des éléments dans le tableau non trié. Cela arrive quand on souhaite trier un tableau et appliquer la même transformation à un second tableau. Il est toujours préférable de ne pas reprogrammer un tri (moins d'erreur). Il suffit d'appliquer la même idée que pour la fonction minindex.

Editeur

```
tab = ["zero", "un", "deux"]                    # tableau à trier
pos = sorted((t, i) for i, t in enumerate(tab))  # tableau de couples
print(pos)
```

Résultat

IDLE Shell

```
[('deux', 2), ('un', 1), ('zero', 0)]
```

Si cette écriture est trop succincte, on peut la décomposer en :

Editor

```
tab = ["zero", "un", "deux"]
tab_position = [(t, i) for i, t in enumerate(tab)]
tab_position.sort()
print(tab_position)
```

Résultat

IDLE Shell

```
[('deux', 2), ('un', 1), ('zero', 0)]
```

Constructions négatives

Eviter d'effectuer le même appel deux fois

Modifier un dictionnaire en le parcourant

Eviter d'effectuer le même appel deux fois

Dans cette fonction on calcule la variance d'une série d'observations.

Editeur

```
def moyenne(serie):
    return sum(serie) / len(serie)

def variance_a_eviter(serie):
    s = 0
    for obs in serie :
        s += (obs-moyenne(serie))**2
    return s / len(serie)
```

La fonction variance_a_eviter appelle la fonction moyenne à chaque passage dans la boucle. Or, rien ne change d'un passage à l'autre. Il vaut mieux stocker le résultat dans une variable :

Modifier un dictionnaire en le parcourant

Il faut éviter de modifier un container lorsqu'on le parcourt. Lorsqu'on supprime un élément d'un dictionnaire, la structure de celui-ci s'en trouve modifiée et affecte la boucle qui le parcourt. La boucle parcourt toujours l'ancienne structure du dictionnaire, celle qui existait au début au début de la boucle.

Editeur

```
d = { k: k for k in range(10) }
for k, v in d.items():
    if k == 4 :
        del d[k]
```

En Python, cela produit l'erreur qui suit mais d'autres langages ne préviennent pas (C++) et cela aboutit à une erreur qui intervient plus tard dans le code (comme une valeur numérique inattendue).

```
Traceback (most recent call last):
File "session1.py", line 176, in <module>
    l = liste_modifie_dans_la_boucle()
```

File "session1.py", line 169, in liste_modifie_dans_la_boucle
  for k,v in d.items():
RuntimeError: dictionary changed size during iteration

Il faut pour éviter cela stocker les éléments qu'on veut modifier pour les supprimer ensuite.

Editeur

```
d = { k:k for k in l }
rem = [ ]
for k,v in d.items():
  if k == 4 :
    rem.append(k)
for r in rem :
  del d[r]
```

Même si Python autorise cela pour les listes, il est conseillé de s'en abstenir ainsi que pour tout type d'objets qui en contient d'autres. C'est une habitude qui vous servira pour la plupart des autres langages.

Variable Globale et variable locale

Si un nom est déclaré comme global, alors toutes les références et affectations vont directement dans l'avant-dernière portée contenant les noms globaux du module. Pour pointer une variable qui se trouve en dehors de la portée la plus locale, vous pouvez utiliser l'instruction ***nonlocal.*** Si une telle variable n'est pas déclarée ***nonlocal,*** elle est en lecture seule (toute tentative de la modifier crée simplement une nouvelle variable dans la portée la plus locale, en laissant inchangée la variable du même nom dans sa portée d'origine).

Habituellement, la portée locale référence les noms locaux de la fonction courante. En dehors des fonctions, la portée l***ocale*** référence le même espace de nommage que la portée globale : l'espace de nommage du module. Les définitions de classes créent un nouvel espace de nommage dans la portée locale.

Portée globale d'une fonction

Il est important de réaliser que les portées sont déterminées de manière textuelle : la portée globale d'une fonction définie dans un module est l'espace de nommage de ce module, quelle que soit la provenance de l'appel à la fonction. En revanche, la recherche réelle des noms est faite dynamiquement au moment de l'exécution. Cependant la définition du langage est en train d'évoluer vers une résolution statique des noms au moment de la « compilation », donc ne vous basez pas sur une résolution dynamique (en réalité, les variables locales sont déjà déterminées de manière statique) !

Une particularité de Python est que, si aucune instruction **global ou nonlocal** n'est active, les affectations de noms vont toujours dans la portée la plus proche. Les affectations ne copient aucune donnée : elles se contentent de lier des noms à des objets. Ceci est également vrai pour l'effacement : l'instruction del x supprime la liaison de x dans l'espace de nommage référencé par la portée locale. En réalité, toutes les opérations qui impliquent des nouveaux noms utilisent la portée locale : en particulier, les instructions import et les définitions de fonctions effectuent une liaison du module ou du nom de fonction dans la portée locale.

L'instruction global peut être utilisée pour indiquer que certaines variables existent dans la portée globale et doivent être reliées en local ; **l'instruction nonlocal** indique que certaines variables existent dans une portée supérieure et doivent être reliées en local.

Exemple de portées et d'espaces de nommage

Ceci est un exemple montrant comment utiliser les différentes portées et espaces de nommage, et comment global et nonlocal modifient l'affectation de variable :

Editeur

```
#Définition de la fonction tes_de_porté qui imbrique les trois fonctions
#Locale, non_locale, globale
def test_de_porte():
    def locale ():
        spam = " spam est local "

    def non_locale():
        nonlocal spam
        spam = "spam n'est pas local"

    def globale():
        global spam
        spam = " spam est global "

    spam = "teste spam"
    locale()
    print("Après affectation locale:", spam)
    non_locale()
    print("Après affectation non locale:", spam)
    globale()
    print("Après affectation global:", spam)
#programme principale
test_de_porte ()
print("Porte globale:", spam)
```

Résultat

IDLE Shell

```
•
testé spam              #Après affectation locale
spam n'est pas local    #Après affectation non locale:
spam n'est pas local    #Après affectation global:
spam est global         #Porte globale:
```

Avec la formulation **, l'argument reçoit le dictionnaire de tous les arguments précisés par nom :

Editeur

```
def myFunc(**args):
        for kw in args.keys():
                print(kw, ':', args[kw])
```

Avec la formulation *, l'argument reçoit le tuple de tous les arguments :

Editeur

```
def myFunc(*args):
        for kw in args:
                print(kw)
```

On peut mélanger tout cela :

Editeur

```
def myFunc(a, *b, **c):
   print(a)
   print(b)
   print(c)

myFunc(4, 5, 7, x = 4, y = 8)
```

Résultat

IDLE Shell

```
4
(5, 7)
{'titi': 8, 'toto': 4}
```

Unpacking

Si fonction définie par :

Editeur

```
def myFunc(**args):
        for kw in args.keys():
                print(kw, ':', args[kw])

myFunc(**{'a' = 3, 'b' = 4})
```

Si fonction définie par :

Editeur

```
def myFunc(a, b):
        return a + b

#On peut aussi l'appeler avec
myFunc(**{'a' = 3, 'b' = 4})
# Ou également
myFunc(*[3, 4])
#Ou même
myFunc(*(3, 4)).
```

Variable statique

Variable statique dans une fonction, qui garde la mémoire entre 2 appels :
On peut définir une variable statique dans une fonction, par exemple :

Editeur

```
def myFunc(a):
    if getattr(myFunc, 'myVar', None) is None:
        # initialisation au premier appel
        myFunc.myVar = 0
    myFunc.myVar += a
    print(myFunc.myVar)
```

Résultat

IDLE Shell

```
1  # pour  myFunc(1)
3   # pour myFunc(2)
```

Test si un attribut est une fonction (ou méthode) : callable(myMeth) : renvoie True si myMeth est une fonction ou méthode, False sinon.

Appel des attributs par nom

hasattr(myObj, myField) renvoie True si l'objet a le champ défini, False sinon.
callable(myMeth) : renvoie True si myMeth est une fonction ou méthode.
appel d'une méthode par son nom sur un objet : l = [6, 3, 5, 1, 9]; getattr(l, 'sort')() : tri la liste en appliquant la méthode sort.
pour fixer la valeur d'un champ : setattr(myObj, 'myField', 35)
abs(-2) : valeur absolue du nombre.
Il n'y a pas de fonction standard sign en python pour avoir le signe d'un nombre !
On peut cependant en créer une facilement : sign = lambda x: (x > 0) - (x < 0). Alors :

sign(2) : donne 1.
sign(-2) : donne -1.
sign(0) : donne 0.

Ou alors, on peut utiliser numpy.sign(2), une fonction du package numpy (qui renvoie un int64 ou float64).

Fonction et tableau linéaire

Exemple Compte le nombre d'occurrences d'une valeur donnée dans un vecteur.

Editeur

```
def compte_occurrences(vecteur, valeur):
    """
    Arguments:
        vecteur (list): Le vecteur d'entrée.
        valeur: La valeur dont on souhaite compter les occurrences.

    Retourne:
        int: Le nombre d'occurrences de la valeur dans le vecteur.
    """
    return vecteur.count(valeur)

# Exemple d'utilisation
v = [1, 2, 3, 2, 4, 2, 5, 2]
valeur = 2
resultat = compte_occurrences(v, valeur)
print(f"La valeur {valeur} apparaît {resultat} fois dans le vecteur.")
```

Exemple : Calcule la somme de deux tableaux linéaires.

Editeur

```
def somme_deux_tableaux(tab1, tab2):
    """
        Arguments:
        tab1 (list): Premier tableau.
        tab2 (list): Deuxième tableau.

    Retourne:
        list: Tableau résultant de la somme des éléments de tab1 et tab2.
    """
    if len(tab1) != len(tab2):
        raise ValueError("Les tableaux doivent avoir la même longueur.")

    somme = [a + b for a, b in zip(tab1, tab2)]
    return somme

# Exemple d'utilisation
tableau1 = [2, 8, 5, 3, 11, 7, 9]
tableau2 = [1, 4, 6, 2, 9, 3, 5]

resultat = somme_deux_tableaux(tableau1, tableau2)
print("La somme des deux tableaux est :", resultat)
```

Exemple : Insère l'élément donné à l'index spécifié dans le tableau.

Editeur

```
def inserer_element(tableau, index, element):
    """
        Arguments:
        tableau (list): Le tableau dans lequel insérer l'élément.
        index (int): L'index où insérer l'élément.
        element: L'élément à insérer.
    """
    try:
        tableau.insert(index, element)
        print(f"L'élément {element} a été inséré à l'index {index}.")
    except IndexError:
        print("L'index spécifié est hors des limites du tableau.")

# Exemple d'utilisation
mon_tableau = [1, 2, 3, 4, 5]
inserer_element(mon_tableau, 2, 100)
print(mon_tableau)  # Affiche : [1, 2, 100, 3, 4, 5]
```

Exemple Utilisation de la méthode reverse() des listes : Pour inverser les éléments d'une liste.

Editeur

```
def revverse_T(vec):
    #    Arguments:
#        vec (list): Le vecteur d'entrée.

#    Retourne:
#      Le vecteur est renversé.
    vec.reverse()
    return vec

v = [1, 2, 3, 2, 4, 2, 5, 2]
#v=["A", "B", "C", "D", "E"]
revverse_T(v)
print(v)
```

Exemple

Utilisation de la fonction reversed() : La fonction reversed() renvoie un itérateur qui accède à la séquence donnée dans l'ordre inverse. Voici un exemple avec une liste :

Editeur

```
def revversed_T(vec):
#    Arguments:
#        vec (list): Le vecteur d'entrée.

#    Retourne:
#      Le vecteur est renversé.
    reversed_list= list(reversed(vec))
    return reversed_list

#v = [1, 2, 3, 2, 4, 2, 5, 2]
v=["A", "B", "C", "D", "E"]

t=revversed_T(v)
print(t)
```

Cette boucle affichera les éléments de la liste dans l'ordre inverse : "E", "D", "C", "B", **"A"**

Exemple Ecrire une fonction Python qui vérifie si une sous-liste `[1, 2, 3]` appartient à une liste `[1, 2, 3, 5, 6, 7]` :

Editeur

```
def est_sous_liste(sous_liste, liste_principale):
    """
    Vérifie si une sous-liste est présente dans la liste principale.
    Arguments:
        sous_liste (list): La sous-liste à vérifier.
        liste_principale (list): La liste principale dans laquelle rechercher.
    Retournes:
        bool: True si la sous-liste fait partie de la liste principale, False sinon.
    """
    # Convertit les deux listes en ensembles pour une vérification efficace d'appartenance
    ensemble_sous_liste = set(sous_liste)
    ensemble_liste_principale = set(liste_principale)

    # Vérifie si l'ensemble de la sous-liste est un sous-ensemble de l'ensemble de la liste principale
    return ensemble_sous_liste.issubset(ensemble_liste_principale)

# Exemple d'utilisation
liste_principale = [1, 2, 3, 5, 6, 7]
sous_liste1 = [1, 2, 3]
sous_liste2 = [4, 5, 6]

print(f"Est-ce que {sous_liste1} est une sous-liste de {liste_principale}? {est_sous_liste(sous_liste1, liste_principale)}")
print(f"Est-ce que {sous_liste2} est une sous-liste de {liste_principale}? {est_sous_liste(sous_liste2, liste_principale)}")
```

Fonction et Matrices

Exemple

Ecrire une fonction qui calcul la somme de deux matrices de même dimensions

Editor

```
def somme_matrices(matrice1, matrice2):
    """
    Calcule la somme de deux matrices de même taille.
    :param matrice1: Première matrice (liste de listes)
    :param matrice2: Deuxième matrice (liste de listes)
    :return: Matrice résultante (somme des deux matrices)
    """
    # Vérification que les matrices ont la même taille
    if len(matrice1) != len(matrice2) or len(matrice1[0]) != len(matrice2[0]):
        raise ValueError("Les matrices doivent avoir la même taille.")

    # Initialisation de la matrice résultante
    resultat = [[0] * len(matrice1[0]) for _ in range(len(matrice1))]

    # Calcul de la somme des éléments
    for i in range(len(matrice1)):
        for j in range(len(matrice1[0])):
            resultat[i][j] = matrice1[i][j] + matrice2[i][j]

    return resultat

                                    # Programme Principal
matrice_A = [[1, 2], [3, 4]]
matrice_B = [[5, 6], [7, 8]]

matrice_somme = somme_matrices(matrice_A, matrice_B)
for ligne in matrice_somme:
    print(ligne)
```

Exemple

Ecrire une fonction python qui permet de faire le produit de deux matrices m_1 et m_2.

Editor

```
import numpy as np

def produit_matrices(m1, m2):
    try:
        # Vérification des dimensions des matrices
        if m1.shape[1] != m2.shape[0]:
            raise ValueError("Les dimensions des matrices ne sont pas compatibles pour la
multiplication.")

        # Calcul du produit matriciel
        resultat = np.dot(m1, m2)
        return resultat
    except ValueError as e:
        print(f"Erreur : {e}")
        return None

                                        # Programme Principal
matrice1 = np.array([[1, 2], [3, 4]])
matrice2 = np.array([[5, 6], [7, 8]])

resultat = produit_matrices(matrice1, matrice2)
```

Numpy

Voici quelques écritures classiques avec le module numpy.

Editor

```
import numpy as np
mat = np.matrix ( [[1,2],[3,4]] ) # crée une matrice 2*2
s   = mat.shape          # égale à (nombre de lignes, nombre de colonnes)
l   = mat [0,:]          # retourne la première ligne
c   = mat [:,0]          # retourne la première colonne
iv  = mat.I              # inverse la matrice
mat [:,0] = mat [:,1]    # la première ligne est égale à la seconde
o   = np.ones ( (10,10) ) # crée un matrice de 1 10x10
d   = np.diag (mat)      # extrait la diagonale d'une matrice
dd  = np.matrix (d)      # transforme d en matrice
t   = mat.transpose ()   # obtient la transposée
e   = mat [0,0]          # obtient de première élément
k   = mat * mat          # produit matriciel
k   = mat @ mat          # produit matriciel à partir de Python 3.5
m   = mat * 4            # multiplie la matrice par 4
mx  = np.max (mat [0,:])  # obtient le maximum de la première ligne
s   = np.sum (mat [0,:])  # somme de la première ligne


mat = np.diagflat ( np.ones ( (1,4) ) )
print (mat)  # matrice diagonale
t   =  mat == 0
print (t)    # matrice de booléens
mat [ mat == 0 ] = 4
print (mat)  # ...
print (iv)  # ...
```

Les fonctions récursives

Définition
Une fonction récursive est une fonction qui s'appelle elle-même jusqu'à ce qu'elle ne le fasse plus.

La fonction fn() suivante est une fonction récursive, car elle a un appel à elle-même :

Syntaxe

```
def fn():
    # ...
    fn()
    # ...
```

Pour pouvoir s'arrêter, une fonction récursive doit avoir une condition d'arrêt. Nous devons donc ajouter une instruction if comme celle-ci :

Syntaxe

```
def fn():
    # ...
    if condition:
        # ne s'appelle pas elle-même
    else:
    fn()
    # ...
```

En règle générale, on utilise une fonction récursive pour diviser un problème difficile à résoudre en problèmes plus petits qui sont plus faciles à résoudre.

En programmation, vous trouverez souvent les fonctions récursives utilisées dans les structures de données et les algorithmes comme les arbres, les graphiques et les recherches binaires.

Exemples de fonctions récursives Python
Prenons quelques exemples d'utilisation des fonctions récursives Python.

Exemple
Supposons que nous ayons besoin de développer une fonction de compte à rebours qui compte à rebours à partir d'un nombre spécifié jusqu'à zéro.

Par exemple, si vous appelez la fonction qui compte à rebours à partir de 3, elle affichera la sortie suivante :
3
2
1

Ce qui suit définit la fonction count_down() :

Editor

```
def count_down(nombre_de_depart):
    """ Compte à rebours à partir d'un nombre """
    print(nombre_de_depart)
# Appelez la fonction count_down() maintenant :
count_down(3)
```

Résultat

IDLE Shell

```
# Elle n'affichera que le numéro de départ :
3
```

Pour afficher respectivement dans l'ordre les nombres 3, 2 et 1, nous devons :

1- Tout d'abord, appeler count_down(3) pour afficher 3.
2- Puis, appeler count_down(2) pour afficher 2.
3- Enfin, appeler count_down(1) pour afficher 1.

Pour ce faire, à l'intérieur de la fonction count_down(), nous devons définir une logique pour appeler la fonction count_down() avec les arguments 2 et 1.

Pour ce faire, nous devons rendre la fonction count_down() récursive.

Ce qui suit définit une fonction count_down() récursive et l'appelle en passant le nombre 3 :

Editor

```
def count_down(nombre):
    """ Compte à rebours à partir d'un nombre """
    print(nombre)
    count_down(nombre-1)
count_down(3)
```

Résultat

IDLE Shell

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

La raison en est que count_down()s'appelle indéfiniment jusqu'à ce que le système l'arrête. Par défaut, Python considère que la limite maximale à ne pas dépasser est de 1000 appels.

Nous pouvons vérifier cette limite à l'aide de sys.getrecursionlimit() :

IDLE Shell

```
>>> import sys
>>>
>>> print(sys.getrecursionlimit())
1000
```

Et au besoin, nous pouvons le modifier avec sys.setrecursionlimit() :

IDLE Shell

```
>>> import sys
>>>
>>> sys.setrecursionlimit(1500)
>>> print(sys.getrecursionlimit())
1500
```

Attention, si cette limite existe, c'est qu'il y a une bonne raison.

Revenons-en à notre compte à rebours, nous devons l'arrêter une fois le nombre zéro atteint. Pour ce faire, ajoutons une condition comme celle-ci :

Editor

```
def count_down(nombre):
    """ Compte à rebours à partir d'un nombre """
    print(nombre)
    # Appeler la fonction count_down si le nombre suivant à décompter est positif
    nombre_suivant = nombre - 1
    if nombre_suivant > 0:
        count_down(nombre_suivant)
count_down(3)
```

Résultat

IDLE Shell

```
3
2
1
```

Dans cet exemple, la fonction count_down() s'appelle uniquement lorsque le nombre suivant qui doit être décompté est supérieur à zéro. En d'autres termes, si sa valeur est zéro, la fonction arrête de s'appeler.

Exemple

Utiliser une fonction récursive pour calculer la somme d'une séquence

Supposons que nous ayons besoin de calculer la somme d'une séquence, par exemple de 1 à 100. Un moyen simple de le faire est d'utiliser une boucle for avec la fonction range() :

Editor

```
def somme(n):
    total = 0
    for i in range(n+1):
        total += i
    return total
result = somme(100)
print(result)
```

Résultat

IDLE Shell

```
5050
```

Pour faire la même chose en utilisant la récursivité, nous pouvez calculer la somme de la séquence de 1 à n comme suit :

Editor

```
somme(n) = n + somme(n-1)
somme(n-1) = n-1 + somme(n-2)
…
somme(0) = 0
```

La fonction somme() continuera de s'appeler tant que son argument sera supérieur à zéro.

Ce qui suit définit la version récursive de la fonction somme() :

Editor

```
def somme(n):
    if n > 0:
        return n + somme(n-1)
    return 0
result = somme(100)
print(result)
```

Comme vous pouvez le voir, la fonction récursive est beaucoup plus courte et plus lisible.

Si vous utilisez l'opérateur ternaire, le code de la fonction somme() sera encore plus concis :

Editor

```
def somme(n):
    return n + somme(n-1) if n > 0 else 0
result = somme(100)
print(result)
```

Exemple

Fonction pour faire la somme de chiffre constituant un entier n utilisant la récursivité

Programme Python

Editor

```
def sum_of_digit( n ):
   if n == 0:
      return 0
   return (n % 10 + sum_of_digit(int(n / 10)))

# Programme globale
n = 12345
result = sum_of_digit(n)
print("Somme des chiffres dans ",n,"est", result)
```

Exemple

Ecrire un programme récursif en python pour calculer le factoriel de n

Algorithme

```
entier factoriel(entier n)
Debut
   si (n > 0)
      retouner (n * factoriel(n-1));
   retouner 1;
fin

#Programme principale
Debut
entier result = factoriel(100);
imprimer(result);
fin
```

Programme Python

Editor

```
def factoriel(n):
   if  n > 0:
      return (n * factoriel(n-1))
   return 1
result = factoriel(100)
print(result)
```

Exemple
Ecrire une fonction récursif en python pour calculer la suite de 1 à n

Algorithme

```
entier somme(entier n)
Debut
   si (n > 0)
      retourner (n + somme(n-1));
   retourner 0;
fin

#Programme principale
Debut
entier result = somme(100);
imprimer (result);
fin
```

Programme Python

Editor

```
def somme(n):
   if n > 0:
      return (n + somme(n-1))
   return 0
result = somme (100)
print(result)
```

Exemple
Ecrire une fonction récursif en python pour calculer la puissance x^n

Algorithme

```
reel puissance( reelt x, entier n)
Debut
      si (n>0)
            retourner(x*puissance(x,n-1));
         retourner 1;
fin
#Programme principale
Debut
         reel x;
         entier n;
         imprimer("Entrez n : ");
         lire(n);
         imprimer("Entrez x : ");
         lire(x);
         reel result = puissance( x,n);
         imprimer (result);
fin
```

Programme Pyhthon

Python

Editor

```
def puissance(x,n):
   if n > 0:
      return (x*puissance(x,n-1))
   return 1

n = input("Entrez n : ")
x= input ("Entrez x : ")
result = puissance(x,n)
print(result)
```

Exemple

Calculez la somme de la Suite s=1/1+1/2+1/3+…..1/n, en écrivant une fonction récursif en python.

Algorithme

```
entier somme(entier n)
Debut
   si (n > 0)
      retourner (1/n + somme(n-1));
   retourner 0;
fin

#Programme principale
Debut
int result = somme(10);
imprimer(result);
fin
```

Programme Python

Editor

```
def somme_suite(n):
   if n > 0:
      return (1/n + somme_suite(n-1))
   return 0
result = somme_suite (100)
print(result)
```

Exemple
Calcul la somme de la Suite s=x/1+x/2+x/3+…..x/n, en écrivant une fonction récursif en python.

Algorithme

```
reel som_suite( reel x, entier n)
Debut
si(n>0)
        retourner(x/n+som_suite(x,n-1));
        retourner 1;
fin
```

```
#Programme principale
Debut
        reel x;
        entier n;
        imprimer("Entrez n : ");
        lire(n);
        imprimer("Entrez x : ");
        lire(x);
        reel result = som_suite( x,n);
        imprimer(result);
fin
```

Programme Python

Editor

```
def som_suite(x, n):
    if n > 0:
        return (x / n + som_suite(x, n - 1))
    return 1

#   Lecture de  n et x
n = int(input("Enter n: "))
x = float(input("Enter x: "))

# Calcul  la  valeur de som_suite(x, n)
result = som_suite(x, n)

# imprime le résyltat
print(f"La valeur de som_suite({x}, {n})  est {result:.2f}")
```

Exemple
Soit une chaine de caractères, écrire un algorithme récursif permettant de déterminer sa longueur

Programme Python

Editor

```
def longueur(ch):

   if not ch:
      return 0
   else:
      return 1+longueur(ch[1:])
                                        #Programme Principal
 ch = "Bonjour Azzeddine"
print(longueur(ch))
```

Exemple
Ecrire une fonction récursive « Binaire » permettant d'imprimer à l'écran la représentation binaire d'un nombre N (voir exemple en face).

Programme Python

Editor

```
def binaire(N):
   if N == 0:
      return []
   return binaire(N//2)+[N % 2]
                                        #Programme Principal
print(binaire(13))
```

Exemple
La suite de Fibonacci est définie comme suit :

$$F_n = \begin{cases} 1 & n < 2 \\ F_{n-1} + F_{n-2} & sinon \end{cases}$$

Ecrire un programme récursif calculant Fib(n)

Programme Python

Editor

```
def Fib(n):
   if n <= 2:
      return 1
   return Fib(n-1)+Fib(n-2)
                                        #Programme Principal
 print(Fib(4))
```

Exemple
Soit la suite définie par :

$$U_n = \begin{cases} 1 & n < 2 \\ 3U_{n-1} + U_{n-2} & sinon \end{cases}$$

Ecrire un programme récursif permettant de calculer le nième terme de la suite;

Programme Python

Editor

```
def Suite(n):
    if n <= 2:
        return 1
    return 3*Suite(n-1)+Suite(n-2)
                                    #Programme Principal
print(Suite(4))
```

Exemple
Un nombre est pair si est impair, et un nombre N est impair si est pair.
Ecrire deux fonctions récursives mutuelles pair(N) et impair(N) permettant de savoir si un nombre N est pair et si un nombre N est impair.

Programme Python

Editor

```
def Pair(N):
    if N == 1:
        return False
    return Impair(N-1)

def Impair(N):
    if N == 1:
        return True
    return Pair(N-1)
```

Exemple
Soit un tableau X de N entiers, écrire une fonction récursive simple permettant de déterminer le maximum du tableau

Programme Python

Editor

```
def maximum(T):
    if len(T) == 1:
        return T[0]

    # principe de la recherche dichotomique
    m = len(T)//2
    max1 = maximum(T[:m])
    max2 = maximum(T[m:])
    if max1 > max2:
        return max1
    return max2
```

Programme Python

Editor

```
def somme_suite_x(x,n):
    if n > 0:
        return (x/n + somme_suite_x(x,n-1))
    return 0
n = input("Entrez n : ")
x= input ("Entrez x : ")

result = somme_suite x(x,n)
print(result)
```

Fonction récursif et tableau linéaire

Exemple
Calcul la somme des éléments d'un tableau linéaire tab(n), en écrivant une fonction récursif en python.

Algorithme

```
entier somme(entier n, entier tab[])
Debut
   si (n > 0)
      retourner (tab[n-1] + somme(n-1, tab));
   retourner 0;
fin
```

#Programme principale

```
Debut
entier T[5]={1,2, 3, 4, 5};
entier result = somme(5,T);
imprimer(result);
fin
```

Programme Python

Editor

```
def somme_tableau_recursive(tab):
   if len(tab) == 0:
      return 0
   else:
      return tab[0] + somme_tableau_recursive(tab[1:])

# Exemple d'utilisation avec une liste :
T = [1, 2, 3, 4, 5]
resultat = somme_tableau_recursive(T)
print(f"La somme des éléments du tableau est : {resultat}")
```

Exemple
Calcul la somme des éléments d'un tableau linéaire tab(n), en écrivant une fonction récursif en python.

Algorithme

```
entier produit(entier n, entier tab[])
Debut
   if (n > 0)
      retourner (tab[n-1] * produit(n-1, tab));
   retourner 1;
fin
```

#Programme principale
Debut
entier T[5]={1,2, 3, 4, 5};
entier result = produit(5,T);
printf(result);
fin

Programme Python

Editor

```
def produit_recursive(T, i=0):

    """
    Calcule le produit des éléments d'un tableau de manière récursive.

  Args:
     #tableau (list): Le tableau d'entiers.
     #index (int): L'index actuel dans le tableau (par défaut 0).

  Returns:
     int: Le produit des éléments du tableau.
    """

    if i >= len(T):
        return 1  # Condition d'arrêt : fin du tableau

    # Appel récursif en multipliant l'élément actuel avec le produit des éléments restants
    return T[i] * produit_recursive(T, i + 1)
                                        #Programme principale
tableau = [2, 3, 4, 5]
resultat = produit_recursive(tableau)
print(f"Le produit des éléments du tableau est : {resultat}")
```

Exemple
Ecrire une onction récursive pour compter les occurrences d'un élément dans un tableau en python.

Algorithme

// Fonction récursive pour compter les occurrences d'un élément dans un tableau
entier compterOccurrences(entier tab[], entier n, entier element, entier i)
Debut
    // Cas de base : si l'index dépasse la taille du tableau, retourner 0
    si (i >= n) {
        retourner 0;
    }
    // Si l'élément à l'index courant est égal à l'élément recherché, incrémenter le compteur
    entier compteur = (tab[i] == element) ? 1 : 0;
    // Appel récursif pour le reste du tableau
    retourner compteur + compterOccurrences(tab, n, element, i+=1);
fin

#Programme principale
Debut

```
    entier tableau[] = {2, 1, 2, 4, 4, 3, 9, 9, 8, 9};
    entier n = sizeof(tableau) / sizeof(tableau[0]);
    entier elementRecherche = 9; // Remplacez par l'élément que vous souhaitez rechercher
    entier occurrences = compterOccurrences(tableau, n, elementRecherche, 0);
    imprimer("Le nombre d'occurrences de %d dans le tableau est :", elementRecherche,
occurrences);
```

fin

Programme Python

Editor

```
def compter_occurrences_recursive(tab, element, i=0, occurrences=0):
    """
    Compte récursivement le nombre d'occurrences de l'élément dans le tableau.
   :param tableau: Le tableau dans lequel chercher les occurrences.
   :param element: L'élément dont on veut compter les occurrences.
   :param index: L'index actuel dans le tableau (défaut : 0).
   :param occurrences: Le nombre d'occurrences trouvé jusqu'à présent (défaut : 0).
   :return: Le nombre total d'occurrences de l'élément dans le tableau.
    """
    if i < len(tab):
        if tab[i] == element:
            occurrences += 1
        return compter_occurrences_recursive(tab, element, i + 1, occurrences)
    else:
        return occurrences
```

#Programme principale

```
tableau = [2, 1, 2, 4, 4, 3, 9, 9, 8, 9]
element_recherche = 1
nombre_occurrences = compter_occurrences_recursive(tableau, element_recherche)
print(f"L'élément {element_recherche} apparaît {nombre_occurrences} fois dans le tableau.")
```

Exemple
Un tableau X est trié par ordre croissant si $x(i) \leq x(i+1), \forall i$, écrire un algorithme récursif permettant de vérifier qu'un tableau X est trié ou non

Programme Python

Editor

```
def Esttrier(T):
    if len(T) == 0 or len(T) == 1:
        return True
    if T[0] <= T[1]:
        return Esttrier(T[1:])
    return False
                                    #Programme Principal
T = [1, 2, 3, 4, 4, 5, 7, 8]
print(Esttrier(T))
```

Exemple
Fonction récursive pour calculer la somme de deux tableaux en python

Programme Python

Editor

```
def somme_recursive(tableau1, tableau2, i=0):
    """
    Calcule la somme des éléments correspondants de deux tableaux de même longueur.
    :param tableau1: Premier tableau
    :param tableau2: Deuxième tableau
    :param index: Index courant (défaut : 0)
    :return: Somme des éléments correspondants
    """
    if i>= len(tableau1) or i >= len(tableau2):
        # Condition d'arrêt : si l'index dépasse la longueur des tableaux
        return 0
    else:
        # Appel récursif pour la somme des éléments correspondants
        return tableau1[index] + tableau2[index] + somme_recursive(tableau1, tableau2, i + 1)

                                    #Programme principale
tableau_a = [1, 2, 3, 4]
tableau_b = [5, 6, 7, 8]
resultat = somme_recursive(tableau_a, tableau_b)
print(f"La somme des éléments correspondants est : {resultat}")
```

Exemple

Un mot est un palindrome si on peut le lire dans les deux sans de gauche à droite et de droite à gauche. Exemple KAYAK est un palindrome. Ecrire une fonction récursive permettant de vérifier si un mot est palindrome.

Programme Python

Editor

```
def palindrome(ch):
    if len(ch) == 1 or len(ch)==0:
        return True
    if ch[0] == ch[-1]:
        return palindrome(ch[1:len(ch)-1])
    return False

ch = "KAYAK"
print(palindrome(ch))
```

Exemple

Soit un tableau d''entiers contenant des valeurs 0 ou bien 1. On appel composante connexe une suite contigue de nombres égaux à 1. On voudrait changer la valeur de chaque composante connexe de telle sorte que la première composante ai la valeur 2 la deuxième ai la valeur 3, la 3ème ait la valeur 4 et ainsi de suite. Réaliser deux fonctions :

La première fonction n'est pas récursive et a pour rôle de chercher la position d'un 1 dans un tableau.

La deuxième fonction est récursive. Elle reçoit la position d'un 1 dans une séquence et propage une valeur x à toutes les valeurs 1 de la composante connexe.

Programme Python

Editor

```
# Trouver l'indice du premier 1
def trouver(T):
    for i in range(len(T)):
        if T[i] == 1:
            return i
    return None

# Fonction principale
def propage(T, x, val):
    i = trouver(T)
    if i is None:
        return
    T[i] = val
    if (len(T)-i+1) > 2:
        if T[i+1] == 1:
            propage(T, x, val+1)
        else:
            propage(T, x, x)

# tester la fonction
T = [0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1]
x = 2
propage(T, x, x)
print(T)
```

Fonction récursif et Matrices

Exemple

Soit une image binaire représentés dans une matrice à 2 dimension. Les éléments m[i][j] sont dits pixels et sont égaux soit à 0 soit à 1. Chaque groupement de pixels égaux à 1 et connectés entre eux forment

une composante connexe(figure). L'objectifs est de donner une valeur différente de 1 à chaque composante (2 puis 3 puis 4 etc.)

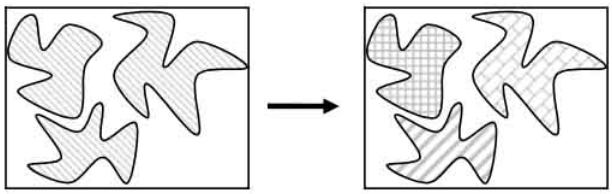


Image binaire Image étiquetée

Ecrire une fonction récursive propager permettant de partir d'un point (i,j) situé à l'intérieur d'une composante connexe et de propager une étiquette T à tous les pixels situés à l'intérieur de la composante.

Ecrire une fonction etiqueter permettant d'affecter une étiquette différente à chaque composante connexe.

Programme Python

Editor

```
def propager(M, i, j, val):
    if M[i][j] == 0:
        return
    M[i][j] = val
    if((i-1) >= 0 and M[i-1][j] == 1):  # l'élément en haut
        propager(M, i-1, j, val)
    if((i+1) < len(M) and M[i+1][j] == 1):  # l'élément en bas
        propager(M, i+1, j, val)
    if((j-1) < len(M) and M[i][j-1] == 1):  # l'élément a gauche
        propager(M, i, j-1, val)
    if((j+1) < len(M) and M[i][j+1] == 1):  # l'élément a droite
        propager(M, i, j+1, val)

 def etiqueter(M):
    L, C = len(M), len(M[0])
    val = 2
    for i in range(L):
        for j in range(C):
            if(M[i][j] == 1):
                propager(M, i, j, val)
                val += 1
                                                    #Programme Prinipal
M = [[0, 0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 0], [0, 1, 1, 0]]
print(M)
etiqueter(M)
print(M)
atrice.")
else:
    print(f"L'élément {element_recherche} n'a pas été trouvé dans la matrice.")
```

Exemple

Fonction récursive pour trouver un élément dans une matrice en python

Programme Python

Editor

```
def recherche_element(matrice, element, ligne=0, colonne=0):
    """
    Recherche récursive d'un élément dans une matrice.
    :param matrice: La matrice à rechercher
    :param element: L'élément à trouver
    :param ligne: L'indice de la ligne actuelle
    :param colonne: L'indice de la colonne actuelle
    :return: True si l'élément est trouvé, False sinon
    """
    # Vérification des limites de la matrice
    if ligne >= len(matrice) or colonne >= len(matrice[0]):
        return False

    # Vérification de l'élément actuel
    if matrice[ligne][colonne] == element:
        return True

    # Appel récursif pour explorer les cases voisines
    return (recherche_element(matrice, element, ligne + 1, colonne) or
            recherche_element(matrice, element, ligne, colonne + 1))

# Exemple d'utilisation
ma_matrice = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

element_recherche = 5
if recherche_element(ma_matrice, element_recherche):
    print(f"L'élément {element_recherche} a été trouvé dans la matrice.")
else:
    print(f"L'élément {element_recherche} n'a pas été trouvé dans la matrice.")
```

Exemple

Fonction récursive pour la somme des éléments de la diagonale d'une matrice en python

Programme Python

Editor

```
def somme_diagonale(matrice, n, i=0, j=0):
    """
    Calcule récursivement la somme des éléments de la diagonale d'une matrice.
    :param matrice: La matrice (représentée sous forme de liste de listes).
    :param n: La taille de la matrice (nombre de lignes ou de colonnes).
    :param i: L'indice de ligne (défaut : 0).
    :param j: L'indice de colonne (défaut : 0).
    :return: La somme des éléments de la diagonale.
    """
    if i == n:
        return 0  # Condition d'arrêt : nous avons parcouru toutes les lignes

    # Si l'élément est sur la diagonale, ajoute-le à la somme
    if i == j:
        return matrice[i][j] + somme_diagonale(matrice, n, i + 1, j + 1)
    else:
        return somme_diagonale(matrice, n, i + 1, j)  # Sinon, passe à la ligne suivante

# Exemple d'utilisation
matrice_exemple = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

taille_matrice = len(matrice_exemple)
somme = somme_diagonale(matrice_exemple, taille_matrice)

print(f"La somme des éléments de la diagonale est : {somme}")
```

Exemple

Si nous avons deux matrices A et B, la matrice résultante R est définie comme suit :

$$R_{ij}=A_{ij}+B_{ij}$$

Maintenant, voici une fonction récursive qui additionne deux matrices en Python :

Programme Python

Editor

```
def addition_matrices_recursive(A, B):
    """
    Calcule la somme de deux matrices A et B (de même taille).
    """
    rows, cols = len(A), len(A[0])
    result = [[0] * cols for _ in range(rows)]  # Initialisation de la matrice résultante

    def add_recursive(i, j):
        if i >= rows:
            return  # Condition d'arrêt : toutes les lignes ont été traitées
        if j >= cols:
            add_recursive(i + 1, 0)  # Passer à la ligne suivante
            return

        result[i][j] = A[i][j] + B[i][j]
        add_recursive(i, j + 1)  # Passer à la colonne suivante

    add_recursive(0, 0)  # Appel initial
    return result
                                        # Programme Principal
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
resultat = addition_matrices_recursive(A, B)
print("Matrice A :", A)
print("Matrice B :", B)
print("Somme des matrices A et B :", resultat)

    print("Produit des matrices :")
    print(resultat)
```

Exemple

Fonction récursive pour multiplier la matrice A par la matrice B en python

Programme Python

Editor

```
def multiply_matrices_recursive(matrix_a, matrix_b):
    """
    Multiplies two matrices recursively.
    Args:
        matrix_a (list of lists): First matrix.
        matrix_b (list of lists): Second matrix.
    Returns:
        list of lists: Resultant matrix.
    """
    # Vérification des dimensions des matrices
    rows_a, cols_a = len(matrix_a), len(matrix_a[0])
    rows_b, cols_b = len(matrix_b), len(matrix_b[0])

    if cols_a != rows_b:
        raise ValueError("Le nombre de colonnes de la matrice A doit être égal au nombre de lignes de la matrice B.")

    # Initialisation de la matrice résultante
    result_matrix = [[0] * cols_b for _ in range(rows_a)]

    # Fonction récursive pour calculer les éléments de la matrice résultante
    def multiply_recursive(i, j, k):
        if i >= rows_a:
            return
        if j >= cols_b:
            return multiply_recursive(i + 1, 0, 0)
        if k >= cols_a:
            return multiply_recursive(i, j + 1, 0)

        result_matrix[i][j] += matrix_a[i][k] * matrix_b[k][j]
        multiply_recursive(i, j, k + 1)

    multiply_recursive(0, 0, 0)
    return result_matrix

# Exemple d'utilisation
matrix_a = [[1, 2], [3, 4]]
matrix_b = [[5, 6], [7, 8]]
result = multiply_matrices_recursive(matrix_a, matrix_b)
print(result)
```

Introduction aux graphiques en Python avec matplotlib.pyplot

Tracé de courbes

Pour tracer des courbes, **Python** n'est pas suffisant et nous avons besoin de la bibliothèque **Matplotlib**.
Il existe deux styles de programmation :

- le style « pyplot » qui utilise directement des fonctions du module pyplot ;
- le style « Orienté Objet (OO) » qui est recommandé dans la documentation de **Matplotlib**.

Nous verrons rapidement le module "pyplot" de la bibliothèque "matplotlib" qui nous permettra de faire des graphes.

Le module matplotlib

Il s'agit sûrement de l'une des bibliothèques python les plus utilisées pour représenter des graphiques en 2D.
Le module pyplot de matplotlib est l'un de ses principaux modules.
Il regroupe un grand nombre de fonctions qui servent à créer des graphiques et les personnaliser (travailler sur les axes, le type de graphique, sa forme et même rajouter du texte). Avec lui, nous avons déjà de quoi faire de belles choses.

Remarque

Le fonctionnement de matplotlib est très semblable à celui de matlab. Le fonctionnement, et même les noms des fonctions par exemple, sont quasiment toujours les mêmes.

Installation sous Linux

C'est sans doute sous Linux que matplotlib est le plus simple à installer. Il suffit d'utiliser son gestionnaire de paquets (en ligne de commande ou en graphique). Voici quelques exemples de commande d'installation.

```
sudo pacman -S python-matplotlib          # Sous Arch linux
sudo apt-get install python-matplotlib    # Sous Ubuntu
```

Utiliser le gestionnaire de paquets est la méthode la plus simple mais nous pouvons également utiliser le programme **pip** (qui est souvent installé par défaut) en entrant cette commande dans un terminal. Nous commençons par le mettre à jour avec la première ligne, avant d'installer matplotlib avec la seconde.

```
python3 -m pip install --user -U --upgrade pip
python3 -m pip install --user -U matplotlib
```

Il se chargera d'installer toutes les dépendances nécessaires au bon fonctionnement de matplotlib.

Installation sous Windows

Sous Windows, nous pouvons également utiliser pip pour installer matplotlib. Il nous suffit donc d'ouvrir un terminal et d'entrer ces deux commandes. La première commande permet de mettre à jour pip et la seconde installe matplotlib.

py -m pip install --user -U --upgrade pip
py -m pip install --user -U matplotlib

Voir l’installation de la bibliothèque "matplotlib" avec l’invite de commande en mode Administrateur page suivante

Administrateur : Invite de commande

```
Microsoft Windows [version 10.0.19045.4046]
(c) Microsoft Corporation. Tous droits réservés.

C:\Windows\system32>py -m pip install --user -U --upgrade pip
Requirement already satisfied: pip in c:\users\musta\appdata\local\programs\python\python312\lib\site-packages (23.3.1)
Collecting pip
  Downloading pip-24.0-py3-none-any.whl.metadata (3.6 kB)
Downloading pip-24.0-py3-none-any.whl (2.1 MB)
   ---------------------------------------- 2.1/2.1 MB 1.4 MB/s eta 0:00:00
Installing collected packages: pip
  WARNING: The scripts pip.exe, pip3.12.exe and pip3.exe are installed in 'C:\Users\Musta\AppData\Roaming\Python\Python312\Scripts' which is not on
PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pip-24.0

[notice] A new release of pip is available: 23.3.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Windows\system32>py -m pip install --user -U matplotlib
Collecting matplotlib
  Downloading matplotlib-3.8.3-cp312-cp312-win_amd64.whl.metadata (5.9 kB)
Collecting contourpy>=1.0.1 (from matplotlib)
  Downloading contourpy-1.2.0-cp312-cp312-win_amd64.whl.metadata (5.8 kB)
Collecting cycler>=0.10 (from matplotlib)
  Downloading cycler-0.12.1-py3-none-any.whl.metadata (3.8 kB)
Collecting fonttools>=4.22.0 (from matplotlib)
  Downloading fonttools-4.48.1-cp312-cp312-win_amd64.whl.metadata (162 kB)
     ---------------------------------------- 162.2/162.2 kB 1.2 MB/s eta 0:00:00
Collecting kiwisolver>=1.3.1 (from matplotlib)
  Downloading kiwisolver-1.4.5-cp312-cp312-win_amd64.whl.metadata (6.5 kB)
Requirement already satisfied: numpy<2,>=1.21 in c:\users\musta\appdata\local\programs\python\python312\lib\site-packages (from matplotlib) (1.26.2)
Collecting packaging>=20.0 (from matplotlib)
  Downloading packaging-23.2-py3-none-any.whl.metadata (3.2 kB)
Collecting pillow>=8 (from matplotlib)
  Downloading pillow-10.2.0-cp312-cp312-win_amd64.whl.metadata (9.9 kB)
Collecting pyparsing>=2.3.1 (from matplotlib)
  Downloading pyparsing-3.1.1-py3-none-any.whl.metadata (5.1 kB)
Collecting python-dateutil>=2.7 (from matplotlib)
  Downloading python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
     ---------------------------------------- 247.7/247.7 kB 461.1 kB/s eta 0:00:00
Collecting six>=1.5 (from python-dateutil>=2.7->matplotlib)
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Downloading matplotlib-3.8.3-cp312-cp312-win_amd64.whl (7.6 MB)
   ---------------------------------------- 7.6/7.6 MB 1.2 MB/s eta 0:00:00
Downloading contourpy-1.2.0-cp312-cp312-win_amd64.whl (187 kB)
   ---------------------------------------- 187.7/187.7 kB 1.9 MB/s eta 0:00:00
Downloading cycler-0.12.1-py3-none-any.whl (8.3 kB)
Downloading fonttools-4.48.1-cp312-cp312-win_amd64.whl (2.2 MB)
   ---------------------------------------- 2.2/2.2 MB 1.5 MB/s eta 0:00:00
Downloading kiwisolver-1.4.5-cp312-cp312-win_amd64.whl (56 kB)
   ---------------------------------------- 56.0/56.0 kB ? eta 0:00:00
Downloading packaging-23.2-py3-none-any.whl (53 kB)
   ---------------------------------------- 53.0/53.0 kB ? eta 0:00:00
Downloading pillow-10.2.0-cp312-cp312-win_amd64.whl (2.6 MB)
   ---------------------------------------- 2.6/2.6 MB 1.2 MB/s eta 0:00:00
Downloading pyparsing-3.1.1-py3-none-any.whl (103 kB)
   ---------------------------------------- 103.1/103.1 kB 3.0 MB/s eta 0:00:00
Installing collected packages: six, pyparsing, pillow, packaging, kiwisolver, fonttools, cycler, contourpy, python-dateutil, matplotlib
  WARNING: The scripts fonttools.exe, pyftmerge.exe, pyftsubset.exe and ttx.exe are installed in
'C:\Users\Musta\AppData\Roaming\Python\Python312\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed contourpy-1.2.0 cycler-0.12.1 fonttools-4.48.1 kiwisolver-1.4.5 matplotlib-3.8.3 packaging-23.2 pillow-10.2.0 pyparsing-3.1.1
python-dateutil-2.8.2 six-1.16.0

C:\Windows\system32>py -m pip install --user -U matplotlibpy -m pip install --user -U matplotlib

Usage:
  C:\Users\Musta\AppData\Local\Programs\Python\Python312\python.exe -m pip install [options] <requirement specifier> [package-index-options] ...
  C:\Users\Musta\AppData\Local\Programs\Python\Python312\python.exe -m pip install [options] -r <requirements file> [package-index-options] ...
  C:\Users\Musta\AppData\Local\Programs\Python\Python312\python.exe -m pip install [options] [-e] <vcs project url> ...
  C:\Users\Musta\AppData\Local\Programs\Python\Python312\python.exe -m pip install [options] [-e] <local project path> ...
  C:\Users\Musta\AppData\Local\Programs\Python\Python312\python.exe -m pip install [options] <archive url/path> ...

no such option: -m
C:\Windows\system32>
```

Création d'une courbe

Utilisation de **plot()**

L'instruction **plot()** permet de tracer des courbes qui relient des points dont les abscisses et ordonnées sont fournies en arguments.

Syntaxe :

```
plt.plot(x_data, y_data, options)
```

Explications des éléments de la syntaxe :

1. **x_data:** Une liste contenant les valeurs des données sur l'axe des abscisses.
2. **y_data:** Une liste contenant les valeurs des données sur l'axe des ordonnées.
3. **options:** Paramètres optionnels qui permettent de personnaliser le tracé du graphique. Ces options incluent :
   - **color:** Couleur de la ligne ou des points.
   - **linestyle:** Style de ligne (par exemple, '-' pour une ligne solide, '--' pour une ligne en pointillés).
   - **marker:** Type de marqueur à utiliser pour les points ('o' pour un cercle, 's' pour un carré, etc.).
   - **label:** Étiquette de légende pour le tracé (utilisé avec plt.legend()).
   - **xlabel:** Étiquette de l'axe des abscisses.
   - **ylabel:** Étiquette de l'axe des ordonnées.
   - **title:** Titre du graphique.
   - **plt.show():** Affiche le graphique à l'écran.

Remarque : pour utiliser, il faut importé le module « pyplot ».

import matplotlib.pyplot **as** plt

Example:

Editor :

```
import  matplotlib.pyplot as plt

plt.show()
plt.close()
```

1. close sert tout simplement à fermer la fenêtre qui s'est ouverte avec show. Lorsque nous appuyons sur la croix de notre fenêtre, celle-ci se ferme également. Néanmoins, il vaut mieux toujours utiliser close.

Editor :

```
import  matplotlib.pyplot as plt

plt.plot()
plt.show()
plt.close()
```

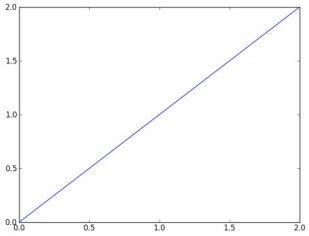
Tracer des lignes brisées

Pour tracer des lignes, nous devons utiliser la commande plot du module pyplot. Elle peut ne prendre aucun argument comme nous venons de le voir, mais c'est bien avec des arguments qu'elle est utile. En effet, si nous lui passons une liste [a, b, c] en argument, elle reliera le points A(0, a) au point B(1, b) et ce point B au point C(2, c). En fait, nous fournissons les ordonnées dans une liste, et les abscisses, elles, sont automatiquement générées et vont de 0 à len(liste) - 1. Ainsi, le code suivant…

Editor :

```
import matplotlib.pyplot as plt

plt.plot([0, 1, 2])
plt.show()
plt.close()
```

Le résultat permet d'obtenir la droite passant par les points A(0, 0), B(1, 1) et C(2, 2).



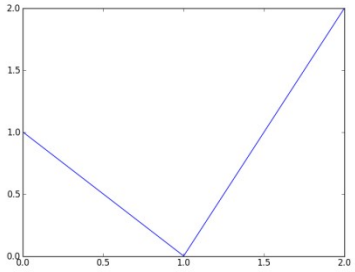
Exemple

Tracer la droite passant par les points A(0, 1), B(1, 0) et C(2, 2), puis l'enregistrer sous le nom "Graphe1.png"

Editor :

```
import matplotlib.pyplot as plt

plt.plot([1, 0, 2])
plt.show()
plt.savefig("Graphe1.png")
plt.close()
```

Résultat



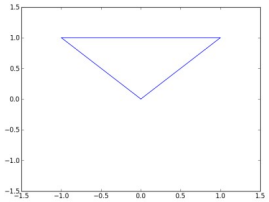
Exemple :

Dessiner un triangle en utilisant les coordonnées suivant :

Editor :

```
import matplotlib.pyplot as plt

x = [0, 1, -1, 0]
y = [0, 1, 1, 0]
plt.plot(x, y)
plt.show()
plt.close()
```

Résultat

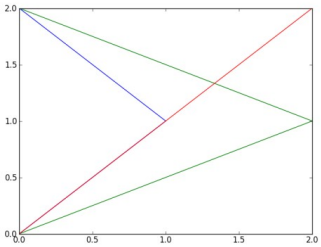


(0,0), (1,1), (-1,1), (0,0)

Editor :

```
import matplotlib.pyplot as plt

x = [0, 1, 0]
y = [0, 1, 2]
x1 = [0, 2, 0]
y1 = [2, 1, 0]
x2 = [0, 1, 2]
y2 = [0, 1, 2]

plt.plot(x, y)
plt.plot(x1, y1)
plt.plot(x2, y2)
plt.show()
plt.close()
```

Le résultat

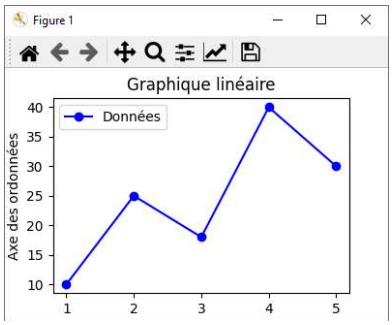


Exemple : Tracé d'un graphique linéaire simple

Editor :

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 25, 18, 40, 30]

plt.plot(x, y, marker='o', linestyle='-', color='b', label='Données')
plt.xlabel('Axe des abscisses')
plt.ylabel('Axe des ordonnées')
plt.title('Graphique linéaire')
plt.legend()
plt.show()
plt.close()
```

Le résultat :

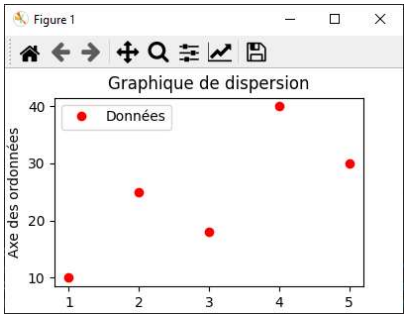


Exemple: Tracé d'un graphique de dispersion

Editor :

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 25, 18, 40, 30]

plt.plot(x, y, marker='o', linestyle='None', color='r', label='Données')
plt.xlabel('Axe des abscisses')
plt.ylabel('Axe des ordonnées')
plt.title('Graphique de dispersion')
plt.legend()
plt.show()
plt.close()
```

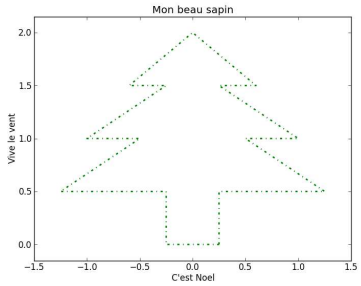
Le résultat :



Editor :

```
import matplotlib.pyplot as plt

x = [0.25, 0.25, 1.25, 0.5, 1, 0.25, 0.6, 0, -0.6, -0.25, -1, -0.5, -1.25, -0.25, -0.25, 0.25]
y = [0, 0.5, 0.5, 1, 1, 1.5, 1.5, 2, 1.5 , 1.5, 1, 1, 0.5, 0.5, 0, 0]
plt.plot(x, y, '-.', color = "green", lw = 2)
plt.title("Mon beau sapin")
plt.axis('equal')
plt.xlabel("C'est Noel")
plt.ylabel("Vive le vent")
plt.show()
plt.close()
```

Le résultat :



Tracer les fonctions

Pour tracer des fonctions, nous allons relier des points dont nous savons qu'ils appartiennent à la courbe. Par exemple, pour tracer la fonction cosinus sur l'intervalle [0, 2π], nous allons relier les point A(0, cos(0)) et B(2π, cos(2π)).

Pour obtenir la courbe (en tout cas l'approcher), nous allons relier des points très proche qui appartiennent à la courbe. Par exemple, nous pouvons couper l'intervalle en 100 et donc relier 101 points plutôt que deux points. On appelle cela, subdiviser l'intervalle.

Notre subdivision sera à pas constant (le pas δ est l'espace entre deux points de la subdivision). Le pas est donc égal à la longueur de l'intervalle divisé par le nombre «n » de points de la subdivision.

$$ \delta = \frac{x{max} - x{min}}{n}. $$

Ainsi pour dessiner la fonction cosinus :

On coupe l'intervalle en « n » tranches (on choisit par exemple n=100) ;

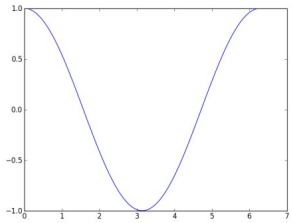
Le pas est donc $\delta = \frac{2\pi}{100}$;

On relie les points de la subdivision.

Editor :

```
import matplotlib.pyplot as plt
from math import cos, pi

x = []
y = [] # On a créé deux listes vides pour contenir les abscisses et les ordonnées
pas = 2 * pi / 100
abscisse = 0 # L'abscisse de départ est 0
for k in range(0, 101): # On veut les points de 0 à 100
    x.append(abscisse)
    y.append(cos(abscisse)) # On rajoute l'abscisse et son image par la fonction cos aux listes
    abscisse += pas # on augmente abscisse de pas pour passer au point suivant
plt.plot(x, y)
plt.show()
plt.close()
```

Le résultat :

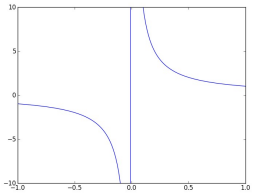


Les fonctions discontinues ou à valeurs interdites

Cependant, la méthode précédente peut présenter quelques problèmes dans des cas particuliers. Pour bien le voir, essayons de dessiner la fonction inverse (x→1/x) sur l'intervalle [-1, 1].

Editor :

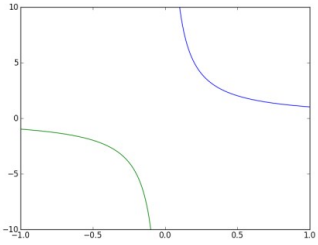
```
import matplotlib.pyplot as plt
#On dessine la fonction y=f(x)=1/x
a = -1# interval [a,b]
b = 1
x = []
y = []
pas = (b - a) / 200
abscisse = a
for k in range(0, 201):
    x.append(abscisse)
    y.append(1 / abscisse)# y=f(x)=1/x
    abscisse += pas
plt.axis([-1, 1, -10, 10])
plt.plot(x, y)
plt.show()
plt.close()
```



La fonction inverse n'est pas définie en 0. Normalement, nous aurions dû obtenir une erreur lorsque nous demandions 1 / abscisse et qu'abscisse valait 0. Cependant, les erreurs d'approximation de python font que l'on ne passe pas par 0 mais par un point proche.
Néanmoins, il ne reste pas moins vrai que la fonction inverse est fortement divergente au voisinage de 0. Elle tend vers $-\infty$ lorsque x tend vers 0 par valeur négative mais tend au contraire vers $+\infty$ lorsque x tend vers 0 en étant positif.
La droite que nous observons est donc une conséquence de cela. On relie un point qui a une ordonnée très négative (le premier point avant 0) à un autre point qui a une ordonnée très positive (le premier point après 0).
Pour éviter cela, on est obligé de dessiner la fonction en deux fois. Une fois avant 0 et une fois après. Finalement, notre code est le suivant :

Editor :

```
import  matplotlib.pyplot as plt
a = 1 / 1000 # Pour éviter 0 ]a,b]
b = 1
x = []
y = []
x1 = []
y1 = []
pas = (b - a) / 200
abscisse = a
for k in range(0, 201): # On fait une seule boucle
    x.append(abscisse) # abscisse représente les abscisses à droite de 0
    y.append(1 / abscisse) #1/x
    x1.append(-abscisse) # -abscisse représente les abscisses à gauche de 0
    y1.append(-1 / abscisse) #-1/x1
    abscisse += pas
plt.axis([-1, 1, -10, 10])
plt.plot(x, y, x1, y1)
plt.show()
plt.close()
```



Exercice :
Ecrire le même programme précédent en fonction, ainsi qu'un programme principal qui fait appel a cette application.

Editor :

```
import matplotlib.pyplot as plt
def fplot(f, a, b, n):
    x = []
    y = []
    abscisse = a
    pas  = (b - a)/ n
    for k in range(0, n + 1):
        x.append(abscisse)
        y.append(f(abscisse))
        abscisse += pas
    plt.plot(x, y)
#Appel de la fonction fplot, pour avoir la courbe de la fonction cosinus sur [0, 2π] :
from math import cos, pi
fplot(cos, 0, 2 * pi, 100)
plt.show()
plt.close()
```

Exercice :
Traçons la courbe de la fonction inverse sur l'intervalle [1, 10] des deux manières, avec la fonction fplot.

1) En définissant la fonction **inverse**, on obtient ce code.

Editor :

```
import matplotlib.pyplot as plt
def fplot(f, a, b, n):
    x = []
    y = []
    abscisse = a
    pas  = (b - a)/ n
    for k in range(0, n + 1):
        x.append(abscisse)
        y.append(f(abscisse))
        abscisse += pas
    plt.plot(x, y)
#Appel de la fonction fplot, pour avoir la courbe de la fonction inverse sur [1, 10] :
def inverse(x):
    return 1 / x
fplot(inverse, 1, 10, 100)
plt.show()
plt.close()
```

2) Et avec la fonction **lambda**, on obtient ce code.

Editor :

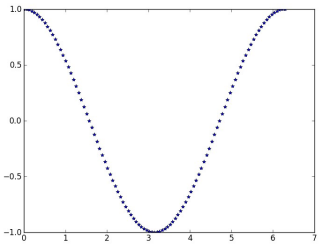
```
import matplotlib.pyplot as plt

def fplot(f, a, b, n):
    x = []
    y = []
    abscisse = a
    pas  = (b - a)/ n
    for k in range(0, n + 1):
        x.append(abscisse)
        y.append(f(abscisse))
        abscisse += pas
    plt.plot(x, y)

#Appel de la fonction fplot, pour avoir la courbe de la fonction lambda sur [1, 10] :

fplot(lambda x : 1 / x, 1, 10, 200)
plt.show()
plt.close()
```

Personnaliser fplot

Nous pouvons personnaliser en ajoutant des paramètres (par exemple style pour le style de lignes et lw pour l'épaisseur du trait). Ces paramètres doivent être facultatifs pour ne pas avoir à les préciser à chaque fois (nous mettrons en paramètre par défaut les paramètres par défaut de plot). On obtient donc la fonction suivante.

Editor :

```
import matplotlib.pyplot as plt

def fplot(f, a, b, n, style = '-', lw = '1'):#les valeurs de style et lw sont par défaut
    x = []
    y = []
    abscisse = a
    pas  = (b - a)/ n
    for k in range(0, n + 1):
        x.append(abscisse)
        y.append(f(abscisse))
        abscisse += pas
    plt.plot(x, y, style , lw = lw)
#Appel de la fonction fplot, pour avoir la courbe de la fonction lambda sur [1, 10] :
from math import cos, pi
fplot(cos, 0, 2 * pi, 100 , style = '*')
plt.show()
plt.close()
```



La fonction plot avec la bibliothèque « numpy » et « panda »

La fonction plot accepte la bibliothèque « numpy » ainsi que la bibliothèque « panda », ce qui permet de faire ce que nous venons de faire plus simplement.

Pour la fonction cosinus, on peut alors écrire ce code.

Editor :

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 2 * np.pi, 0.01) # On crée un array qui va de 0 à 2pi exclu avec un pas de 0.01
plt.plot(x, np.cos(x)) # On utilise plot avec l'array x et l'array cos(x)
plt.show()
```

Et pour la fonction inverse celui-là.

Editor :

```
import matplotlib.pyplot as plt
import numpy as np

def inverse(x): # Retourne l'array contenant les 1 / x
    return np.array([1/i for i in x])

x = np.arange(0.01, 1, 0.01)
plt.plot(x, inverse(x), -x, inverse(-x))
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import matplotlib.pyplot as plt
import numpy as np

x=np.array([1, 3, 4, 6])
y = np.array([2, 3, 5, 1])
plt.plot(x,y)
plt.show()
plt.close()
```

Style « pyplot »

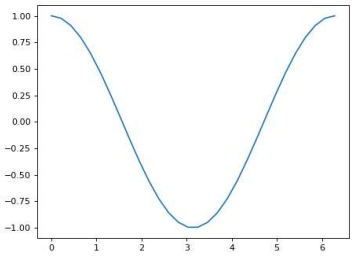
Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

y = np.cos(x)
plt.plot(x, y)
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
plt.close()
```



Définition du domaine des axes - xlim(), ylim() et axis()

Dans le style pyplot, il est possible fixer les domaines des abscisses et des ordonnées en utilisant les fonctions **xlim()**, **ylim()** ou **axis()**.

Syntaxe :

```
xlim(xmin, xmax)
ylim(ymin, ymax)
```

Exemple avec xlim()

Style « pyplot »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
plt.plot(x, y)
plt.xlim(-1, 5)
plt.show()
plt.close()
```

Style « Orienté Objet »

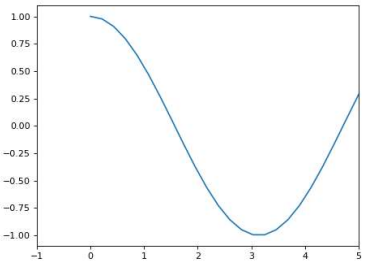
Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(-1,5)
plt.show()
plt.close()
```

Remarque

Dans le Style « Orienté Objet » on à utiliser set_xlim() au lieux xlim() pour le Style « pyplot ».

Le graphe associe



Exemple avec ylim()

Style « pyplot »

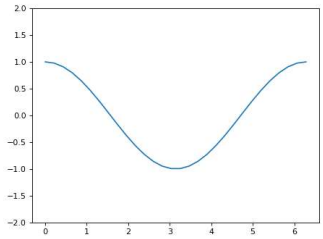
Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
plt.plot(x, y)
plt.ylim(-2, 2)
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_ylim(-2,2)
plt.show()
plt.close()
```



Exemple avec axis()

Style « pyplot »

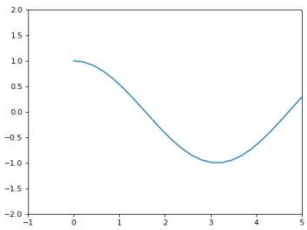
Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
plt.plot(x, y)
plt.axis([-1, 5,-2, 2])
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)
fig, ax = plt.subplots()
ax.plot(x, y)
plt.axis([-1, 5,-2, 2])
plt.show()
plt.close()
```



Affichage de plusieurs courbes

Pour afficher plusieurs courbes sur un même graphe, on peut procéder de la façon suivante :

Style « pyplot »

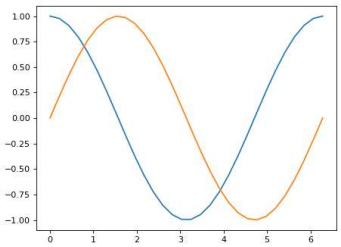
Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2= np.sin(x)
plt.plot(x, y1)
plt.plot(x, y2)
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2= np.sin(x)
fig, ax = plt.subplots()
ax.plot(x, y1)
ax.plot(x, y2)
plt.show()
plt.close()
```



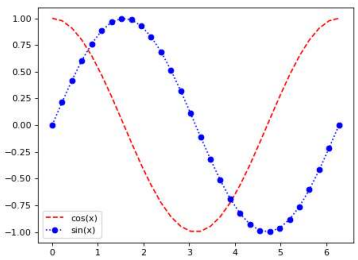
Formats de courbes

Il est possible de préciser la couleur, le style de ligne et de symbole (« marker ») en ajoutant une chaîne de caractères de la façon suivante :

Style « pyplot »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2= np.sin(x)
plt.plot(x, y1, "r--", label="cos(x)")
plt.plot(x, y2, "b:o", label="sin(x)")
plt.legend()
plt.show()
plt.close()
```



Style de ligne
Les chaînes de caractères suivantes permettent de définir le style de ligne :

Chaîne	Effet
-	ligne continue
--	tirets
:	ligne en pointillé
-.	tirets points

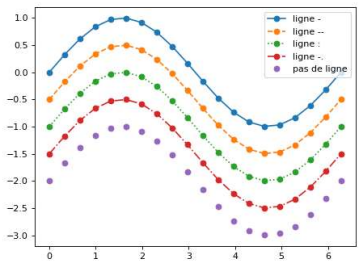
Remarque :
Si on ne veut pas faire apparaître de ligne, il suffit d'indiquer un symbole sans préciser un style de ligne.

Exemple
Style « pyplot »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y = np.sin(x)
plt.plot(x, y, "o-", label="ligne -")
plt.plot(x, y-0.5, "o--", label="ligne --")
plt.plot(x, y-1, "o:", label="ligne :")
plt.plot(x, y-1.5, "o-.", label="ligne -.")
plt.plot(x, y-2, "o", label="pas de ligne")
plt.legend()
plt.show()
plt.close()
```

Résultat



Symbole (« marker »)
Les chaînes de caractères suivantes permettent de définir le symbole « marker »:

Chaîne	Effet
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

Couleur
Les chaînes de caractères suivantes permettent de définir la couleur :

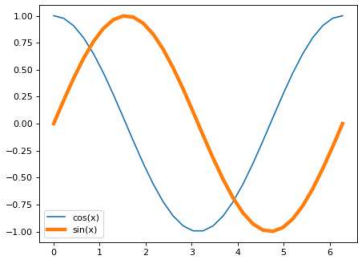
Chaîne	Couleur en anglais	Couleur en français
b	blue	bleu
g	green	vert
r	red	rouge
c	cyan	cyan
m	magenta	magenta
y	yellow	jaune
k	black	noir
w	white	blanc

Largeur de ligne
Pour modifier la largeur des lignes, il est possible de changer la valeur de l'argument « linewidth ».

Style « pyplot »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2= np.sin(x)
plt.plot(x, y1, label="cos(x)")
plt.plot(x, y2, label="sin(x)" , linewidth=4)
plt.legend()
plt.show()
plt.close()
```



Tracé de formes

Comme la fonction **plot()** ne fait que relier des points, il est possible de lui fournir plusieurs points avec la même abscisse.

Style « pyplot »

Editor :

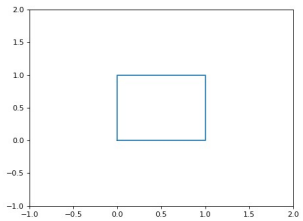
```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
plt.plot(x, y)
plt.xlim(-1, 2)
plt.ylim(-1, 2)
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlim(-1, 2)
ax.set_ylim(-1, 2)
plt.show()
plt.close()
```

Résultat



L'instruction axis ("equal")

L'instruction axis ("equal") permet d'avoir la même échelle sur l'axe des abscisses et l'axe des ordonnées afin de préserver la forme lors de l'affichage. En particulier, grâce à cette commande un carré apparaît vraiment comme un carré, de même pour un cercle.

Style « pyplot »

Editor :

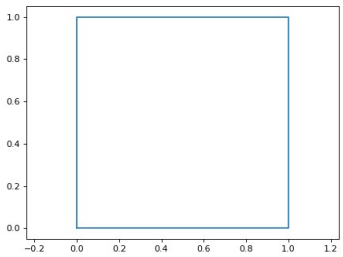
```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
plt.plot(x, y)
plt. axis("equal")

plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
fig, ax = plt.subplots()
ax.plot(x, y)
ax.axis("equal")
plt.show()
plt.close()
```

Résultat



Il est aussi possible de fixer le domaine des abscisses en utilisant **axis()**.

Style « pyplot »

Editor :

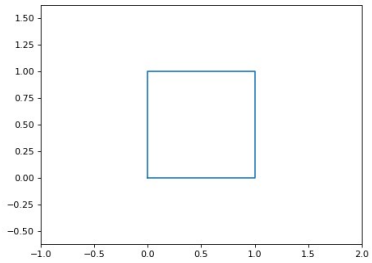
```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
plt.plot(x, y)
plt. axis("equal")
plt. axis ([-1, 2, -1, 2])
plt.show()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])
fig, ax = plt.subplots()
ax.plot(x, y)
ax.axis("equal")
ax.axis ([-1, 2, -1, 2])
plt.show()
```

Résultat



Remarque

On peut noter que les limites ne sont pas respectées en même temps pour les abscisses et les ordonnées, mais la forme est bien préservée.

Tracé d'un cercle

On peut tracer utiliser une courbe paramétrique pour tracer un cercle.
Sans axis ("equal"), la courbe n'apparaît pas comme un cercle.

Style « pyplot »

Editor :

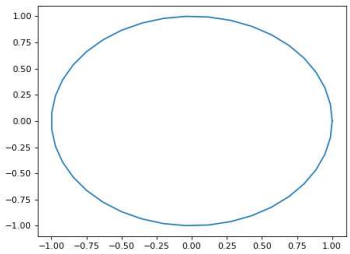
```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
plt.plot(x, y)
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
plt.close()
```

Résultat



Avec axis("equal"), on obtient bien un cercle.

Style « pyplot »

Editor :

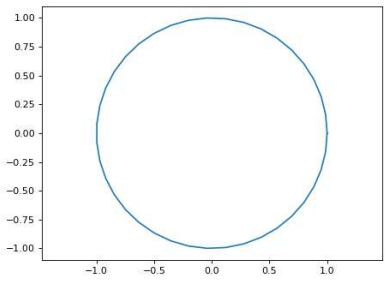
```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
plt.plot(x, y)
plt.axis("equal")
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.axis("equal")
plt.show()
plt.close()
```

Résultat



Il est aussi possible de fixer le domaine des abscisses en utilisant **axis()**.

Style « pyplot »

Editor :

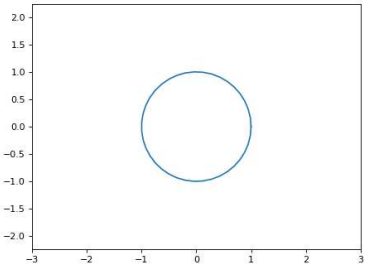
```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
plt.plot(x, y)
plt.axis("equal")
plt.axis([-3, 3, -3, 3])
plt.show()
plt.close()
```

Style « Orienté Objet »

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np

theta = np.linspace(0, 2*np.pi, 40)
x = np.cos(theta)
y = np.sin(theta)
fig, ax = plt.subplots()
ax.plot(x, y)
ax.axis("equal")
ax.axis([-3, 3, -3, 3])
plt.show()
plt.close()
```

Résultat



Remarque

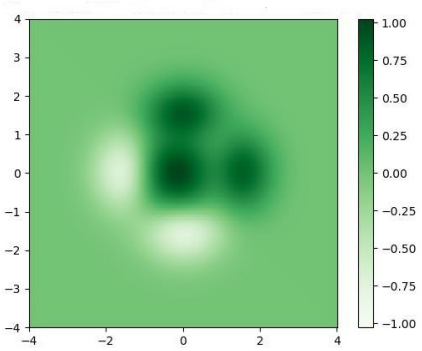
A nouveau, on peut noter que les limites ne sont pas respectées en même temps pour les abscisses et les ordonnées, mais la forme est bien préservée.

Exemple :

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LogNorm

dx, dy = 0.015, 0.05
y, x = np.mgrid[slice(-4, 4 + dy, dy),
                slice(-4, 4 + dx, dx)]
z = (1 - x / 3. + x ** 5 + y ** 5) * np.exp(-x ** 2 - y ** 2)
z = z[:-1, :-1]
z_min, z_max = -np.abs(z).max(), np.abs(z).max()

c = plt.imshow(z, cmap ='Greens', vmin = z_min, vmax = z_max,
               extent =[x.min(), x.max(), y.min(), y.max()],
                  interpolation ='nearest', origin ='lower')

plt.colorbar(c)

plt.title('matplotlib.pyplot.imshow() function Example',  fontweight ="bold")
plt.show()
plt.close()
```

Résultat

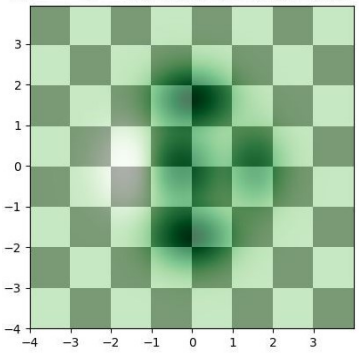


Exemple :

Editor :

```
import  matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LogNorm

dx, dy = 0.015, 0.05
x = np.arange(-4.0, 4.0, dx)
y = np.arange(-4.0, 4.0, dy)
X, Y = np.meshgrid(x, y)

extent = np.min(x), np.max(x), np.min(y), np.max(y)

Z1 = np.add.outer(range(8), range(8)) % 2
plt.imshow(Z1, cmap ="binary_r", interpolation ='nearest',
                        extent = extent, alpha = 1)

def geeks(x, y):
   return (1 - x / 2 + x**5 + y**6) * np.exp(-(x**2 + y**2))

Z2 = geeks(X, Y)

plt.imshow(Z2, cmap ="Greens", alpha = 0.7,
         interpolation ='bilinear', extent = extent)

plt.title('matplotlib.pyplot.imshow() function Example',
                              fontweight ="bold")
plt.show()
plt.close()
```

Résultat



Figures

Pour afficher les figures ici dans le notebook, il faut également utiliser l'instruction %matplotlib nbagg quand on importe les différents modules (sinon les figures s'affichent dans une fenêtre à part). (Quand vous utiliserez IDLE vous n'aurez pas besoin de l'instruction %matplotlib nbagg.)

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import numpy as np
import  matplotlib.pyplot as plt

plt.figure(1)
plt.plot(range(5),[4,3.6,2.5,3.2,4.1])

plt.figure(2)
x = np.arange(0., 10., 0.1)
y1 = 2*x
y2 = np.sqrt(x)+6*np.log(x+1)
y3 = x**2-10*x
plt.plot(x,y1, x,y2, x,y3)
plt.title('Titre')
plt.ylabel("Axe des ordonnees")
plt.xlabel("Axe des abscisses")

plt.figure(3)
x = np.arange(0., 5., 0.1)
y = x**2
plt.plot(x,y)
plt.plot(x,y+1, linewidth=5.0, color='r', linestyle='--')
plt.plot(x,y+2, 'r--')
plt.plot(x,y+3, 'g:')
plt.plot(x,y+4, 'b-.')
plt.plot(x,y+5, 'c^')

courbes = plt.plot(x,y+6, x,y+7, x,y+8)
plt.setp(courbes, color='m', linestyle='-.')

courbes2 = plt.plot(x,30-y, x,y-3)
plt.setp(courbes2,color='k', marker = '.',markevery=5, markersize=50,markeredgecolor='g',
        markerfacecolor='w')
```

Plusieurs figures dans une fenêtre

Voyons maintenant comment créer plusieurs figures dans une seule fenêtre. Voici un exemple, dont le code est expliqué après :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import numpy as np
import  matplotlib.pyplot as plt

#Suite des figures de la page précédente
def f(t):
   return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(4)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')

plt.close(1)
plt.close(2)
plt.close(3)
plt.close(4)
```

Projection 3D Matplotlib

Matplotlib Matplotlib 3D

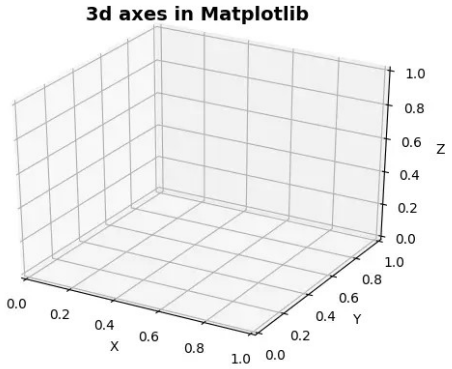
1. Tracer des axes 3D dans Matplotlib
2. Diagramme de dispersion 3D dans Matplotlib
3. Diagramme de dispersion 3D dans Matplotlib

Ce tutoriel explique comment créer des tracés 3D dans Matplotlib en utilisant le paquet mplot3d de la bibliothèque mpl_toolkits.

Tracer des axes 3D dans Matplotlib

Editor :

```
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

axes = plt.axes(projection="3d")
axes.set_title("3d axes in Matplotlib", fontsize=14, fontweight="bold")
axes.set_xlabel("X")
axes.set_ylabel("Y")
axes.set_zlabel("Z")
plt.show()
```



Il crée un tracé en 3D avec les axes X, Y et Z. Pour créer un tracé 3d Matplotlib, nous importons le paquet mplot3d de la bibliothèque mpl_toolkits. Le paquet mpl_toolkits est installé pendant que nous installons Matplotlib en utilisant pip.

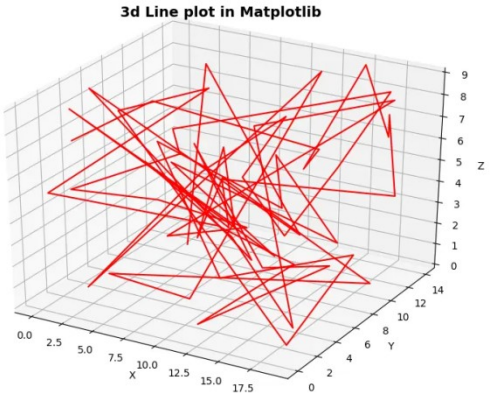
Le tracé des axes 3D sur une figure Matplotlib est similaire au tracé des axes 2D. Nous venons de définir projection="3d" dans matplotlib.pyplot.axes() pour tracer un axe 3D dans Matplotlib.

Nous devons nous assurer que la version de Matplotlib est 1.0 ou supérieure.

Diagramme de dispersion 3D dans Matplotlib

Editor :

```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

x = np.random.randint(20, size=60)
y = np.random.randint(15, size=60)
z = np.random.randint(10, size=60)

fig = plt.figure(figsize=(8, 6))
axes = plt.axes(projection="3d")
axes.plot3D(x, y, z, color="red")
axes.set_title("3d Line plot in Matplotlib", fontsize=14, fontweight="bold")
axes.set_xlabel("X")
axes.set_ylabel("Y")
axes.set_zlabel("Z")
plt.tight_layout()
plt.show()
```

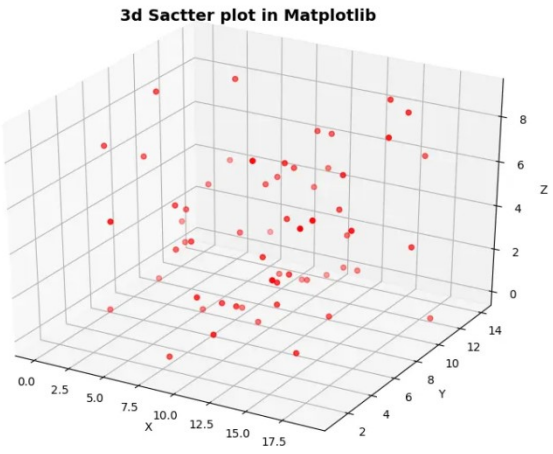


Il crée un tracé de ligne en 3D dans Matplotlib. Pour créer un tracé de ligne en 3D dans Matplotlib, nous créons d'abord les axes, puis nous utilisons la méthode plot3D() pour créer le tracé de ligne en 3D. Nous passons les coordonnées X, Y et Z des points à tracer en argument à la méthode plot3D().

Diagramme de dispersion 3D dans Matplotlib

Editor :

```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

x = np.random.randint(20, size=60)
y = np.random.randint(15, size=60)
z = np.random.randint(10, size=60)

fig = plt.figure(figsize=(8, 6))
axes = plt.axes(projection="3d")
axes.scatter3D(x, y, z, color="red")
axes.set_title("3d Sactter plot in Matplotlib", fontsize=14, fontweight="bold")
axes.set_xlabel("X")
axes.set_ylabel("Y")
axes.set_zlabel("Z")
plt.tight_layout()
plt.show()
```



Il crée un nuage de points en 3D dans Matplotlib. Pour créer un nuage de points en 3D dans Matplotlib, nous créons d'abord les axes, puis nous utilisons la méthode scatter3D() pour créer le nuage de points en 3D. Nous passons les coordonnées X, Y et Z des points à tracer en argument à la méthode scatter3D().
Notez que nous ajoutons la 3D à la fin du nom des fonctions de traçage 2D pour générer les tracés 3D correspondants. Par exemple, la fonction plot() réalise le tracé de lignes 2D tandis que plot3D() génère le tracé de lignes 3D.

La bibliothèque Pandas

Pandas est une bibliothèque open source qui fournit des structures de données et des outils d'analyse de données performants et faciles à utiliser pour Python.

Pandas offre plusieurs avantages pour la visualisation de données, notamment :

- Une manipulation facile des ensembles de données volumineux
- Une intégration avec d'autres bibliothèques Python
- Une large gamme de types de graphiques
- Une personnalisation pour des visualisations complexes

Pandas prend en charge une grande variété de types de graphiques, notamment :

- Graphiques linéaires
- Graphiques à barres
- Histogrammes
- Diagrammes en boîte
- Graphiques de dispersion
- Graphiques hexbin
- Graphiques de zone
- Graphiques circulaires

Installation de Panda

Voir le site officiel https://www.anaconda.com/download
Sou forme windows
Installation avec invite de commande(voir le lien suiant :)
https://www.journaldunet.fr/developpeur/developpement/1441209-comment-installer-pandas-en-utilisant-pip-sur-la-cmd-de-windows/

Pour utiliser le gestionnaire de paquets "pip", il faut saisir la commande "py" et l'argument "-m", qui permet d'appeler un module de Python.
1- py -m pip install pandas

Si vous souhaitez utiliser les commandes des autres systèmes, vous pouvez les ajouter à la variable PATH du système
2-setx PATH "%PATH%;C:\<chemin\vers\le\repertoire\de\python>\Scripts"

3-pip install pandas
4-py -m pip install --trusted-host pypi.python.org pip pandas

PIP va alors installer Pandas dans le répertoire %APPDATA% de votre compte
5-py -m pip install --user pandas

La dernière possibilité consiste à créer un environnement virtuel dans lequel on installera Pandas. Il faut faire appel au module "venv" pour gérer les environnements virtuels. En indiquant le chemin de l'environnement, on le créera.
6-py -m venv c:\chemin\vers\le\nouvel\environnement

Comment importer des bibliothèques et des ensembles de données dans Pandas ?
Voici un exemple d'importation de Pandas et de lecture d'un fichier CSV :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
```

Qu'est-ce qu'un graphique linéaire dans Pandas ?
Un graphique linéaire est un type de graphique qui affiche des informations sous la forme d'une série de points de données reliés par des segments de droite. Il est utile pour visualiser les tendances et les relations entre les variables au fil du temps.

Exemple : Création d'un simple graphique linéaire dans Pandas :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
data.plot(kind='line', x='date', y='price')
```

Comment créer un graphique à barres dans Pandas ?

Exemple : Création d'un simple graphique à barres dans Pandas qui affiche des données:

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
data.plot(kind='bar', x='category', y='value')
```

Remarque
La longueur des barres représente les valeurs des données.

Exemple 1 : Graphique linéaire des prix mensuels des actions
Dans cet exemple, nous allons tracer les prix mensuels des actions de trois géants de la technologie : Facebook, Microsoft et Apple.

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

stock_data = pd.read_csv('stock_data.csv')

# rééchantillonner les données pour obtenir le prix moyen des actions #pour chaque mois
monthly_data = stock_data.resample('M', on='Date').mean()

# On trace les prix mensuels des actions pour chaque entreprise
monthly_data.plot(kind='line', x='Date', y=['Facebook', 'Microsoft', 'Apple'])

plt.title("Prix mensuels des actions de Facebook, Microsoft et Apple")
plt.xlabel('Date')
plt.ylabel('Prix de l'action')
plt.legend(['Facebook', 'Microsoft', 'Apple'])
plt.show()
plt.close()
```

Ce graphique montre les tendances mensuelles des prix des actions de Facebook, Microsoft et Apple, nous permettant de comparer leurs performances au fil du temps.

Exemple 2 : Graphique à barres des ventes de produits
Dans cet exemple, nous allons créer un graphique à barres pour visualiser les ventes de différents produits dans un magasin. Tout d'abord, importons les bibliothèques nécessaires et chargeons les données de ventes :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

sales_data = pd.read_csv('sales_data.csv')

# nous regrouperons les données de vente par produit
monthly_data = stock_data.resample('M', on='Date').mean()
product_sales = sales_data.groupby('Product')['Sales'].sum()

# Création  d'un diagramme à barres pour visualiser les ventes de chaque produit
product_sales.plot(kind='bar')
plt.title('Ventes des produits')
plt.xlabel('Produit')
plt.ylabel('Ventes')
plt.show()
plt.close()
```

Ce diagramme à barres affiche les ventes de chaque produit, ce qui facilite l'identification des produits les plus vendus dans le magasin.

Exemple 3 : Personnalisation des graphiques avec Pandas
Pandas vous permet de personnaliser vos graphiques de différentes manières, telles que le changement des couleurs, l'ajout d'étiquettes et l'ajustement de la taille du graphique. Voici un exemple de personnalisation d'un graphique linéaire avec Pandas :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
data.plot(kind='line', x='date', y='price', figsize=(10, 6), color='red', linestyle='dashed', linewidth=2)
plt.title('Graphique linéaire personnalisé')
plt.xlabel('Date')
plt.ylabel('Prix')
plt.show()
plt.close()
```

Dans cet exemple, nous avons personnalisé le graphique linéaire en changeant la couleur en rouge, en utilisant un style de ligne en pointillés et en définissant l'épaisseur de la ligne à 2. Nous avons également ajusté la taille du graphique à l'aide du paramètre "figsize".

Exemple 4 : Tracer un DataFrame Pandas avec plusieurs axes
Parfois, vous souhaiterez afficher plusieurs graphiques dans la même figure. Pandas facilite la création de sous-graphiques à l'aide du paramètre "subplots". Voici un exemple de création d'une grille 2x2 de sous-graphiques :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')

fig, axes = plt.subplots(2, 2, figsize=(10, 6))

data.plot(kind='line', x='date', y='price', ax=axes[0, 0])
data.plot(kind='bar', x='category', y='value', ax=axes[0, 1])
data.plot(kind='scatter', x='date', y='price', ax=axes[1, 0])
data.plot(kind='hist', y='price', ax=axes[1, 1])

plt.tight_layout()
plt.show()
plt.close()
```

Dans cet exemple, nous avons créé une grille 2x2 de sous-graphiques, chacun contenant un type de graphique différent.

Exemple 5 : Ajouter des barres d'erreur à un graphique Pandas
Les barres d'erreur sont utiles pour afficher la variabilité ou l'incertitude des points de données sur un graphique. Voici un exemple d'ajout de barres d'erreur à un graphique à barres avec Pandas :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
 data.plot(kind='bar', x='category', y='value', yerr='error')

plt.title('Graphique à barres avec barres d'erreur')
plt.xlabel('Catégorie')
plt.ylabel('Valeur')
plt.show()
plt.close()
```

Exemple 6 : Personnalisation de la légende dans un graphique Pandas
Vous pouvez personnaliser la légende dans un graphique Pandas en ajustant sa position, sa taille et d'autres propriétés. Voici un exemple de personnalisation de la légende dans un graphique linéaire :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')

ax = data.plot(kind='line', x='date', y='price')

ax.legend(loc='upper right', fontsize=12, title='Price', title_fontsize=14, frameon=False)

plt.title('Graphique linéaire avec légende personnalisée')
plt.xlabel('Date')
plt.ylabel('Prix')
plt.show()
plt.close()
```

Dans cet exemple, nous avons personnalisé la légende en définissant sa position dans le coin supérieur droit, en modifiant la taille de la police à 12, en ajoutant un titre, en définissant la taille de la police du titre à 14 et en supprimant le cadre autour de la légende.

Exemple 7 : Gestion des données catégorielles dans un graphique Pandas
Pandas facilite la gestion des données catégorielles lors de la création de graphiques. Voici un exemple de création d'un graphique à barres à l'aide de données catégorielles :

Editor :

```
%matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('categorical_data.csv')

data['category'] = data['category'].astype('category')

data.plot(kind='bar', x='category', y='value')

plt.title('Graphique à barres avec données catégorielles')
plt.xlabel('Catégorie')
plt.ylabel('Valeur')
plt.show()
plt.close()
```

Dans cet exemple, nous avons converti la colonne "category" en un type de données catégorielles à l'aide de la méthode "astype", ce qui permet à Pandas de gérer correctement les données catégorielles lors de la création du graphique à barres.

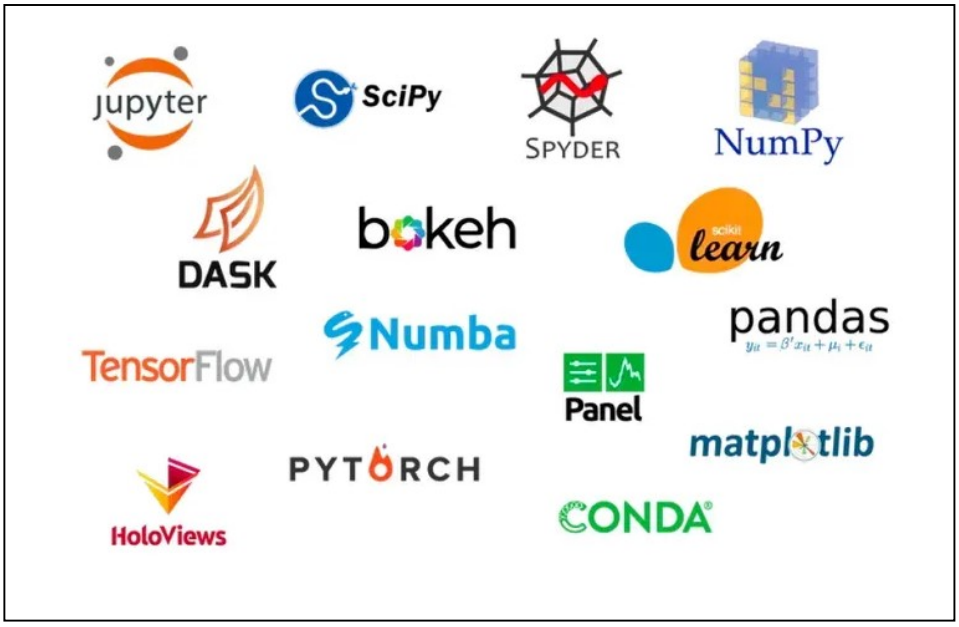
Exemple 8 : Tracé de données avec des échelles différentes sur plusieurs axes
Parfois, vous souhaiterez tracer des données avec des échelles différentes sur la même figure. Vous pouvez le faire avec Pandas en utilisant plusieurs axes. Voici un exemple de création d'un graphique linéaire avec deux axes y :

Editor :

```
#matplotlib nbagg
import matplotlib
import pandas as pd
import  matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
 fig, ax1 = plt.subplots()
 ax1.plot(data['date'], data['price'], color='blue', label='Price')
ax1.set_xlabel('Date')
ax1.set_ylabel('Price', color='blue')
ax1.tick_params(axis='y', labelcolor='blue')

ax2 = ax1.twinx()

ax2.plot(data['date'], data['volume'], color='red', label='Volume')
ax2.set_ylabel('Volume', color='red')
ax2.tick_params(axis='y', labelcolor='red')

fig.legend(loc='upper right')
plt.title('Graphique linéaire avec deux axes y')
plt.show()
plt.close()
```

Dans cet exemple, nous avons créé un graphique linéaire avec deux axes y, un pour le prix et un pour le volume. Les données de prix sont tracées en bleu sur l'axe y de gauche, tandis que les données de volume sont tracées en rouge sur l'axe y de droite.

Les bibliothèques pour python



Fichiers

On sait que la plus part de l'information utilisé par l'utilisateur est stockée sous forme de texte dans des fichiers. Pour traiter cette information, vous devez le plus souvent lire ou écrire dans un ou plusieurs fichiers. Python possède pour cela de nombreux outils qui vous simplifient la vie.
Les informations sont écrites dans un fichier « text.txt ».

Exemple
Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom « animaux_sauvage.txt » et le contenu suivant :

animaux_sauvages.txt

```
Lion
Loup
Girafe
Tigre
Singe
Renard
Eléphant
Guépard
Hyène
Alligator
.
.
.
```

Editer un fichier
Pour **ouvrir un fichier** en **python** on utilise la fonction open .

Cette fonction prend en premier paramètre le chemin du fichier (relatif ou absolu) et en second paramètre le type d'ouverture

Chemin relatif / chemin absolu
Un **chemin relatif** en informatique est un chemin qui prend en compte l'emplacement de lecture.

Un **chemin absolu** est un chemin complet qui peut être lu quelque soit l'emplacement de lecture.

La syntaxe est la suivante

Syntaxe

fichier = **open**(nom du fichier, paramètre d'ouverture)

Paramètres d'ouvertures

Il existe plusieurs modes d'ouverture:

modes	Désignations
'r'	Pour une ouverture en lecture (READ). **Exemple :** fic = **open**('animaux_sauvage.txt' ,'r')
'w'	Pour une ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier est écrasé. Si le fichier n'existe pas python le crée. **Exemple :** fic = **open**('animaux_sauvage.txt' ,'w')
' a'	Pour une ouverture en mode ajout à la fin du fichier (APPEND). Si le fichier n'existe pas python le crée. **Exemple :** fic = **open**('animaux_sauvage.txt' ,'a')
' b '	Pour une ouverture en mode binaire. **Exemple :** fic = **open**('animaux_sauvage.txt' ,'b')
't '	Pour une ouverture en mode texte. **Exemple :** fic = **open**('animaux_sauvage.txt' ,'t')
'x '	Crée un nouveau fichier et l'ouvre pour écriture **Exemple :** fic = **open**('animaux_sauvage.txt' ,'x')

L'instruction **open**('animaux_sauvage.txt ','a') suppose que le fichier 'animaux_sauvage.txt' est dans le répertoire depuis lequel l'interpréteur Python a été lancé. Si ce n'est pas le cas, il faut préciser le **chemin d'accès** au fichier. Par exemple pour Windows:

C:\Users\mustapha\ animaux_sauvage.txt.

Remarque : « fic » est une variable de type fichier

Fermeture d'un fichier

Comme tout élément ouvert, il faut le refermer une fois les instructions terminées. Pour cela on utilise la méthode close() .

La syntaxe est la suivante :

Syntaxe

```
fichier.close()
```

Lire le contenu d'un fichier

Pour afficher tout le contenu d'un fichier, vous pouvez utiliser la méthode read sur l'objet-fichier.

Editor :

```
# coding: utf-8
fic = open('animaux_sauvage.txt' ,'r')
print fic.read()
fic.close()
```

Il existe plusieurs possibilités pour récupérer le contenu d'un fichier (après ouverture de ce fichier, relié à une variable fic) :

modes	Désignations
fic.read()	va lire l'intégralité du fichier et la mettre dans une chaîne de caractères ;
fic.readlines()	va lire toutes les lignes du fichier et les mettre dans un tableau (une liste Python), qu'on pourra ensuite parcourir avec une boucle for ;
fic.readline()	va lire une ligne à la fois, de façon séquentielle ;
for ligne in fic :	peut aussi être utilisée en Python.

Editor :

```
# coding: utf-8

fic = open('animaux_sauvage.txt' ,'r')
lignes = fic.readlines()
lignes
fic.close()
```

Résultat

IDLE Shell

```
['Lion\n          ','Loup\','Girafe\n','Tigre\n','Singe\n','Renard\n','Eléphant\n','Guépard\n','Hyène\n',
'Alligator\n','Crocodile\n']
```

Itérations directe sur le fichier

Editor :

```
# coding: utf-8

fic = open('animaux_sauvage.txt' ,'r')
lignes = fic.readlines()
```

s:

IDLE Shell

```
Lion
Loup
Girafe
Tigre
Singe
Renard
Eléphant
Guépard
Hyène
Alligator
Crocodile
```

Le mot clé with

Il existe une autre syntaxe plus courte qui permet de s'émanciper du problème de fermeture du fichier: le mot clé with .

Voici la syntaxe:

Syntaxe

```
with open('nom_fichier.txt', 'r') as fichier:
        fichier.read()
        print (fichier)
```

Editor :

```
# coding: utf-8

with open('animaux_sauvage.txt' ,'r') as fic
lignes = fic.read()
lignes
fic.close()
```

Résultat

IDLE Shell

```
['Lion\n          ','Loup\','Girafe\n','Tigre\n','Singe\n','Renard\n','Eléphant\n','Guépard\n','Hyène\n',
'Alligator\n','Crocodile\n']
```

Editor :

```
# coding: utf-8

with open('animaux_sauvage.txt' ,'r') as fic
   ligne = fic.readlines()
   while ligne != "":
   print(ligne)
   ligne = fic.readlines()
fic.close()
```

Résultat

IDLE Shell

```
Lion
Loup
Girafe
Tigre
Singe
Renard
Eléphant
Guépard
Hyène
Alligator
Crocodile
```

Itérations directe sur le fichier
Voici un autre moyen à la fois simple et élégant de parcourir un fichier en utilisant with.

Editor :

```
# coding: utf-8

with open('animaux_sauvage.txt' ,'r') as fic
    for ligne in fic:
    print(ligne)
fic.close()
```

Résultat

IDLE Shell

```
Lion
Loup
Girafe
Tigre
Singe
Renard
Eléphant
Guépard
Hyène
Alligator
Crocodile
```

L'objet filin est « itérable », ainsi la boucle for va demander à Python d'aller lire le fichier ligne par ligne.

Ecrire dans un fichier
Voici la syntaxe pour écrire dans un fichier:

Editor :

```
# coding: utf-8

fic = open('animaux_sauvage.txt' ,'a')
fic.write('Crocodile')
fic.close()
```

A noter que pour le monde d'ouverture a , si vous voulez écrire à la ligne, vous pouvez utiliser le saut de ligne \n :

Editor :

```
# coding: utf-8

fic = open('animaux_sauvage.txt' ,'a')
fic.write('\nCrocodile')
fic.close()
```

Exemple
Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

Editor :

```
# coding: utf-8
animaux = ['poisson', 'abeille', 'chat']
with open("animaux_1.txt'","w") as fic:
    for animal in animaux:
        fic.write(animal)
fic.close()
```

Utiliser l'écriture formatée
Nous voulions le nom de chaque animal sur une ligne. On doit ajouter le caractère fin de ligne après chaque nom d'animal.
Pour ce faire, nous pouvons utiliser l'écriture formatée :

Editor :

```
# coding: utf-8
animaux = ['poisson', 'abeille', 'chat']
with open("animaux_1.txt'","w") as fic:
    for animal in animaux:
#  L'écriture formatée  permet d'ajouter un retour à la ligne (\n) après le nom de chaque animal.
        fic.write(f'{animal}\n')
fic.close()
```

Windows utilise deux caractères spéciaux pour le retour à la ligne :
\r correspondant à un retour chariot (hérité des machines à écrire) et \n.

la fonction open() « dans Python 3 » gère tout ça automatiquement et renvoie uniquement des sauts de ligne sous forme d'un seul \n (même si le fichier a été conçu sous Windows et qu'il contient initialement des \r).

Importance des conversions de types avec les fichiers
Vous avez sans doute remarqué que les méthodes qui lisent un fichier (par exemple .readlines()) vous renvoient systématiquement des chaînes de caractères. De même, pour écrire dans un fichier il faut fournir une chaîne de caractères à la méthode .write().

Pour tenir compte de ces contraintes, il faudra utiliser les fonctions de conversions de types vues au chapitre 2 Variables : int(), float() et str(). Ces fonctions de conversion sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier.

En effet, les nombres dans un fichier sont considérés comme du texte, donc comme des chaînes de caractères, par la méthode .readlines(). Par conséquent, il faut les convertir (en entier ou en float) si on veut effectuer des opérations numériques avec.
Fonctions sur les file handles

Fonctions	**Désignation**
fh.read(1000)	lit 1000 caractères.
content = fh.read()	lit tout le fichier (ou le reste) (chaîne vide si fichier fini).

Attention : fh.read() est bloquant. Si ce qu'on lit est en cours d'écriture, il va attendre que l'EOF final soit disponible ! Si on veut faire un read non bloquant, **il faut transformer le file handle en non bloquant :**

import fcntl; fcntl.fcntl(fh, fcntl.F_SETFL, fcntl.fcntl(fh, fcntl.F_GETFL) | os.O_NONBLOCK)

Fonctions	Désignation
fh.readline()	lit la ligne suivante (chaîne vide si fichier fini).
fh.readlines()	renvoie une liste de toutes les lignes.
fh.write('myString')	écrit une string
fh.tell()	renvoie la position courante dans le fichier.
fh.close()	ferme le fichier
fh.seek(offset, from_what)	repositionne le curseur : si from_what = 0 : offset à partir du début si from_what = 1 : offset à partir de la position courante si from_what = 2 : offset à partir de la fin si from_what omis : il vaut 0. (offset peut être négatif)

Boucle sur les lignes d'un fichier

Editor :

```
# coding: utf-8
file = open('myFile')
for line in file:
  print(line)
file.close()
```

Attention : ça fait du buffering, contrairement à un simple readline() ! Donc utiliser readline() si on veut l'éviter.
On peut récupérer directement la liste des lignes d'un fichier en faisant : [x.replace(' ', '') for x in open('myFile')
Lecture d'un fichier avec fermeture automatique de celui-ci à la fin, comme s'il y avait un finally (même en cas d'exception) :

Editor :

```
# coding: utf-8
with open('myfile.txt') as file:
    for line in file:
        print(line)
file.close()
```

Diverses fonctions

Fonctions	Désignation
os.environ :	renvoie un dictionnaire de tous les variables d'environnement avec leur valeur
os.getcwd()	renvoie le directory courant.

os.chdir(myDir)	change le directory courant.
os.getuid(), os.geteuid(), os.getgid(), os.getegid()	renvoient les uid ou gid, ou les uid ou gid effectifs
os.getlogin()	renvoie le login.
os.getpid()	renvoie le process id (PID).

os.getenv('MY_VAR'), os.putenv('MY_VAR', '1'), os.unsetenv('MY_VAR') lit, fixe ou détruit une variable d'environnement (valable pour les sous-process lancés)
os.umask(0o002) : positionne le umask à 002. Attention, un nombre en octal doit être précédé de 0o !

Fonctions	**Désignation**
os.chmod('myfile', 0o755	fait un changement des droits. Attention, un nombre en octal doit être précédé de 0o !
os.uname()	renvoie un tuple avec les infos de uname.
os.listdir('/myDir')	renvoie la liste des fichiers (entrées) du directory.
os.mkdir(myDir)	crée un directory. os.mkdir(myDir, 0o777) en indiquant le mode, mais en appliquant par dessus le umask qui vient restreindre les droits. Lève une exception si le directory existe déjà.
os.makedirs(myDir)	crée le directory en créant tous les intermédiaires si nécessaire.
os.remove(myFile)	détruit un fichier (pas un directory).
os.unlink(myFile)	comme os.remove().
os.rmdir(myDir)	détruit un directory seulement s'il est vide.
os.removedirs(myDir)	détruit un directory et si son parent est vide, le détruit aussi et remonte comme ça toute l'arborescence jusqu'à un directory non vide. Pour détruire un directory et tout ce qu'il contient, utiliser shutil.rmtree
os.rename(oldFile, newFile)	renomme un fichier ou directory. Attention : écrase le fichier destination s'il existe !

Attention : ne marche pas entre filesystems (utiliser shutil.move() pour cela).

Fonctions	**Désignation**
os.symlink(myFile, myLink)	crée un lien symbolique.
os.symlink('myFile', '/myDir/myFile2')	Pour faire un lien relatif de myFile vers myFile2 dans le directory /myDir
os.link(myFile, myLink)	crée un hard lien (spécifique unix/linux).
os.walk(myDir)	renvoie un generator qui renvoie à chaque fois un tuple avec (dirpath, dirnames, filenames) où dirpath est le directory courant (chemin complet à partir de myDir inclus), dirnames la liste de tous les sous-directories, et filenames la liste des fichiers (sans les directories).
for dirpath, dirnames, filenames in os.walk('docs'): print(dirpath, dirnames, filenames)	
os.stat(myFile)	permet d'avoir les infos sur l'inode comme la date de dernière modification, la taille du fichier ou le user : **import stat**

	print(os.stat(myFile)[stat.ST_MTIME])
stat.ST_UID	le user id
stat.ST_GID	le group id
stat.ST_SIZE	la taille
stat.ST_ATIME	le timestamp de la dernière date d'accès.
stat.ST_MTIME	le timestamp de la dernière date de modification

Accès en lecture ou écriture
Pour tester si un directory peut être écrit : os.access(myDir, os.W_OK | os.X_OK)
Pour tester si un fichier peut être lu : os.access(myFile, os.R_OK)

Fonctions	Désignation
os.kill(myPid, 9)	kill le process (utiliser import signal; os.kill(myPid, signal.SIGKILL) pour être plus propre.

Appels systèmes

Fonctions	Désignation
os.system(myCommand)	avec retour du statut, mais on ne peut pas récupérer le stdout du programme appelé. Cette méthode est déconseillée. L'appel est bloquant.

Exceptions

Syntaxe des exceptions :

```
while 1:
   try:
      x = int(raw_input('number:'))
      break
   except ValueError:
      print('try again')
```

Une clause except peut avoir plusieurs exceptions :
Séquentiellement (mais seul le premier qui matche sera exécuté, même si plusieurs matchent)
.

Syntaxe des exceptions :

```
try:
   f = open('toto')
   s = f.readline()
   i = int(string.strip(s))
except IOError:
   print('I/O error')
except ValueError:
   print('could not convert')
except: # all other exceptions
   print('unexpected')
   raise # reraise exception after message printing
else: # done if no exception raised
   f.close()
```

except sans type d'exception catche tous les types.
raise tout seul dans un except : lève à nouveau la même exception.
ou récupération de différents types d'exception en même temps :

```
try:
   ...
except (RuntimeError, TypeError, NameError):
   ...
```

Les exceptions sont des classes :
Elles dérivent toutes de la classe BaseException, mais si on veut créer ses propres classes d'exception, les faire dériver de la classe Exception, plutôt que BaseException (voir ci-dessous).
Lors de l'appel à **sys.exit()**, une exception SystemExit est levée. SystemExit dérive directement de BaseException, donc si on catche BaseException, on va aussi la catcher !
Lors d'un Ctrl-C, une exception KeyboardInterrupt est levée. KeyboardInterrupt dérive aussi directement de BaseException, donc si on catche BaseException, on va aussi la catcher !
On peut définir ses propres classes d'exception en les faisant dériver de la classe Exception (éviter de les faire dériver de BaseException).
On peut catcher toutes les exceptions "normales" et examiner leur type ou le message associé .

Editeur :

```
try:
   x = 'a' + 8
#except Exception as e:
   print(type(e))
   print(str(e))
```

StandardError est une exception dérivée d'Exception et de laquelle la plupart des exceptions dérivent (notamment RuntimeError, KeyError, ...)
On peut exécuter du code si aucune exception levée :

Syntaxe des exceptions :

```
try:
   ...
except TypeError:
   ...
else:
   print('Aucune exception n'a eu lieu')
```

Certaines exceptions ont des arguments que l'on peut alors récupérer :

Editeur :

```
except NameError as e: # un argument, e, contenant les arguments de l'exception.
   print(arg:', e)
```

Les exceptions non traitées remontent d'appel en appel.

Pour catcher toutes les exceptions en récupérant leur message :

Syntaxe des exceptions :

```
try:
   ...
except Exception as e:
  print(str(e))
  ...
```

Levée d'exception explicite

Fonctions	Désignation
raise NameError('badName') ou aussi raise NameError, 'badName'	lève une exception NameError avec un argument.

On peut lever une exception avec plusieurs arguments de types quelconques (récupérables sous forme de tuple) :

raise Exception(arg1, arg2)

On peut alors récupérer ces arguments en faisant :

Editeur :

```
except Exception as e:
  print(e.args)
```

Syntaxe des exceptions :

```
try:
   1 / 0
except TypeError:
   print('type error')
finally: # toujours exécuté, avant de lever l'exception
   print('always done')
```

Syntaxe des exceptions :

```
# Traceback : permet de voir la pile des appels lors d'une erreur :

import traceback

# imprime sur stderr la pile des appels de la dernière exception.
traceback.print_exc()

# permet de récupérer la pile des appels sous forme de chaîne de caractères.
traceback.format_exc()
```

Pour avoir la pile des appels

Editeur :

```
import inspect

stack = inspect.stack()
# renvoie une liste avec un élément par élément de la pile avec le premier élément étant l'appel
#à cette fonction stack()
```

pour avoir le nom de la fonction appelante la fonction contenant cet ordre : stack[1][3]
pour avoir la ligne ou la fonction est appelée : stack[1][2]
Pour avoir la liste des numéros de lignes après une exception :

Editeur :

```
info = sys.exc_info()[2]
while True:
  print(str(info.tb_lineno)+ ' at ' + info.tb_frame.f_code.co_filename)
  info = info.tb_next
  if info is None:
    break
```

Assertions en python

Fonctions	Désignation
assert x == 2	vérifie si la condition est vraie, et si non, lance une exception AssertionError.
assert x == 2	'x ne valait pas 2' : idem, mais en plus, affiche le message d'erreur indiqué.

Les assertions sont vérifiées, sauf si on fournit à python l'option -O est fournie (python -O ...)

Nmap et les fichiers

"mmap" est une bibliothèque Python qui permet de mapper des fichiers ou des segments de mémoire en Python. Elle fournit une interface qui ressemble à celle de tableaux Python, ce qui permet d'accéder aux données du fichier ou du segment de mémoire comme s'il s'agissait d'un tableau Python.

L'utilisation de "mmap" peut être une approche plus efficace pour accéder aux données d'un fichier ou d'un segment de mémoire que l'utilisation de fonctions de lecture et d'écriture standard, car elle permet de mapper le fichier ou le segment de mémoire en mémoire et d'accéder aux données de manière directe, sans passer par l'overhead de l'interpréteur Python. Cela peut être particulièrement utile pour traiter de grandes quantités de données ou pour accéder à des données de manière répétée.

Voici comment utiliser "mmap" pour lire un fichier en Python :

Editeur :

```
import mmap

# Ouvre le fichier en mode lecture et récupère sa taille
with open('mon_fichier.txt', 'r') as f:
    size = f.seek(0, 2)
    f.seek(0)

    # Crée un objet mmap à partir du fichier
    mm = mmap.mmap(f.fileno(), size)

    # Lit le contenu du fichier à l'aide de l'objet mmap
    contents = mm.read()

# Affiche le contenu du fichier
print(contents)
```

Classes

Les classes sont un moyen de réunir des données et des fonctionnalités. Créer une nouvelle classe crée un nouveau type d'objet et ainsi de nouvelles instances de ce type peuvent être construites. Chaque instance peut avoir ses propres attributs, ce qui définit son état. Une instance peut aussi avoir des méthodes (définies par la classe de l'instance) pour modifier son état.

La notion de classes en Python s'inscrit dans le langage avec un minimum de syntaxe et de sémantique nouvelles. C'est un mélange des mécanismes rencontrés dans C++ et Modula-3. Les classes fournissent toutes les fonctionnalités standards de la programmation orientée objet : l'héritage de classes autorise les héritages multiples, une classe dérivée peut surcharger les méthodes de sa ou ses classes mères et une méthode peut appeler la méthode d'une classe mère qui possède le même nom. Les objets peuvent contenir n'importe quel nombre ou type de données. De la même manière que les modules, les classes participent à la nature dynamique de Python : elles sont créées pendant l'exécution et peuvent être modifiées après leur création.

Dans la terminologie C++, les membres des classes (y compris les données) sont publics (sauf exception, voir Variables privées ou protégés) et toutes les fonctions membres sont virtuelles. Comme avec Modula-3, il n'y a aucune façon d'accéder aux membres d'un objet à partir de ses méthodes : une méthode est déclarée avec un premier argument explicite représentant l'objet et cet argument est transmis de manière implicite lors de l'appel. Comme avec Smalltalk, les classes elles-mêmes sont des objets. Il existe ainsi une sémantique pour les importer et les renommer. Au contraire de C++ et Modula-3, les types natifs peuvent être utilisés comme classes mères pour être étendus par l'utilisateur. Enfin, comme en C++, la plupart des opérateurs natifs avec une syntaxe spéciale (opérateurs arithmétiques, indiçage, etc.) peuvent être redéfinis pour les instances de classes.

En l'absence d'une terminologie communément admise pour parler des classes, nous utilisons parfois des termes de Smalltalk et C++. Nous voulions utiliser les termes de Modula-3 puisque sa sémantique orientée objet est plus proche de celle de Python que C++, mais il est probable que seul un petit nombre de lecteurs les connaissent.

Objets et noms : préambule

Les objets possèdent une existence propre et plusieurs noms peuvent être utilisés (dans divers contextes) pour faire référence à un même objet. Ce concept est connu sous le nom d'alias dans d'autres langages. Il n'apparaît pas au premier coup d'œil en Python et il peut être ignoré tant qu'on travaille avec des types de base immuables (nombres, chaînes, n-uplets). Cependant, les alias peuvent produire des effets surprenants sur la sémantique d'un code Python mettant en jeu des objets muables comme les listes, les dictionnaires et la plupart des autres types. En général, leur utilisation est bénéfique au programme car les alias se comportent, d'un certain point de vue, comme des pointeurs. Par exemple, transmettre un objet n'a aucun coût car c'est simplement un pointeur qui est transmis par l'implémentation ; et si une fonction modifie un objet passé en argument, le code à l'origine de l'appel voit le changement. Ceci élimine le besoin d'avoir deux mécanismes de transmission d'arguments comme en Pascal.

Portées et espaces de nommage en Python

Avant de présenter les classes, nous devons aborder la notion de portée en Python. Les définitions de classes font d'habiles manipulations avec les espaces de nommage, vous devez donc savoir comment les portées et les espaces de nommage fonctionnent. Soit dit en passant, la connaissance de ce sujet est aussi utile aux développeurs Python expérimentés.

Commençons par quelques définitions.

Un espace de nommage est une table de correspondance entre des noms et des objets. La plupart des espaces de nommage sont actuellement implémentés sous forme de dictionnaires Python, mais ceci n'est normalement pas visible (sauf pour les performances) et peut changer dans le futur. Comme exemples d'espaces de nommage, nous pouvons citer les primitives (fonctions comme abs() et les noms des exceptions de base) ; les noms globaux dans un module ; et les noms locaux lors d'un appel de fonction. D'une certaine manière, l'ensemble des attributs d'un objet forme lui-même un espace de nommage. L'important à retenir concernant les espaces de nommage est qu'il n'y a absolument aucun lien entre les noms de différents espaces de nommage ; par exemple, deux modules différents peuvent définir une fonction maximize sans qu'il n'y ait de confusion. Les utilisateurs des modules doivent préfixer le nom de la fonction avec celui du module.

À ce propos, nous utilisons le mot attribut pour tout nom suivant un point. Par exemple, dans l'expression **z.real, real est un attribut de l'objet z**. Rigoureusement parlant, les références à des noms dans des modules sont des références d'attributs : dans l'expression **nommodule.nomfonction, nommodule est un objet module et nomfonction est un attribut de cet objet.** Dans ces conditions, il existe une correspondance directe entre les attributs du module et les noms globaux définis dans le module : ils partagent le même espace de nommage 1 !

Les attributs peuvent être en lecture seule ou en écriture. Dans ce dernier cas, l'affectation aux attributs est possible. Les attributs du module sont accessibles en écriture : vous pouvez écrire
nomMod :la_reponse = 42.
Les attributs inscriptibles peuvent également être supprimés avec l'instruction del.

Exemple :

IDLE Shell

```
>>del nomMod
#la_reponse supprimera l'attribut la_reponse de l'objet nommé par nomMode.
```

Les espaces de nommage sont créés à différents moments et ont différentes durées de vie. L'espace de nommage contenant les primitives est créé au démarrage de l'interpréteur Python et n'est jamais effacé. L'espace de nommage globaux pour un module est créé lorsque la définition du module est lue. Habituellement, les espaces de nommage des modules durent aussi jusqu'à l'arrêt de l'interpréteur. Les instructions exécutées par la première invocation de l'interpréteur, qu'elles soient lues depuis un fichier de script ou de manière interactive, sont considérées comme faisant partie d'un module appelé __main__, de façon qu'elles possèdent leur propre espace de nommage (les primitives vivent elles-mêmes dans un module, appelé bulletins).
L'espace des noms locaux d'une fonction est créé lors de son appel, puis effacé lorsqu'elle renvoie un résultat ou lève une exception non prise en charge (en fait, « oublié » serait une

meilleure façon de décrire ce qui se passe réellement). Bien sûr, des invocations récursives ont chacune leur propre espace de nommage.
La portée est la zone textuelle d'un programme Python où un espace de nommage est directement accessible. « Directement accessible » signifie ici qu'une référence non qualifiée à un nom est cherchée dans l'espace de nommage.
Bien que les portées soient déterminées de manière statique, elles sont utilisées de manière dynamique. À n'importe quel moment de l'exécution, il y a au minimum trois ou quatre portées imbriquées dont les espaces de nommage sont directement accessibles :
La portée la plus au centre, celle qui est consultée en premier, contient les noms locaux ;
Les portées des fonctions imbriquées, qui sont consultées en commençant avec la portée imbriquée la plus proche, contiennent des noms non-locaux mais aussi non-globaux ;
L'avant-dernière portée contient les noms globaux du module courant :
La portée imbriquée, consultée en dernier, est l'espace de nommage contenant les primitives.

Exemples Vaiables globales et locals pour les fonctions dans une classe

Editeur :

```
class Test:
   """Un exemple simple de classe"""
   i = 12345

   def f(self):
      return 'Bonjour'
```

Résultat

IDLE Shell

```
>> x = Test()
>>> p rint(x.i)
>>> print(x.f())
```

Résultat

```
12345
Bonjour
```

Exemple 2

Editeur :

```
class point:
   def __init__(self,dx,dy,dz):
      self.x=dx
      self.y=dy
      self.z=dz

d=point(3,4,5)
```

Résultat

IDLE Shell

```
>> d.x
```

Résultat

```
3
```

Exemple 3

Editeur :

```
class point:
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz
    def ajout(self,dxx,dyy,dzz):
        self.x = self. x + dxx
        self.y = self.y + dyy
        self.z = self.z + dzz
```

Résultat

IDLE Shell

```
>> p=point(3,2,1)
>>> p.ajout(1,1,1)
>>> p.x
```

Résultat

```
4
>>> p.y
```

Résultat

```
3
```

Résultat

```
>>>p.z
2
```

Exemple 4

Editeur :

```
class point:
    x = 0
    y = 0
    z = 0
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz
    def ajout(self,dxx,dyy,dzz):
        self.x = self. x + dxx
        self.y = self.y + dyy
        self.z = self.z + dzz
```

Exemple 5

Editeur :

```
class menu:
   x=[]
   y= []
   z =[]
   def __init__(self, num1,num2,num3):
      self.x.append(num1)
      self.y.append(num2)
      self.z.append(num3)
```

Résultat

IDLE Shell

```
>>>a=menu(3,2,4)
>>>a.x
```
Résultat
```
[3]
>>>a.z
```
Résultat
```
[4]
>>>a.y
```
Résultat
```
[2]
```

Un autre appel

IDLE Shell

```
>>>b=menu(7,8,9)
>>>b.x
```
Résultat
```
[3, 7]
>>>b.y
```
Résultat
```
[2, 8]
>>>b.z
```
Résultat
```
[4, 9]
```

Exemple 4

Editeur :

```
class point:
    x = 0
    y = 0
    z = 0
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz
    def ajout(self,dxx,dyy,dzz):
        self.x = self. x + dxx
        self.y = self.y + dyy
        self.z = self.z + dzz
```

Exemple 6

Editeur :

```
class point:
#Déclaration des listes#
    x1 = []
    y1 = []
    z1  = []
#Déclaration des listes#
    x = 0
    y = 0
    z  = 0
#Initialisé les variables#
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz
#Ajouter dse élémens à plusieurs variables#
    def ajout(self,dxx,dyy,dzz):
        self.x = self. x + dxx
        self.y = self.y + dyy
        self.z = self.z + dzz
#Ajouter des elements à plusieurs listes#
    def  add_x_y_z(self, dxx,dyy,dzz):
        self.x1.append(dxx)
        self.y1.append(dyy)
        self.z1.append(dzz)
```

Résultat

IDLE Shell

```
>>>a=point(3,2,5)
>>>a.x
Résultat
3
>>>b=point(2,3,2)
>>>b.x1
Résultat
[]
>>>b.add_x_y_z(3,4,5)
>>>b.x1
Résultat
[3]
```

IDLE Shell

```
>>>a=point(7,-8,-3)
>>>a.x1
Résultat
[3]
>>>a.add_x_y_z (-8,-7,-1)
>>>a.x1
Résultat
[3, -8]
```

Surcharge des opérateurs en python

La surcharge d'opérateurs vous permet de redéfinir la signification d'opérateur en fonction de votre classe. C'est la magie de la surcharge d'opérateurs que nous avons pu utiliser l'opérateur + pour ajouter deux objets numériques, ainsi que pour concaténer deux objets chaîne.

Cette fonctionnalité en Python, qui permet à un même opérateur d'avoir une signification différente en fonction du contexte, est appelée surcharge d'opérateur.

Alors que se passe-t-il lorsque nous les utilisons avec des objets d'une classe définie par l'utilisateur ? Considérons la classe suivante :

Exemple 7

Editeur :

```
class point:
#Déclaration des listes#
   x = 0
   y = 0
   z  = 0
#Initialisé les variables#
   def __init__(self,dx,dy,dz):
      self.x=dx
      self.y=dy
      self.z=dz
```

Résultat

IDLE Shell

```
>>>p1 = point(2, 4,-1)
>>>p2 = point(5, 1,3)

>>>p3 = p1+p2
```

Résultat

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    p3=p1+p2
TypeError: unsupported operand type(s) for +: 'point' and 'point'
```

TypeError a été généré car Python ne savait pas comment ajouter deux objets Point ensemble. Cependant, la bonne nouvelle est que nous pouvons apprendre cela à Python en surchargeant les opérateurs. Mais tout d'abord, parlons des fonctions spéciales.
La surcharge de l'opérateur est obtenue en définissant une méthode spéciale dans la définition de classe. Les noms de ces méthodes commencent et finissent par un double soulignement (__).

Opérateurs arithmétiques
Opérateur +
Pour surcharger l'opérateur +, nous devrons implémenter la fonction __add __ () dans la classe. Un grand pouvoir implique de grandes responsabilités. Nous pouvons faire ce que nous voulons, dans cette fonction.

Exemple 8 :

Editeur :

```
class point:
    x = 0
    y = 0
    z = 0
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz

    def __str__(self):
        return "({0},{1},{2})".format(self.x, self.y, self.z)#mettre un espace après la virgule dans
                                                              #le format
    def __add__(self, p):
        a = self. x + p.x
        b = self.y + p.y
        c = self.z + p.z
        return point(a,b,c)
```

Résultat

IDLE Shell

```
>>>p1 = point(2, 4,-1)
>>>p2 = point(5, 1,3)

>>>p3 = p1+p2
>>> print(p3)
```

Résultat

(7,5,2)

Si l'expression est de la forme x + y, Python l'interprète comme x .__ add __ (y). La version de la méthode __add __ () appelée dépend du type de x et de y.

Fonctions spéciales de surcharge de l'opérateur en Python

Le tableau suivant répertorie les opérateurs et leur méthode spéciale correspondante.

Operateur	Expression	Interprétation Python
Addition	p1 + p2	p1.__add__(p2)
Soustraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Puissance	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Division entière	p1 // p2	p1.__floordiv__(p2)
le reste (modulo)	p1 % p2	p1.__mod__(p2)
Décalage binaire gauche	p1 << p2	p1.__lshift__(p2)

Décalage binaire droite	p1 >> p2	p1.__rshift__(p2)
ET binaire	p1 & p2	p1.__and__(p2)
OU binaire	p1 | p2	p1.__or__(p2)
XOR	p1 ^ p2	p1.__xor__(p2)
NON binaire	~p1	p1.__invert__()

Surcharge des opérateurs de comparaison

Python ne limite pas la surcharge des opérateurs aux seuls opérateurs arithmétiques. Nous pouvons également surcharger les opérateurs de comparaison.

Supposons que nous voulions implémenter le symbole inférieur à < dans notre classe Point. Comparons la magnitude de ces points depuis l'origine et retournons le résultat. Il peut être implémenté comme suit.

Exemple 9 :

Editeur :

```
import math

class point:
    x = 0
    y = 0
    z = 0
    def __init__(self,dx,dy,dz):
        self.x=dx
        self.y=dy
        self.z=dz

    def __lt__(self, p):
        m_self = math.sqrt((self.x ** 2) + (self.y ** 2) + (self.z ** 2))
        m2_p = math.sqrt((p.x ** 2) + (p.y ** 2) + (p.z ** 2))
        return m_self < m2_p
```

Résultat

IDLE Shell

```
>>>p1 = point(2, 4,-1)
>>>p2 = point(5, 1,3)

if p1 < p2:
    print("p2 est plus grande que p1")
```

Résultat

```
p2 est plus grande que p1
```

De même, les fonctions spéciales que nous devons implémenter pour surcharger d'autres opérateurs de comparaison sont résumées ci-dessous.

Operateur	**Expression**	**Interprétation Python**
Inférieur à	p1 < p2	p1.__lt__(p2)
Inférieur ou égal	p1 <= p2	p1.__le__(p2)
Egal	p1 == p2	p1.__eq__(p2)
Différent	p1 != p2	p1.__ne__(p2)
Supérieur à	p1 > p2	p1.__gt__(p2)
Supérieur ou égal	p1 >= p2	p1.__ge__(p2)

Exemple 10 :

Editeur :

```
import math
class point:
    def __init__(self,x=0, y=0, z=0):
        self.__x =x
        self.__y =y
        self.__z =z
    def __str__(self):
        return "({0},{1},{2})".format(self.x, self.y, self.z)#mettre un espace après la virgule dans
                                                              #le format
     def Coord_X(self)
        return self.__x

     def Coord_Y(self)
        return self.__y

     def Coord_Z(self)
        return self.__z

    # Opérateur +
    def __add__(self, p):
        return Point3D(self.__x + p.__x, self.__y + p.__y, self.__z + p.__z)

    # Opérateur -
    def __sub__(self, p):
        return Point3D(self.__x - p.__x, self.__y - p.__y, self.__z - p.__z)

    # Opérateur *
    def __mul__(self, p):
        return Point3D(self.__x * p.__x, self.__y * p.__y, self.__z *p.__z)

    # Opérateur **
    def __pow__(self, p):
        return Point3D(self.__x **p.__x, self.__y ** p.__y, self.__z **p.__z)
    # Opérateur / division
    def __truediv__(self, p):
        if p.__x!=0 and p.__y!=0 and p.__z!=0:
            return Point3D(self.__x /p.__x, self.__y/ p.__y, self.__z/p.__z)
       else:
           return Point3D(0, 0, 0)
```

Suite de la définition de la classe Point3D

Editeur :

```
    # Opérateur // division entière
  def __ floordiv__(self, p):
     if p.__x!=0 and p.__y!=0 and p.__z!=0:
        return Point3D(self.__x //p.__x, self.__y//p.__y, self.__z//p.__z)
    else:
       return Point3D(0, 0, 0)

  # Opérateur %
  def __ mod__(self, p):
     if p.__x!=0 and p.__y!=0 and p.__z!=0:
        return Point3D(self.__x %p.__x, self.__y%p.__y, self.__z%p.__z)

 # Opérateur << décalage binaire à gauche
  def __ lshift__(self, p):
     return Point3D(self.__x<<p.__x, self.__y<<p.__y, self.__z<<p.__z)

  # Opérateur >> décalage binaire à droite
  def __ rshift__(self, p):
     return Point3D(self.__x >>p.__x, self.__y >>p.__y, self.__z >>p.__z)

  # Opérateur & ET binaire
  def __and__(self, p):
     # Surcharge de l'opérateur &
     # Ici, nous effectuons une opération spécifique entre les coordonnées x, y et z
     return Point3D(self.x & p.x, self.y & p.y, self.z & p.z)

 # Opérateur | Ou binaire
  def __or__(self, p):
     #Surcharge de l'opérateur | pour additionner les coordonnées x, y et z.
     return Point3D(self.x|p.x, self.y|p.y, self.z|p.z)

  # Opérateur  XOR
 def __xor__(self, p):
        #Surcharge de l'opérateur xor (^) pour combiner deux points 3D.
     return Point3D(self.x ^ p.x, self.y ^ p.y, self.z ^ p.z)

  # Opérateur ~ NOT binaire
  def __invert__(self):
     # Inversion des coordonnées x, y et z
     return Point3D(-self.x, -self.y, -self.z)
```

Editeur :

```
  # Opérateur <
def __lt__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m2_p = math.sqrt((p.__x** 2) + (p.__y** 2) + (p.__x** 2))
    return m_self < m2_p

# Opérateur <=
def __le__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m_p = math.sqrt((p.__x ** 2) + (p.__y** 2) + (p.__z** 2))
    return m_self <= m_p

# Opérateur >
def __gt__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m_p = math.sqrt((p.__x ** 2) + (p.__y** 2) + (p.__z** 2))
    return m_self > m_p

# Opérateur >=
def __ge__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m_p = math.sqrt((p.__x ** 2) + (p.__y** 2) + (p.__z** 2))
    return m_self >= m_p

# Opérateur ==
def __eq__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m_p = math.sqrt((p.__x ** 2) + (p.__y** 2) + (p.__z** 2))
    return m_self == m_p

# Opérateur !=
def __ne__(self, p):
    m_self = math.sqrt((self.__x** 2) + (self.__y** 2)+ (self.__z** 2))
    m_p = math.sqrt((p.__x ** 2) + (p.__y** 2) + (p.__z** 2))
    return m_self != m_p
```

Héritage simple

Déclaration de l'héritage

Pour stipuler la classe père, rien de bien compliqué. Cela se passe à la définition de la classe enfant :

Syntaxe :

```
Class Classe_Enfant(class_Père):
…
```

Initialiser la classe parent

Une fois la classe mère définie, il va vous falloir l'initialiser. Pour cela, il suffit d'appeler le constructeur de la classe père dans le constructeur de la classe enfant :

Syntaxe:

```
class Classe_Enfant(Classe_Pere):
   def __init__(self):
      Classe_Enfant.__init__(self)# déclaration ancienne version inferieur à python 3.x
      …
class Classe_Enfant(Classe_Pere):
   def __init__(self):
      super().__init__()# déclaration nouvelle version à partir de python 3.x
      …
```

Surcharger un attribut

Surcharger un attribut est très simple, puisqu'il suffit de redéfinir sa valeur au sein de la classe enfant.

Syntaxe :

```
class Classe_Enfant(Classe_Pere):
   def __init__(self):
      Classe_Pere.__init__(self)
      self.attribut_classe_Pere = 'nouvelle valeur'
```

Surcharger une fonction

Tout comme pour les attributs, pour surcharger une fonction, il suffit de la redéfinir.

Rien n'empêche l'élément de la classe enfant de faire appel à l'élément de la classe père.

Exemple

Editeur :

```
class Point3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        return f"Point3D({self.x}, {self.y}, {self.z})"

class Vector3D(Point3D):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)# ou syntaxe standard Point3D.__init__(self, x, y, z)

    def __str__(self):
        return f"Vector3D({self.x}, {self.y}, {self.z})"
```

Résultat

IDLE Shell :

```
>>>vector = Vector3D(4, 5, 6)
>>>print(point)
Résultat
Point3D(1, 2, 3)
>>>print(vector)
Résultat
Vector3D(4, 5, 6)
```

Remarque

Veuillez noter qu'à partir de Python 3.X, vous pouvez simplement utiliser la syntaxe :

- super().__init__(x, y, z)

À la place de l'écriture **syntaxe standard**

- Point3D.__init__(self, x, y, z)

Ici, super() est une référence implicite à la classe mère (le terme super-classe est un synonyme de classe mère). À ce titre, cela vous dispense d'utiliser le self lors de l'appel.

La fonction isinstance()

Elle permet de tester le type d'une instance(type d'une objet), cela permet de savoir si un objet appartient à une certaine classe ou pas.

- instance(objet, classe) est True si l'objet est bien du type passé en second argument
- instance(objet, classe) est False sinon.

IDLE Shell :

```
>>> instance(str, Point3D)# objet ici est un attribut ou méthode
Résultat
true
>>> instance(str, Vector3D)
Résultat
true
>>> issubclass (Point3D, Vector3D)
Résultat
true
```

La fonction issubclass()

Elle permet de tester l'héritage d'une classe.

- issubclass(objet enfant, objet père) est true si objet enfant est descendant de l'objet père
- issubclass(objet enfant, objet père) est falsesi objet enfant n'est pas descendant de l'objet père

Exemple

Editeur :

```
class Utilisateur:
    anciennete = 0

    def __init__(self, nom, age):
        self.user_name = nom
        self.user_age = age

    def getNom(self):
        print("salut, je suis ", self.user_name)

class Client(Utilisateur):
    is_client = True
    anciennete = 10

    def __init__(self, nom, age, mail):
        self.user_name = nom
        self.user_age = age
        self.user_mail = mail

    def getNom(self):
        print("Je suis ", self.user_name,". Mon mail est : ", self.user_mail)
```

Résultat

IDLE Shell :

```
>>>omar = Utilisateur("Omar",27)
>>>moad = Client("Moad" , 29, "benkinouar_Moad@yahoo.fr")
>>>moad.user_name
#Résultat
#'Moad'
>>>moad.is_client
#Résultat
True
>>moad.anciente
#Résultat
10
>>>moad.getNom()
#Résultat
Je suis  Moad . Mon mail est : benkinouar_moad@yahoo.fr
```

Modifions à nouveau notre classe `Client` afin d'étendre la fonction `__init__()` de la classe mère :

Exemple

Editeur :

```
class Utilisateur:
    anciennete = 0

    def __init__(self, nom, age):
        self.user_name = nom
        self.user_age = age

    def getNom(self):
        print("salut, je suis ", self.user_name)

class Client(Utilisateur):
    is_client = True
    anciennete = 10

    def __init__(self, nom, age, mail):
        Utilisateur.__init__(self,nom, age)#changement de la fonction def__init__
        Self.user_mail = mail

    def getNom(self):
        print("Je suis ", self.user_name,". Mon mail est : ", self.user_mail)
```

Résultat

IDLE Shell :

```
>>>moad = Client("Moad" , 29, "benkinouar_Moad@yahoo.fr")
>>>moad.user_name
#Résultat
#'Moad'
>>> moad.user_mail
#Résultat
'benkinouar_moad@yahoo.fr'
>>>moad.anciente
#Résultat
10
```

Le polymorphisme en Python orienté objet

Le mot polymorphie vient du grec et signifie "qui peut prendre plusieurs formes". Dans le contexte de la programmation orientée objet, le polymorphisme est un concept qui fait référence à la capacité d'une variable, d'une fonction ou d'un objet à prendre plusieurs formes, c'est-à-dire à sa capacité de posséder plusieurs définitions différentes. On distingue 2 types de polymorphisme, la surcharge et la redéfinition.

La surcharge

La surcharge est une possibilité offerte par certains langages de programmation qui permet de choisir entre différentes implémentations d'une même fonction ou méthode selon le nombre et le type des arguments fournis.

Le polymorphisme ad hoc ne doit pas être confondu avec le polymorphisme d'inclusion des langages à objets, permis par l'héritage de classe et la redéfinition de méthode (overriding en anglais).

La surcharge peut être :

- statique (le choix de l'implémentation est alors fait à la compilation en fonction du nombre d'arguments et de leur type statique déclaré)
- Ou dynamique (le choix de l'implémentation est alors fait à l'exécution en fonction du type dynamique des arguments).

La surcharge dynamique est également appelée « dispatch ».

La redéfinition

La notion de polymorphisme est très liée à celle d'héritage. Grâce à la redéfinition, il est possible de redéfinir une méthode dans des classes héritant d'une classe de base. Par ce mécanisme, une classe qui hérite des méthodes d'une classe de base peut modifier le comportement de certaines d'entre elles pour les adapter à ses propres besoins.

Contrairement à la surcharge, une méthode redéfinie doit non seulement avoir le même nom que la méthode de base, mais **le type et le nombre de paramètres doivent être identiques à ceux de la méthode de base**.

Exemple
Polymorphisme appliqué à des vecteurs 3D et des points 3D en Python:

Supposons que nous ayons deux classes: Vecteur3D et Point3D. Le polymorphisme nous permettra de manipuler ces objets de manière générique, sans se soucier de leur type spécifique.

Editeur :

```
class Vecteur3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def afficher(self):
        print(f"Vecteur 3D: ({self.x}, {self.y}, {self.z})")

class Point3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def afficher(self):
        print(f"Point 3D: ({self.x}, {self.y}, {self.z})")
```

Résultat

IDLE Shell :

```
# Créons quelques objets
>>>v = Vecteur3D(1, 2, 3)
>>>p = Point3D(4, 5, 6)

# Utilisons le polymorphisme pour afficher les objets
>>>def afficher_objet(obj):
        obj.afficher()

>>>afficher_objet(v)
Résultat
# Affiche: Vecteur 3D:
(1, 2, 3)
>>>afficher_objet(p)
Résultat
 # Affiche: Point 3D:
(4, 5, 6)
```

Dans cet exemple, la méthode afficher est redéfinie dans les classes Vecteur3D et Point3D. Lorsque nous appelons afficher_objet, il fonctionne de manière polymorphe, affichant correctement les informations spécifiques à chaque objet.

Héritage multiple

On parle d'héritage multiple en programmation orientée objet lorsqu'une sous-classe peut hériter de plusieurs classes pères différentes.

Dans la pratique, l'héritage multiple est une chose très difficile à mettre en place au niveau du langage puisqu'il faut prendre en charge les cas où plusieurs classes parents définissent les mêmes variables et fonctions et définir une procédure pour indiquer de quelle définition la sous-classe héritera.

En Python, dans la majorité des cas, l'héritage va se faire selon l'ordre des classes parents indiquées et cela de manière récursive. Imaginons qu'une sous-classe **AGrave** hérite de trois classes **A**, **Accent** et **Abracadabra** dans cet ordre et que la classe A hérite elle même de la classe **Alphabet** tandis que **Abracadabra** hérite de **Mot**

On crée un objet en instanciant notre classe **AGrave** et on appelle une méthode depuis notre objet. Python va à priori commencer par chercher la méthode dans **AGrave**, puis si il ne la trouve pas cherchera dans A. S'il ne la trouve pas il cherchera ensuite dans Alphabet, puis dans **Accent**, puis dans **Abracadabra** et finalement dans **Mot**.

Regardez plutôt le code suivant qui illustre bien cette situation :

Exemple

Editeur :

```
class Alphabet:
    def __init__(self, nom):
        self.lettre_nom= nom
    def info(self):
        print("Je suis une lettre de l'alphabet")
    def test1(self):
        print("Fonction test1() de la classe Alphabet")
    def test2(self):
        print("Fonction test1() de la classe Alphabet")

class Mot:
    def info(self):
        print("Je suis un un mot")
    def test1(self):
        print("Fonction test1() de la classe Mot")
    def test3(self):
        print("Fonction test3() de la classe Mot")

class Accent:
    def info(self):
        print("Je suis une lettre accentuée")
    def test2(self):
        print("Fonction test2() de la classe Accent")
    def test3(self):
        print("Fonction test3() de la classe Accent")

class A(Alphabet):
    def info(self):
        print("Je suis un A")
    def test1(self):
        print("Fonction test1() de la classe A")

class Abracadabra(Mot):
    def test1(self):
        print("Fonction test1() de la classe Abracadabra")

class AGrave(A, Accent, Abracadabra):
    lettrA =True
    lettreAccent =True
    #Etc.  Etc.
```

Résultat

IDLE Shell :

```
>>>aAccentGrave = AGrave("à")
>>>aAccentGrave.lettre_nom
Résultat
'à'
>>>aAccentGrave.info()
Résultat
Je suis un A
>>>aAccentGrave.test1()
Résultat
Fonction test1() de la classe A
>>>aAccentGrave.test2()
Résultat
Fonction test1() de la classe Alphabet
>>>aAccentGrave.test2()
Résultat
Fonction test1() de la classe Alphabet
>>>aAccentGrave.test3()
Résultat
Fonction test3() de la classe Accent
```

Héritage multiple en Python orienté objet

En réalité, Python utilise un algorithme relativement complexe qui détermine l'ordre d'appel (method resolution order, ou MRO en anglais) de manière dynamique, c'est-à-dire en fonction des relations entre les différentes classes.

Variables de classe statiques en Python

- Utilisez la staticmethod() pour définir des variables statiques en Python
- Utilisez la @staticmethod pour définir des variables statiques en Python

Variables de classe statiques en Python

Une variable statique en Python est une variable déclarée dans une classe définie mais pas dans une méthode. Cette variable peut être appelée via la classe à l'intérieur de laquelle elle est définie mais pas directement. Une variable statique est également appelée variable de classe. Ces variables sont confinées à la classe, elles ne peuvent donc pas changer l'état d'un objet.

Utilisez la staticmethod() pour définir des variables statiques en Python

La staticmethod() en Python est une fonction intégrée utilisée pour renvoyer une variable statique pour une fonction donnée.

Cette méthode est maintenant considérée comme une ancienne façon de définir une variable statique en Python.

Exemple

Editeur :

```
class StaticVar:
   def random(text):

      print(text)
      print("This class will print random text.")
StaticVar.random = staticmethod(StaticVar.random)
StaticVar.random("This is a random class.")
```

Ici, tout d'abord, nous créons une classe appelée StaticVar. Dans le programme, nous déclarons une variable appelée random comme variable statique en dehors de la classe en utilisant la fonction staticmethod(). Par cela, nous pouvons appeler le random() directement en utilisant la classe StaticVar.

Utilisez la @staticmethod pour définir des variables statiques en Python

@staticmethod est un moyen moderne et le plus utilisé pour définir une variable statique. La @staticmethod est un décorateur intégré à Python. Un décorateur est un modèle conçu en Python utilisé pour créer une nouvelle fonctionnalité sur un objet déjà existant sans modifier sa structure initiale. Ainsi, le décorateur @staticmethod est utilisé pour définir une variable statique à l'intérieur d'une classe en Python.

Exemple

Editeur :

```
class StaticVar:
   @staticmethod
   def random(text):
      # show custom message
      print(text)
      print("This class will print random text.")


StaticVar.random("This is a random class.")
```

Notez que le décorateur @staticmethod est défini avant de définir la variable statique random. De ce fait, nous pouvons facilement appeler la variable random à la fin via la classe StaticVar.

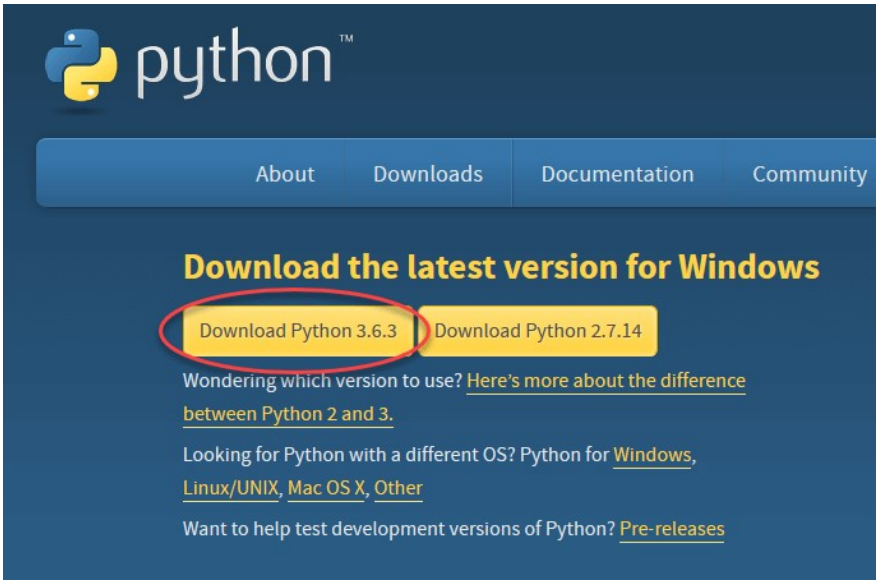
Notez également que, dans les deux méthodes, nous n'utilisons pas l'argument self, qui permet d'accéder aux attributs et méthodes de la fonction lors de la définition de la variable random. C'est parce que les variables statiques n'opèrent jamais à travers des objets.

Comment installer l'IDE Python

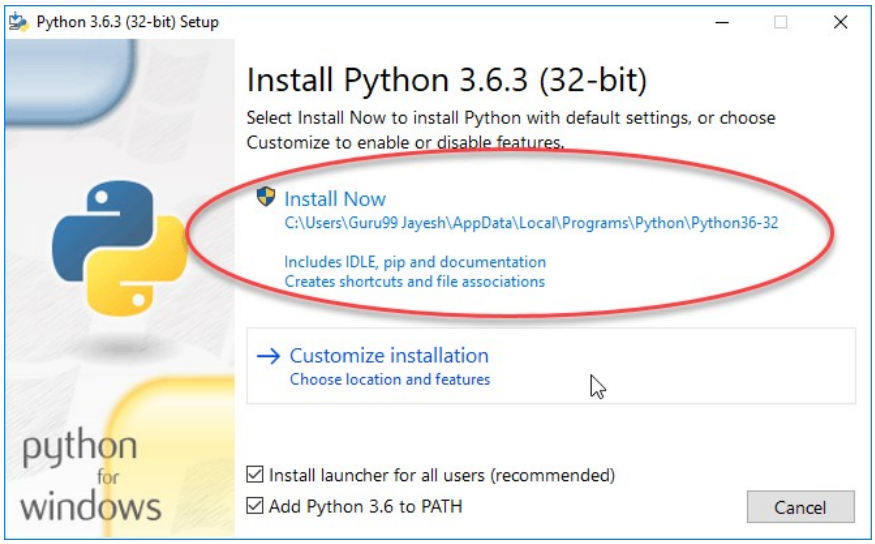
Vous trouverez ci-dessous un processus étape par étape pour télécharger et installer Python sur Windows :

Comment installer Python sur Windows [Pycharm IDE] (guru99.com)

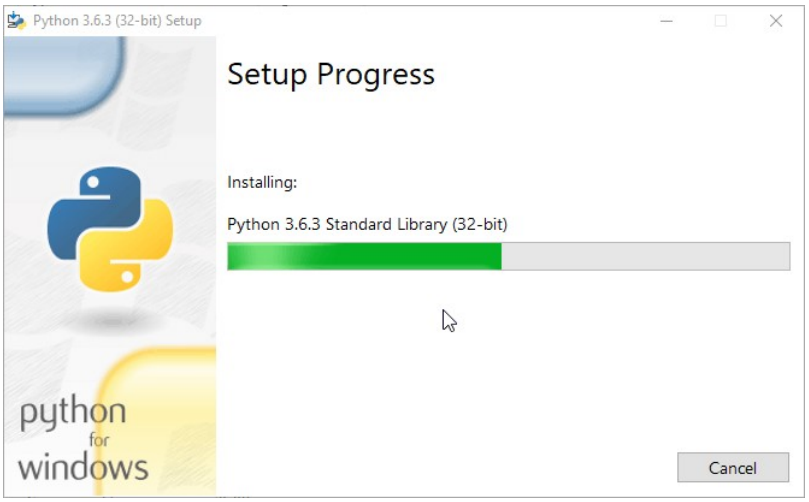
Étape 1) Pour télécharger et installer Python, visitez le site officiel de Python https://www.python.org/downloads/ et choisissez votre version. Nous avons choisi Version Python 3.6.3



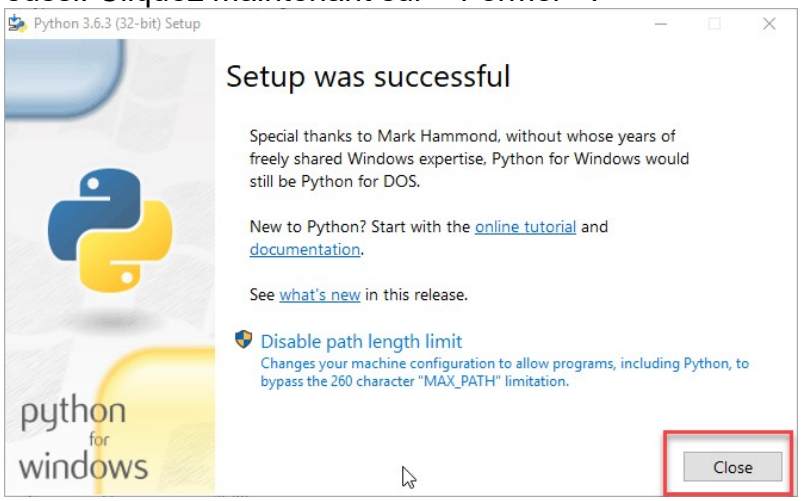
Étape 2) Une fois le téléchargement terminé, exécutez le fichier .exe pour installer Python. Cliquez maintenant sur Installer maintenant.



Étape 3) Vous pouvez voir Python s'installer à ce stade.



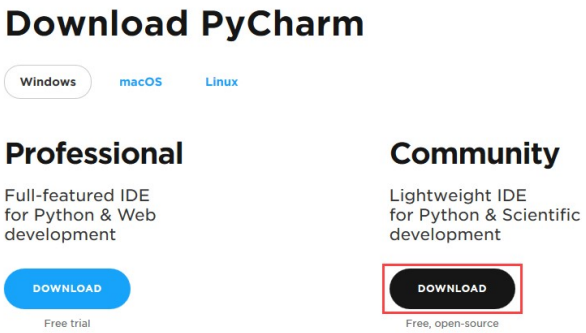
Étape 4) Une fois l'opération terminée, vous pouvez voir un écran indiquant que l'installation a réussi. Cliquez maintenant sur « Fermer ».



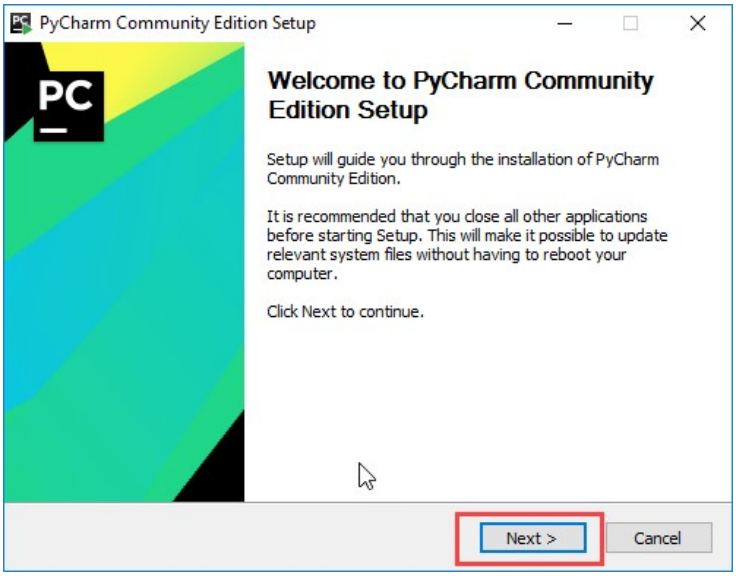
Comment installer Pycharm

Voici un processus étape par étape sur la façon de télécharger et d'installer Pycharm IDE sur Windows :

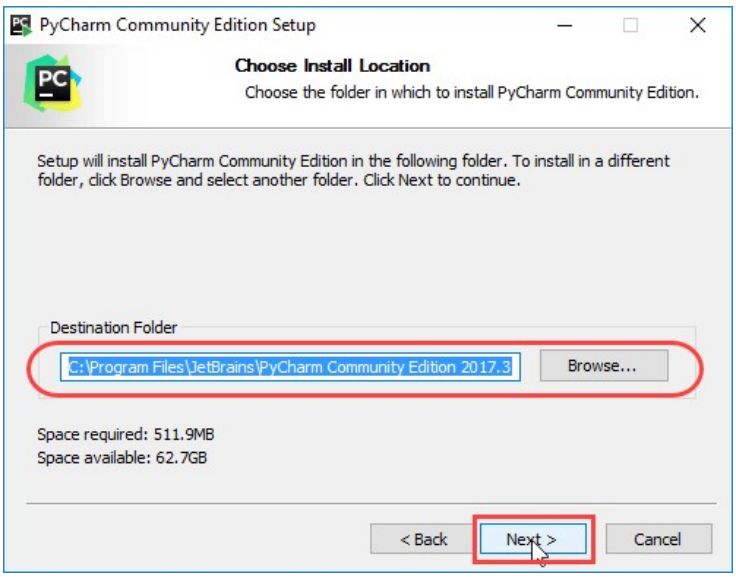
Étape 1) Pour télécharger PyCharm, visitez le site Web https://www.jetbrains.com/pycharm/download/ et cliquez sur le lien « TÉLÉCHARGER » sous la section Communauté.



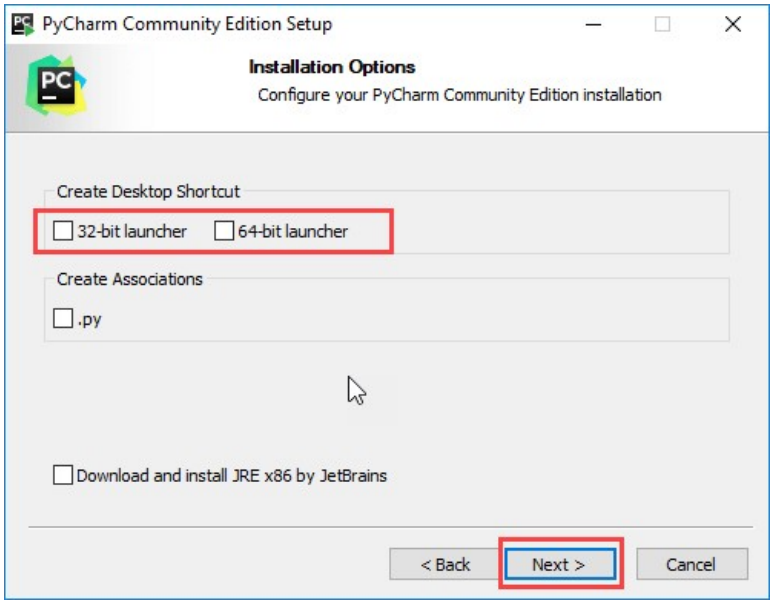
Étape 2) Une fois le téléchargement terminé, exécutez l'exe pour installer PyCharm. L'assistant de configuration aurait dû démarrer. Cliquez sur Suivant".



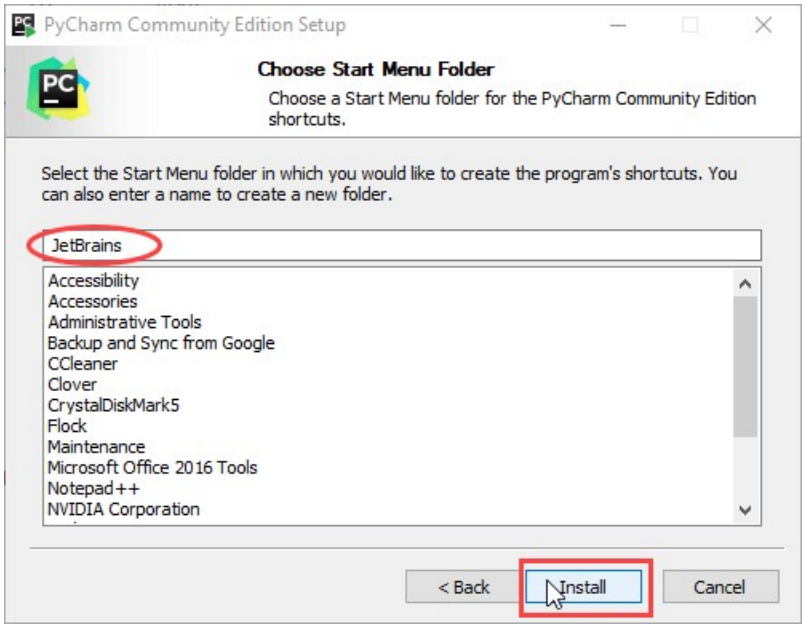
Étape 3) Sur l'écran suivant, modifiez le chemin d'installation si nécessaire. Cliquez sur Suivant".



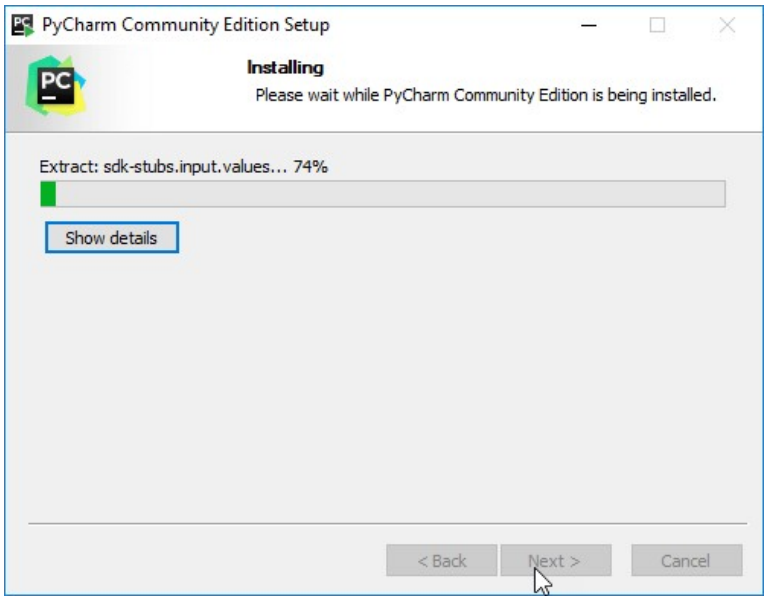
Étape 4) Sur l'écran suivant, vous pouvez créer un raccourci sur le bureau si vous le souhaitez et cliquer sur « Suivant ».



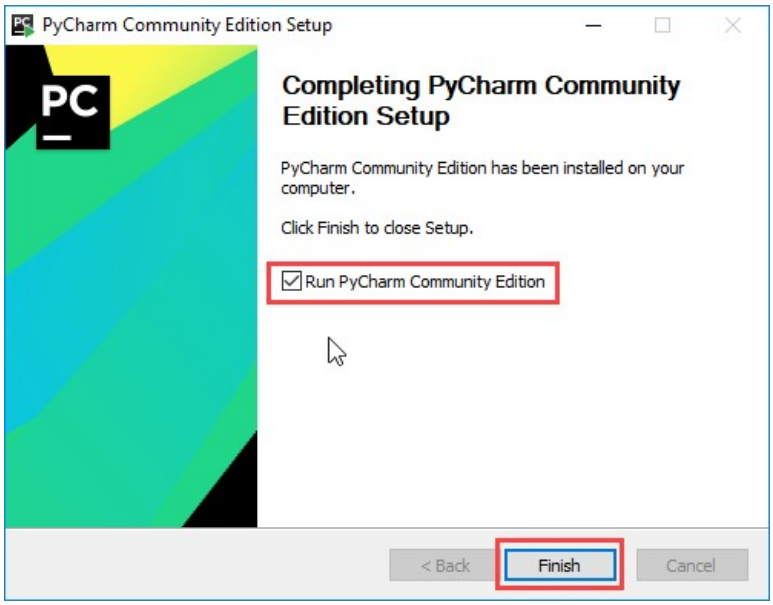
Étape 5) Choisissez le dossier du menu Démarrer. Conservez JetBrains sélectionné et cliquez sur « Installer ».



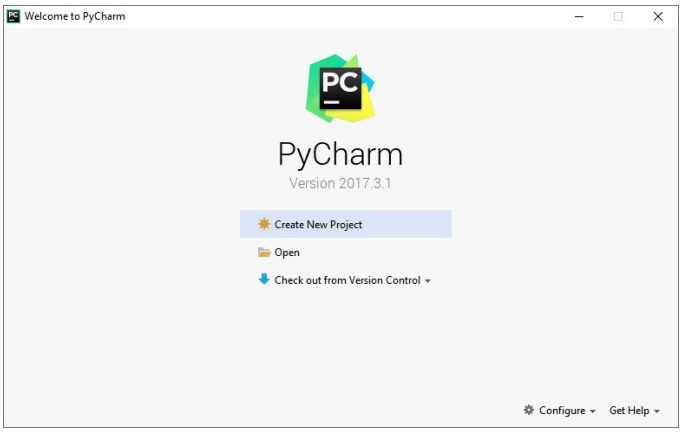
Étape 6) Attendez que l'installation se termine.



Étape 7) Une fois l'installation terminée, vous devriez recevoir un message indiquant que PyCharm est installé. Si vous souhaitez continuer et l'exécuter, cliquez sur « Exécuter PyCharm Community Edition » box d'abord et cliquez sur « Terminer ».



Étape 8) Après avoir cliqué sur « Terminer », le Following l'écran apparaîtra.



Installation d'opencv-contib

Comment installer OpenCV python sur Windows avec pip ?

Pour installer OpenCV en utilisant le gestionnaire de paquets par défaut de python, il vous suffit de taper dans votre windows powershell :

python -m pip install opencv-python-contrib

voir le https://kongakura.fr/article/installer-OpenCV-Python-3