1 Nombres et calculs

Calculer en Python

Calcul à la main	Calcul en Python	Résultat en Python
$2+3{,}5$	`2 + 3.5`	`5.5`
3×5	`3*5`	`15`
2^3	`2**3`	`8`
$125{,}3\times10^{-2}$	`125.3e-2`	`1.253`
$\lvert-2{,}6\rvert$	`abs(-2.6)`	`2.6`

Calcul à la main	Calcul en Python	Résultat en Python
$\frac{3}{8}$	`3/8`	`0.375`
Arrondir $\frac{3}{8}$ à 0,1 près	`round(3/8, 1)`	`0.4`
Division euclidienne de 154 par 6 (quotient et reste)	154 | 6 4 | 25 `154%6` `154//6`	

Types de nombres int et float

Dans un ordinateur, les nombres ne sont pas représentés par leur écriture décimale. On distingue de ce fait deux types de nombres en Python :
– les **entiers**, qui sont représentés de façon exacte. On dit qu'ils sont de type **int** ;
– les autres nombres, qui sont représentés de façon approchée : on dit que ce sont des **flottants** ou qu'ils sont de type **float**. Ils s'écrivent avec un point décimal.

ATTENTION !

- En maths, 4 et 4,0 sont deux écritures du même nombre.
- En Python, 4 et 4.0 ne sont pas le même objet.

Types de nombres et résultats de calcul

- Tout calcul comportant au moins un flottant se fait de façon approchée.

Exemple : `0.1 + 0.7` donne `0.7999999999999999`.

- Pour les entiers, cela dépend de l'opération :
– les additions, soustractions et calculs de puissance d'exposant entier se font de façon exacte, même sur de très grands entiers comme 2^{60} ci-contre ;
– une division a/b donne toujours un flottant ;
– l'utilisation des puissances de 10 avec ...e... donne toujours un flottant.

Exemples

- Le calcul `2**60` donne : `1152921504606846976`
- Le calcul `12/3` donne : `4.0`
- Le calcul `12e4` donne : `120000.0`

2 Variables numériques

Lors de l'exécution d'un programme, un ordinateur doit stocker des informations en mémoire, puis les retrouver. On crée pour cela une **variable** que l'on peut imaginer comme une boîte avec un nom, que l'on choisit, et qui contient une « valeur ».
Si cette valeur est un nombre, on parle de **variable numérique**.

Instruction	En Python	Effet
Affecter 6 à la variable m	`m = 6`	Créer la variable m (si elle ne l'était pas déjà) et lui affecter 6
Afficher la valeur de m	`print(m)`	Afficher le contenu de m, c'est-à-dire 6
Affecter le double de m à la variable n	`n = m*2`	Créer la variable n (si nécessaire) et lui affecter le double de la valeur de m, c'est-à-dire 12

Attention ! L'affectation ne s'écrit que dans un sens. On n'écrira pas `6 = m`. Le signe d'égalité n'a pas la même signification qu'en mathématiques.

Exemple :
Ci-contre, l'instruction `m = m*10` a pour effet :
– de calculer le produit de la valeur de m, c'est-à-dire 5, par 10 ;
– d'affecter le résultat 50 à m (la valeur précédente 5 est effacée).

NOMMER UNE VARIABLE

Donner un nom évocateur à une variable dans un programme en facilite la compréhension. Ce nom doit débuter par une lettre et peut contenir des lettres, des chiffres ou le caractère « _ », mais ne peut pas être un mot réservé du langage Python comme `print`, `if`, `for`, etc.

AVEC DEUX VARIABLES

On peut les traiter séparément ou simultanément en Python :
• L'instruction `a,b = 2,3` affecte en même temps 2 à a et 3 à b.
• L'instruction `print(a, b)` fait afficher les deux valeurs de a et b séparées par une espace.

```
1  m = 5
2  m = m*10
3  print(m)
```

On obtient : `50`.

3 Chaînes de caractères

On peut affecter à une variable une chaîne de caractères (une lettre de notre alphabet, un chiffre de 0 à 9, un signe de ponctuation, une parenthèse ou un crochet, une lettre accentuée, etc.). Cette chaîne sera écrite entre des apostrophes ou des guillemets.

Exemple :

On exécute le programme

```
1  pseudo = 'Lucas_0456'
2  print(pseudo)
```

On obtient : `Lucas_0456` (ou `'Lucas_0456'` selon l'outil utilisé).

Attention !

Bien différencier, pour une variable m, les instructions suivantes :
– `print(m)` fait afficher la valeur de m ;
– `print('m')` fait afficher la lettre m.

POUR ALLER PLUS LOIN

À la place des apostrophes, on peut utiliser :
– des **guillemets**. Ils sont parfois indispensables, par exemple quand la chaîne contient elle-même une apostrophe ;
– des **triples apostrophes ou des triples guillemets** pour conserver une mise en forme, par exemple une écriture sur plusieurs lignes.

Exemple

```
1  print("""Bonjour !
2  J'aime Python""")
```

On obtient l'affichage :

```
Bonjour !
J'aime Python
```

Concaténer des chaînes

Il s'agit d'assembler deux chaînes en une seule chaîne. L'opération utilisée est l'addition.

Exemples :

```
print('Bonjour' + 'Bob')
```
On obtient :
`BonjourBob`

```
print('Bonjour ' + 'Bob')
```
On obtient :
`Bonjour Bob`

NOTE

`2*'bla'` revient à écrire : `'bla'+'bla'`
En exécutant, `print(2*'bla')`, on obtient l'affichage : `blabla`

Obtenir la longueur d'une chaîne

`len()` donne la longueur de la chaîne, c'est-à-dire le nombre de caractères.

Extraire un caractère d'une chaîne

chaine`[i]` donne le $(i+1)$-ième caractère de *chaine*.

Exemples :

```
print(len('Lucas_0456'))
```
On obtient : `10`

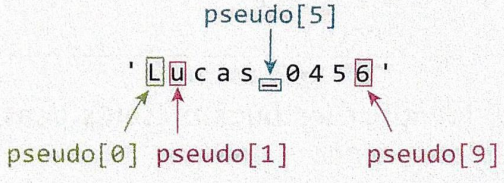
```
print(len('Lucas0456'))
```
On obtient : `9`

```
pseudo ='Lucas_0456'
print(pseudo[0])
```
On obtient : `L`

```
pseudo ='Lucas_0456'
print(pseudo[3])
```
On obtient : `a`

Attention !

Python commence à numéroter à 0 :



4 Fonctions int(), float(), str(), input() et format()

Types int, float et str

• Les nombres sont de deux types : `int` ou `float`. Une chaîne de caractères est du type `str` (*string* = chaîne).
• Les commandes `int()`, `float()` et `str()` transforment le type d'un objet.

Exemples : `str(4)` donne la chaîne `'4'` ; `int('3')` donne l'entier `3` ; `float('5.6')` donne le flottant `5.6`.
Il est important de s'intéresser au type d'un objet. Par exemple, si la variable `a` contient la chaîne `'5.6'`, le calcul `a + 2` provoque une erreur, car on ne peut pas ajouter une chaîne et un nombre.
Dans ce cas, on écrira `float(a) + 2` pour faire calculer `5.6 + 2`.

REMARQUE

La commande `type()` permet de faire afficher le type d'un objet.

Demander et attendre une réponse : input()

L'instruction `a = input('Donner un nombre :')` envoie la question « `Donner un nombre :` » à l'utilisateur, attend sa réponse et la stocke en tant que chaîne dans la variable `a`.
Pour stocker la réponse comme un nombre, entier ou flottant, sur lequel on pourra faire des calculs, on entre donc respectivement comme instruction :

```
a = int(input('Donner un nombre :'))
```
ou
```
a = float(input('Donner un nombre :'))
```

NOTE

En SNT, en enseignement scientifique et en mathématiques, pour pouvoir faire varier aisément certaines valeurs dans un programme, on peut aussi créer une fonction informatique (voir p. 18).

Afficher un message avec du texte et la valeur d'une variable numérique

On veut compléter la 4e ligne du programme ci-contre pour obtenir l'affichage suivant composé de textes et d'un nombre.

`La vitesse moyenne est de (valeur de v à 0,01 près) km/h.`

```
1  distance = 200  # en km
2  duree = 1.5     # en heure
3  v = distance/duree
4  print(........)
```

Pour compléter la ligne 4, il existe plusieurs méthodes, en voici trois :

• **Juxtaposer les affichages :**
```
print('La vitesse est de ', round(v, 2),' km/h')
```

• **Concaténer des chaînes de caractères :**
```
print('La vitesse est de ' + str(round(v, 2)) + ' km/h')
```
`str(round(v, 2))` transforme un nombre en chaîne que l'on peut concaténer avec d'autres chaînes.

• **Utiliser la méthode format() :**
```
print('La vitesse est de {:.2f} km/h'.format(v))
```
Les accolades marquent la place à laquelle la valeur de `v`, indiquée dans `format()`, est à afficher. À l'intérieur de ces accolades, les deux points `:` indiquent que l'on va imposer un format, `f` qu'il s'agit d'un flottant, et `.2` que la précision souhaitée est de 2 décimales.

NOTE

D'autres formats sont disponibles, comme `{:e}` qui impose l'écriture scientifique de l'affichage.

5 Structure conditionnelle : if ... else ... (si ... alors ... sinon ...)

Structure	En Python
Si ... alors ...	if *condition*: *instruction(s) à exécuter quand la condition est vraie*
Si ... alors ... sinon ...	if *condition*: *instruction(s) 1 à exécuter quand la condition est vraie* else: *instruction(s) 2 à exécuter quand la condition est fausse*

Tests possibles

En maths	En Python
$<$	<
$>$	>
$\leq$	<=
$\geq$	>=
$=$	==
$\neq$	!=

Attention !
On teste une égalité par un « double égal », soit ==. Le signe = est réservé à l'affectation.

Rôle de l'indentation (décalage vers la droite) :
En Python, l'indentation automatique des instructions est très importante. L'annulation de l'indentation marque la sortie de la structure conditionnelle.

Exemple 1 : Si ... alors ... (if ...)

```
10  if p > 50:
11      p = p - 10
12      print(p)
```

L'instruction `print(p)` est exécutée uniquement si `p > 50` est vraie.

• Quand p a pour valeur 80 avant exécution, on obtient après exécution l'affichage de 70.
• Quand p a pour valeur 30 avant exécution, les instructions des lignes 11 et 12 ne sont pas exécutées. On n'obtient donc aucun affichage.

Exemple 2 : Si ... alors ... (if ...)

```
10  if p > 50:
11      p = p - 10
12  print(p)
```

L'instruction `print(p)` est en dehors du si ... alors ..., elle est donc toujours exécutée.

• Quand p a pour valeur 80 avant exécution, on obtient après exécution l'affichage de 70.
• Quand p a pour valeur 30 avant exécution, l'instruction de la ligne 11 n'est pas exécutée, mais celle de la ligne 12 l'est. On obtient comme affichage 30.

Exemple 3 : Si ... alors ... sinon ... (if ... else ...)
Examinons deux cas pour l'extrait de programme ci-dessous :

```
10  if p > 50:
11      p = p - 10
12  else:
13      p = p - 5
14  print(p)
```

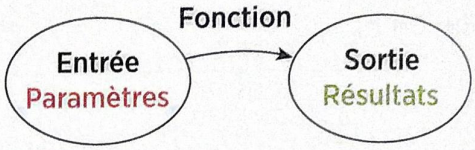
• Si, avant exécution, p a pour valeur 80, on exécute les instructions des lignes 11 puis 14. On obtient comme affichage 70.
• Si, avant exécution, p a pour valeur 30, on exécute les instructions des lignes 13 et 14. On obtient comme affichage 25.

POUR ALLER PLUS LOIN

Quand il y a plus de deux cas possibles, on peut éviter d'imbriquer des « Si ... alors ... » en utilisant `elif` (contraction de `else if` ... pour « Sinon si ... »).

6 Fonctions en Python

Une fonction informatique a une entrée (des informations qu'on lui fournit appelées paramètres ou arguments), exécute un travail sur ces paramètres et **renvoie** des informations en sortie, comme des nombres et/ou des messages.



Définition d'une fonction en Python

```
def nom de la fonction (paramètre1,paramètre2…) :
    instructions éventuelles
    return résultat1, résultat2…
```

REMARQUE

L'exécution de `return résultat` a deux effets :
- renvoyer le(s) résultat(s) obtenu(s) ;
- terminer l'exécution des instructions définissant la fonction.

Exemple :

• D'une formule à une fonction

La vitesse moyenne v est donnée par la formule $v = \frac{d}{t}$ avec d en m, t en s et v en $m \cdot s^{-1}$.

Associer la vitesse aux deux variables d et t, c'est créer une fonction qui à d et t associe la vitesse et que l'on peut noter $v(d, t)$ avec la notation fonctionnelle (comme $f(x)$ en mathématiques).

• Création de la fonction en Python

La fonction informatique créée en exécutant les lignes 1 et 2 du programme ci-contre a pour nom v et pour paramètres d et t.

```
1   def v(d,t):
2       return d/t
3
4   print(v(42,3.0))
```

• Faire afficher un résultat

`v(42,3.0)` renvoie le résultat du calcul de d/t pour d et t ayant pour valeurs respectives 42 et 3.0, donc `14.0`.
Exécuter `print(v(42,3.0))` en ligne 4 produit donc l'affichage `14.0`.
On en déduit que la vitesse moyenne est de 14 $m \cdot s^{-1}$.

• Utiliser une fonction dans un calcul

En ligne 4 du programme ci-contre, on utilise directement `v(42,3.0)` dans un calcul, sans l'avoir calculée auparavant.
Ec contient alors $0{,}5 \times 1\,000 \times 14{,}0^2 = 98\,000{,}0$.

```
1   def v(d,t):
2       return d/t
3
4   Ec = 0.5*1000*v(42,3.0)**2
```

Pourquoi utiliser des fonctions ?

Utiliser des fonctions permet :
– d'effectuer des opérations répétitives sans devoir saisir les instructions à chaque fois ;
– de découper un programme en parties plus simples et de le rendre plus lisible ;
– d'automatiser des calculs pour lesquels on ne dispose pas de formule algébrique.

7 Boucle for... : répéter quand on connaît le nombre de répétitions

Pour répéter des instructions un nombre de fois connu, on utilise une « boucle for » appelée « boucle bornée ».

Syntaxe d'une boucle bornée (*i*, *m* et *n* désignent des entiers avec $m < n$)

```
for i in range(m, n):
    instructions
```

Pour *i* allant de *m* à $n - 1$, on répète les ***instructions*** qui figurent dans le bloc **indenté** (décalé vers la droite).

`for i in … :`	***i* prend successivement les valeurs entières**
`range(5)`	de 0 à 4 donc 0, 1, 2, 3, 4 (attention : 5 est exclu)
`range(2, 8)`	de 2 à 7 donc 2, 3, 4, 5, 6, 7
`range(10, 17, 2)`	de 10 à 16 mais de 2 en 2, donc 10, 12, 14, 16

Exemple 1

```
a = 1
for i in range(2, 5) :
    a = i*a
    print(a)
```

On répète les instructions indentées pour *i* valant 2, puis 3, puis 4

```
a = 2*a
print(a)
a = 3*a
print(a)
a = 4*a
print(a)
```

Après exécution, on obtient l'affichage :
2
6
24

Importance de l'indentation (décalage vers la droite)

Exemple 2

```
a = 1
for i in range(2, 5):
    a = i*a
print(a)
```

- L'instruction `a = i*a` est indentée. Elle est répétée pour *i* prenant les valeurs 2, 3, puis 4, donc 3 fois.
- L'instruction `print(a)` n'est pas indentée, elle est en dehors de la boucle. Elle n'est exécutée qu'une seule fois quand l'exécution de la boucle est terminée.

Après exécution, on obtient l'affichage :

24

8 Boucle while... : répéter quand on connaît une condition d'arrêt

Syntaxe d'une boucle while appelée boucle non bornée

```
while condition:
    instructions
```

Tant que la *condition* est vraie (elle a pour valeur True), on répète les *instructions* qui figurent dans le bloc **indenté** (décalé vers la droite).

Exemple 1

```
a = 1
while a <= 8:
    a = 3*a
    print(a)
```

Exécutons « à la main » le programme de l'exemple 1 :

a prend la valeur 1
On effectue le test a ≤ 8 : True (Vrai)
    Donc a vaut 3 et on affiche 3
On effectue le test a ≤ 8 : True (Vrai)
    Donc a vaut 9 et on affiche 9
On effectue le test a ≤ 8 : False (Faux)
On sort de la boucle.

Après exécution, on obtient l'affichage :
3
9

Importance de l'indentation (décalage vers la droite)

Exemple 2

```
a = 1
while a <= 8:
    a = 3*a
print(a)
```

Dans l'exemple 2, l'instruction `print(a)` n'est pas indentée. Elle n'appartient pas au bloc d'instructions à répéter et n'est exécutée qu'une fois la boucle terminée. Seule la dernière valeur de a, c'est-à-dire 9, est affichée.

9 Modules (exemple du module random)

Nous avons déjà utilisé différentes fonctions Python comme `print()`, `type()`, etc.
De très nombreux outils sont disponibles en Python, certains sont rangés dans des modules (ou bibliothèques). Quand on veut utiliser un tel outil, on doit l'**importer** depuis le module. On a le choix entre importer cet outil seulement ou importer l'ensemble des outils du module. Plusieurs syntaxes possibles sont à connaître.

Exemple : on souhaite utiliser les fonctions `randint()` et `random()` du module *random*.
`randint(a, b)` simule le tirage au hasard d'un entier entre deux entiers *a* et *b* (*a* et *b* inclus) ;
`random()` simule le tirage au hasard d'un nombre compris entre 0 et 1 (0 inclus, 1 exclu).

Importer un seul outil

On peut utiliser l'outil plusieurs fois si nécessaire.

from *module* import *outil*

```
1  from random import randint
2  a = randint(1, 6)
3  b = randint(1, 6)
4  c = randint(1, 6)
5  print(a, b, c)
```

Après une 1re exécution, on peut obtenir par exemple l'affichage :
`3  1  3`

Après une 2e exécution :
`4  2  6`

et ainsi de suite.

Importer tous les outils d'un module

On peut ensuite utiliser n'importe quel outil du module.
Cette importation peut s'effectuer de plusieurs façons.

from *module* import*

```
1  from random import*
2  a = randint(1, 6)
3  b = randint(1, 6)
4  c = random()
5  print(a, b, c)
```

Après une 1re exécution, on peut obtenir par exemple l'affichage :
`5  2  0.1196377190163963`

Après une 2e exécution :
`5  1  0.9157830140796791`

et ainsi de suite.

import *module* (as *alias*)

```
1  import random
2  a = random.randint(1, 6)
3  b = random.randint(1, 6)
4  c = random.random()
5  print(a, b, c)
```

ou, en utilisant un alias comme rd pour raccourcir le nom du module :

```
1  import random as rd
2  a = rd.randint(1, 6)
3  b = rd.randint(1, 6)
4  c = rd.random()
5  print(a, b, c)
```

Remarque : dans la dernière méthode, le nom de la fonction doit être précédé du nom du module (ou de son alias). Cela permet de bien identifier la fonction, de savoir d'où elle vient. En effet, certaines fonctions ont le même nom dans différents modules, mais des utilisations un peu différentes.

En pratique, on utilisera principalement :
- **numpy** (voir p. 28). Il arrive qu'on lui donne un alias, en général **np** ;
- **matplotlib.pyplot** (voir p. 30) auquel on donne en général l'alias **plt** ;
- **random** (voir ci-dessus).

D'autres modules sont utilisés de façon plus épisodique. L'essentiel pour ces modules est de bien comprendre les syntaxes décrites ci-dessus.

LISTE DES OUTILS D'UN MODULE

Pour connaître tous les outils disponibles dans le module *module*, on exécute :

- dans la console :
`import module`
`dir(module)`

- dans un programme :
`import module`
`print(dir(module))`

10 Stocker des données

Pour traiter des relevés d'expérience, on a besoin de stocker de nombreuses mesures. Pour cela, on ne crée pas une nouvelle variable pour chaque valeur à stocker, mais des listes ou des tableaux.

A. Utiliser des listes

Créer une liste

Une liste se présente comme une suite d'éléments séparés par des virgules écrite entre crochets. Elle est de type `list`. On peut la créer de différentes façons.

Exemple : Voici trois façons de créer la liste [0, 10, 20, 30] et de la stocker dans la variable L.

`L = [0, 10, 20, 30]`	**On crée la liste en entrant chaque valeur à la main.**
`L = [10*i for i in range(4)]`	**Les éléments de la liste sont les produits `10*i` quand `i` prend les valeurs 0, 1, 2 et 3.**
`L = []` `for i in range(4):` `  L.append(i*10)`	**On crée la liste vide `[]`, puis on ajoute un par un les éléments.** `L.append(a)` **permet de modifier la liste `L` en lui ajoutant à la fin l'élément `a` ;** **la liste ainsi rallongée s'appelle toujours `L`.**

Remarques :

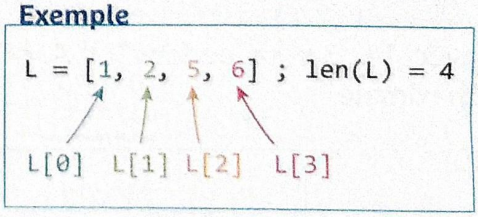
- On pourrait aussi créer `L` en écrivant `L = list(range(0, 40, 10))` où `list()` transforme `(0, 10, 20, 30)` en la liste `[0, 10, 20, 30]`.
- `append()` permet d'ajouter un élément à une liste.

Exemple : `L.append(40)` transforme L en `[0, 10, 20, 30, 40]`.
Attention, `L.append([40,50])` transforme L en `[0, 10, 20, 30, [40, 50]]`.

Accéder à un élément

- Pour une liste L, `L[i]` donne le (i + 1)-ième élément. Le premier terme est `L[0]`.
- `len(L)` renvoie le nombre d'éléments de L.
- Si L n'est pas vide, le dernier élément peut être obtenu par `L[-1]`.

Exemple



Opérations et fonctions

- L'addition de listes les concatène (comme pour les chaînes de caractères).

Exemple : `[1, 2, 3] + [2, 7]` donne `[1, 2, 3, 2, 7]`.

- On peut créer la liste des images par une fonction des éléments d'une liste donnée.

Exemple : on crée la liste L2 contenant les cosinus des éléments de L.

ATTENTION

`2*[1, 2, 3]` donne :
`[1, 2, 3, 1, 2, 3]`

```
1  from math import*
2  L = [1, 2, 3, 4]
3  L2 = [cos(t) for t in L]
```

B. Utiliser des tableaux et le module numpy

Créer un tableau (array)

		Exemple
On importe le module numpy par `import numpy as np`.		`import numpy as np`
`np.array([valeur1, ...])`	Crée le tableau en entrant chaque valeur à la main.	`np.array([0, 10, 20, 30])` crée le tableau : `[0, 10, 20, 30]`
`np.arange(b)`	Crée le tableau des entiers successifs strictement inférieurs à b (b > 0).	`np.arange(5)` crée le tableau : `[0, 1, 2, 3, 4]`
`np.arange(a, b, pas)`	Crée le tableau de valeurs de a à b (b exclu) espacées de pas.	`np.arange(0, 1, 0.2)` crée le tableau : `[0., 0.2, 0.4, 0.6, 0.8]`
`np.linspace(a, b, n)`	Crée le tableau de n valeurs de a à b régulièrement espacées (pas $= \frac{b-a}{n-1}$).	`np.linspace(0, 1, 5)` crée le tableau : `[0., 0.25, 0.5, 0.75, 1]`

Remarques :

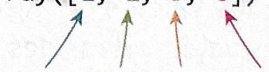
- Dans `np.arange(a, b, pas)`, a, b, pas ne sont pas nécessairement entiers, contrairement à `range()`.
- Les éléments d'un tableau sont tous de même type (dans les exemples ci-dessus, `np.arange(0, 1, 0.2)` ou `np.linspace(0, 1, 5)` donnent des tableaux de nombres tous de type float).

Accéder à un élément

De même que pour les listes, pour un tableau T de longueur *n*, les indices vont de 0 à $n-1$, soit `len(T) - 1`.
`T[i]` donne le (i + 1)-ième élément et `T[-1]` le dernier.

Exemple

```
T = np.array([1, 2, 5, 6]) ; len(T) = 4
```



Opérations et fonctions

- Contrairement aux listes, multiplier un tableau par 2 permet de multiplier chacun de ses éléments par 2.

On peut de même ajouter deux tableaux de nombres de même taille, ce qui revient à ajouter terme à terme les éléments de même indice.

- Contrairement aux listes, on peut appliquer directement à un tableau une fonction que l'on a définie ou une fonction du module numpy.

ATTENTION

`2*np.array[1, 2, 3]` donne : `[2, 4, 6]`

Exemple

En exécutant le programme suivant :

```
1  import numpy as np
2  t = np.array([3, 4, 5])
3  print(t**2)
```

On obtient l'affichage : `[9  16  25]`.

11 Graphiques avec matplotlib.pyplot

On importe le module `matplotlib.pyplot` avec l'alias `plt`.
Les fonctions importées de ce module seront donc précédées de `plt`.
On termine le programme par `plt.show()` pour faire afficher le graphique.

```
import matplotlib.pyplot as plt
    instructions pour les tracés
plt.show()
```

A. Nuages de points et courbes : plt.plot

L'instruction `plt.plot(x, y, 'r+')` permet de marquer par le signe + en rouge :
– le point de coordonnées $(x ; y)$, si x et y sont deux nombres ;
– tous les points ayant pour abscisses les éléments de x et pour ordonnées ceux de y de même indice, si x et y sont des listes ou des tableaux de nombres.

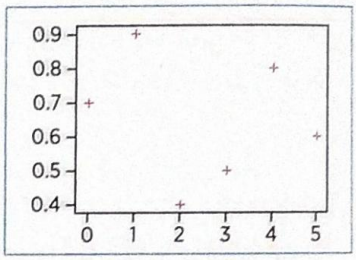
NOTE

Pour un nuage de points, on peut aussi utiliser **`plt.scatter()`**.

Exemple 1 : Nuage de points avec des listes x et y

- x est la liste des abscisses.
- y est la liste des ordonnées.
- Les points de coordonnées (0 ; 0,7), (1 ; 0,9), ..., (5 ; 0,6) sont marqués par un + rouge (`'r+'`).

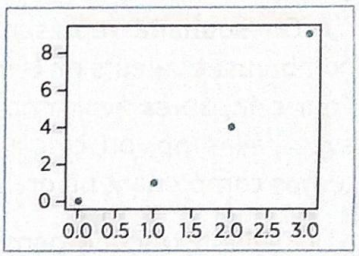
```
1  import matplotlib.pyplot as plt
2  x = [0, 1, 2, 3, 4, 5]
3  y = [0.7, 0.9, 0.4, 0.5, 0.8, 0.6]
4  plt.plot(x, y, 'r+')
5  plt.show()
```



Exemple 2 : Nuage de points avec une boucle

- x prend les valeurs 0, 1, 2 et 3.
- Les points de coordonnées (0 ; 0), (1 ; 1), (2 ; 4) et (3 ; 9) sont marqués par un rond bleu (`'bo'`).

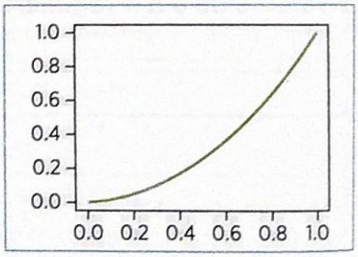
```
1  import matplotlib.pyplot as plt
2  for x in range(4):
3      plt.plot(x, x**2, 'bo')
4  plt.show()
```



Exemple 3 : Courbe avec numpy et un tableau

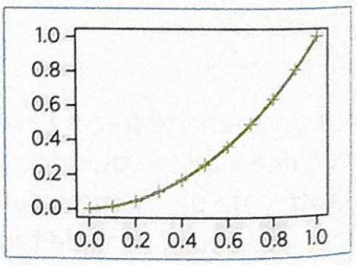
- On importe numpy pour créer le tableau x de 11 nombres de 0 à 1 avec un pas de 0,1.
- Les points de coordonnées $(0 ; 0^2)$, $(0{,}1 ; 0{,}1^2)$, ..., $(1 ; 1^2)$ sont reliés par un trait vert (`'g–'`).

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  x = np.linspace(0, 1, 11)
4  plt.plot(x, x**2, 'g–')
5  plt.show()
```



On peut préciser davantage de paramètres, en utilisant des formulations plus explicites que les raccourcis bien pratiques comme `'r+'` ou `'bo'`.
On obtient ainsi le graphique ci-contre en remplaçant la ligne 4 de l'exemple 3 par :

```
4  plt.plot(x, x**2, 'g+–', markersize = 10)
```



ou encore, pour que la courbe soit en pointillé :

```
4  plt.plot(x, x**2, linestyle = ':', linewidth = 2, color = 'g',
   marker = '+', markersize = 10)
```

Paramètres graphiques :

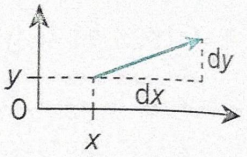
- **`color`** = **`'...'`** : précise la couleur. Les choix de couleurs sont : `'b'` ou `'blue'` ; `'g'` ou `'green'` ; `'r'` ou `'red'` ; `'c'` ou `'cyan'` ; `'m'` ou `'magenta'` ; `'y'` ou `'yellow'` ; `'k'` ou `'black'` ; `'w'` ou `'white'`.
- **`marker`** = **`'...'`** : donne le style de la marque, par exemple, `'+'` (signe +), `'o'` (rond), `'x'` (croix), `'s'` (*square* : carré), `'v'` ou `'>'` ou `'<'` (triangles).
- **`markersize`** = *nombre* : précise la taille de la marque.
- **`linestyle`** = **`'...'`** : précise le style de trait, par exemple `'none'` (pas de trait), `'-'` ou `'solid'` (trait continu), `'--'` ou `'dashed'`, `':'` ou `'dotted'` (trait discontinu).
- **`linewidth`** = *nombre* : précise l'épaisseur du trait.

NOTE

`plt.plot(x, y, linestyle = 'none', color = 'black', marker = '+')`
a le même effet que :
`plt.plot(x, y, 'k+')`

D. Représentation d'un vecteur : matplotlib.pyplot

Pour tracer le vecteur de coordonnées (dx ; dy) à partir du point de coordonnées (x ; y), on utilise l'instruction :

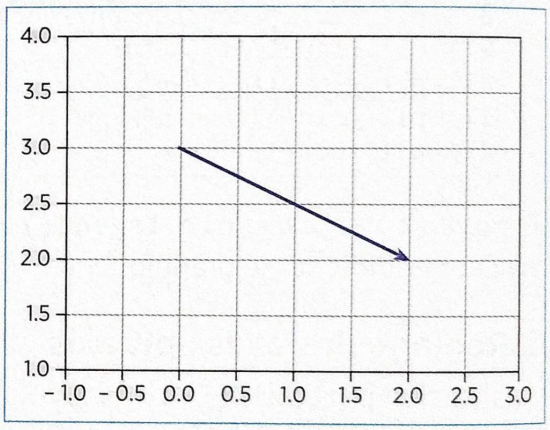



```
plt.quiver(x, y, dx, dy, angles = 'xy', scale = 1, scale_units = 'xy')
```

Ces réglages, à ne pas modifier, assurent la cohérence de la direction du vecteur tracé et de ses dimensions avec le repère et les unités sur les axes.

Exemple

```
1   import matplotlib.pyplot as plt
2   plt.quiver(0, 3, 2, -1, color = 'b', angles = 'xy',
    scale = 1, scale_units = 'xy')
3   plt.axis([-1, 3, 1, 4])
4   plt.grid()
5   plt.show()
```



Remarque : il est préférable d'utiliser `plt.quiver()` mais `plt.arrow()` peut aussi être utilisé, avec des réglages pour éviter des problèmes de positionnement et de dimensions par rapport aux axes.

```
plt.arrow(x, y, dx*echelle, dy*echelle, head_width = 0.05, length_includes_head = True)
```

Largeur de la pointe de la flèche

La pointe de la flèche est incluse dans les dimensions données.

12 Analyser des données numériques

A. Calculer la moyenne et l'écart-type d'une série numérique

Pour une série de données brutes :
– la moyenne est obtenue en divisant la somme des valeurs de la série par le nombre de valeurs ;
– l'écart-type est en revanche long à calculer à la main, et s'obtient toujours avec un outil (calculatrice, tableur...).

Exemple

Série : 2,49 ; 2,51 ; 2,47 ; 2,49 ; 2,48 ; 2,51 ; 2,53.

$$\text{Moyenne} = \frac{2,49+2,51+2,47+2,49+2,48+2,51+2,53}{7}$$

Exemple 1 : Moyenne avec les fonctions sum() et len()

```
1   X = [2.49, 2.51, 2.47, 2.49, 2.48, 2.51, 2.53]
2   somme = sum(X)
3   n = len(X)
4   moy = somme/n
5   print('Moyenne =', round(moy, 2))
```

- X est la **liste** des valeurs de la série.
- **sum**(X) calcule la somme des valeurs de X.
- **len**(X) calcule le nombre de valeurs de X.
- On obtient l'affichage :

```
Moyenne = 2.5
```

Remarques :
- len est le début de *length* et len(X) est la longueur de la liste X.
- L'arrondi à 0,01 près de la moyenne étant 2,50, Python affiche comme résultat 2.5. Il est possible de forcer l'affichage des 2 décimales par exemple en remplaçant la ligne 5 par : `print("Moyenne = %.2f" % moy)`

Exemple 2 : Moyenne avec la fonction mean() du module numpy

```
1   import numpy as np
2   Y = np.array([2.49, 2.51, 2.47, 2.49, 2.48, 2.51, 2.53])
3   moy = np.mean(Y)
4   print('Moyenne =', round(moy, 2))
```

- On importe le module **numpy** (voir p. 28).
- Y est le tableau des valeurs de la série.
- **np.mean**(Y) calcule la moyenne des valeurs de Y.
- On obtient l'affichage :

```
Moyenne = 2.5
```

Exemple 3 : Moyenne et écart-type expérimental avec les fonctions mean() et stdev() du module statistics

```
1   import statistics as sta
2   Z = [2.49, 2.51, 2.47, 2.49, 2.48, 2.51, 2.53]
3   moy = sta.mean(Z)
4   print('Moyenne =', round(moy, 2))
5   ecart = sta.stdev(Z)
6   print('Ecart-type expérimental = ', round(ecart, 2))
```

- On importe le module **statistics** avec l'alias **sta**.
- Z est la liste des valeurs de la série.
- **sta.mean**(Z) calcule la moyenne des valeurs de Z.
- On obtient l'affichage :

```
Moyenne = 2.5
```

- **sta.stdev**(Z) calcule l'écart-type expérimental.
- On obtient l'affichage :

```
Ecart-type expérimental = 0.02
```

B. Réaliser un ajustement polynomial

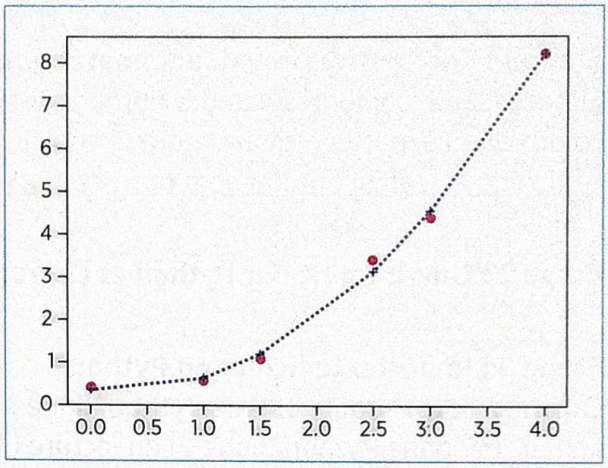
Situation :

On dispose de deux séries de données : la série $x_1, x_2, ..., x_n$ et la série $y_1, y_2, ..., y_n$ ayant le même nombre de valeurs numériques. On cherche une relation exprimant y en fonction de x, de façon approchée. De nombreux cas peuvent se ramener à un ajustement polynomial.

Exemple 1 : Ajustement polynomial de degré 2 avec la fonction polyfit() de numpy

Le nuage de points représentant les séries x et y est formé des points en rouge ci-contre. Compte tenu de son allure, on envisage une modélisation par un polynôme de degré 2 : $y = ax^2 + bx + c$.

Le programme suivant calcule et affiche les coefficients a, b et c et affiche le graphique ci-contre.




```
1   import matplotlib.pyplot as plt
2   import numpy as np
3
4   x = np.array([0, 1, 1.5, 2.5, 3, 4])
5   y = np.array([0.4, 0.55, 1.1, 3.4, 4.4, 8.2])
6   plt.plot(x, y, 'ro')
7
8   mod = np.polyfit(x, y, 2)
9   print(mod)
10
11  ymodel = mod[0]*(x**2) + mod[1]*x + mod[2]
12  plt.plot(x, ymodel, 'b+:')
13  plt.show()
```

- Les lignes `1` à `6` permettent de représenter le nuage de points par des ronds rouges.
- En ligne `8`, **`np.polyfit(x, y, 2)`** fournit un ajustement polynomial de degré 2 : $y = ax^2 + bx + c$.
- La ligne `9` produit l'affichage du tableau des coefficients a, b, c dans cet ordre :

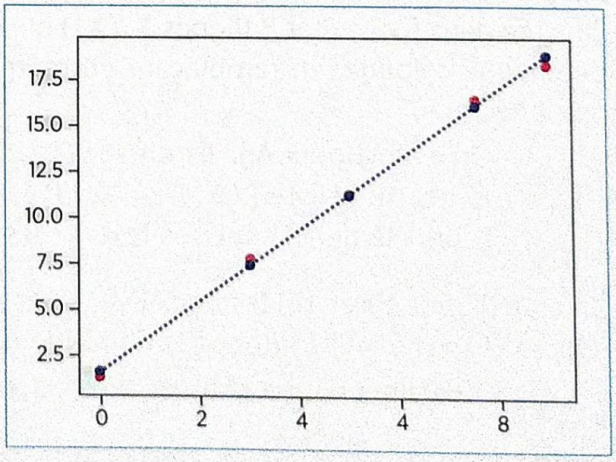
`[0.56428571  -0.2952381  0.35416667]`

- En ligne `11`, `ymodel` contient les images des valeurs de x dans ce modèle.
- On représente, par la ligne `13`, les points obtenus dans ce modèle, en bleu, reliés par une ligne brisée.

Exemple 2 : Ajustement par une relation affine

Quand le nuage de points (en rouge) est allongé comme ci-contre, on peut chercher un ajustement par une relation $y = ax + b$, donc par un polynôme de degré 1.

Alors `polyfit(x, y, 1)` fournit des valeurs de a et b.



Régression linéaire

La fonction **`linregress()`** du module statistique de `scipy` permet d'obtenir un ajustement par une relation affine.

Dans l'exemple ci-contre, l'affichage obtenu de a et b est :

`1.9458984375  1.5250976562500007`

Remarque : linregress() renvoie 5 valeurs, mais seules les deux premières nous intéressent.

```
1   import numpy as np
2   from scipy.stats import linregress
3
4   x = np.array([0, 3, 5, 7.5, 9])
5   y = np.array([1.2, 7.7, 11.3, 16.6, 18.5])
6   (a, b, _, _, _) = linregress(x, y)
7   print(a, b)
```

13 Importer des données expérimentales en Python

Lorsque les données à traiter sont issues de logiciels (Arduino®, Regressi, etc.), il faut les importer dans un programme Python dans des listes ou des tableaux pour pouvoir les exploiter.
Nous considérerons dans la suite les données, présentées en deux colonnes, de l'exemple ci-contre, avec les valeurs de x en colonne A et celles de y en colonne B.

A	B
0,2	0,5
3	2,8
4,5	8
8,2	12,6

Étape 1 : Enregistrer les valeurs numériques dans un fichier .csv
Si nécessaire, copier les valeurs numériques seules dans un tableur (sans les titres de ligne ou de colonne) puis, par Enregistrer sous..., choisir le format *CSV(separateur : point-virgule)*.
Le fichier .csv ainsi obtenu sera nommé donnees par la suite.

Étape 2 : Ouvrir un fichier Python et l'enregistrer dans le même dossier que le fichier .csv créé

Étape 3 : Importer le fichier en Python
On importe le module `csv` de Python (ligne 1), puis le fichier **.csv** dans Python (ligne 2) en lecture (`'r'` pour *read*) et on le nomme d.

On lit (ligne 3) ce fichier d dans lequel le séparateur est le point-virgule. On obtient un nouveau fichier nommé D.

(Un fichier **.csv** est un fichier contenant des textes, des nombres et des caractères de séparation qui peuvent être de différents formats. Seul le séparateur point-virgule est accepté par Python.)

```
1   import csv
2   d = open('donnees.csv', 'r')
3   D = csv.reader(d, delimiter = ';')
4
5   A = []
6   B = []
7   for ligne in D:
8       a = ligne[0].replace(',' , '.')
9       A.append(a)
10      b = ligne[1].replace(',' , '.')
11      B.append(b)
12
13  X = [float(x) for x in A]
14  Y = [float(y) for y in B]
15
16  d.close()
```

Étape 4 : Obtenir deux listes exploitables en Python
On crée deux listes A et B (lignes 5 à 11) qui contiennent les données respectives de chaque colonne, en remplaçant en même temps la virgule décimale par un point décimal.

On a alors pour A la liste `['0.2', '3', '4.5', '8.2']`
et pour B la liste `['0.5', '2.8', '8', '12.6']`.
Leurs éléments sont des textes (des chaînes de caractères).

PROGRAMME TYPE
Le programme ci-dessus est exploitable pour tout autre fichier analogue en changeant juste le nom du fichier (*donnees*) en ligne 2.

On crée (lignes 13 et 14) les deux listes X et Y de nombres correspondantes.
Rappel : `float('4.5')` donne le nombre flottant 4.5.

On a alors pour X la liste `[0.2, 3.0, 4.5, 8.2]` et pour Y la liste `[0.5, 2.8, 8.0, 12.6]`.

Remarque :
On peut aussi obtenir pour X et Y des tableaux de nombres :
– ajouter au début du programme l'importation de numpy par : `import numpy as np`
– remplacer les lignes 13 et 14 par : `X = np.array([float(x) for x in A])`
`Y = np.array([float(y) for y in B])`
On a alors pour X le tableau `[0.2, 3., 4.5, 8.2]` et pour Y le tableau `[0.5, 2.8, 8., 12.6]`.

Étape 5. Fermer d (ligne 16)
On peut exploiter les listes ou tableaux X et Y comme d'habitude.