

Introduction à Python pour l'analyse de données

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue and others in grey.

Bonjour à tous !

Florent COLLOT

PROFIND

<https://www.profind.net>

A decorative network diagram in the bottom-right corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue and others in grey.



Table des matières

<u>Introduction</u>	3
<u>Python</u>	18
<u>Programmation Orientée Objet (POO)</u>	116
<u>Debugging et Tests</u>	135
<u>Analyse de données avec Numpy et Pandas</u>	158

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

1.

Introduction

A decorative network diagram in the bottom-right corner, similar to the one in the top-left, showing a cluster of interconnected nodes and lines. The nodes are small circles, some with concentric circles, and the lines are thin and gray.



Nom

Florent COLLOT

Email

florent.collot@outlook.com

Activité

Freelance Consultant / Formateur

Spécialisation

Data Science / Développement Applicatif

Diplôme

Master en Informatique (SUPINFO)



Description et Objectifs de la formation

Description

Cette formation présente les **fondamentaux** de la **programmation** en mettant l'accent sur l'**analyse des données** via le langage de programmation **Python**.

Objectifs

Cette formation a pour objectifs :

- la **maitrise** des **fondamentaux** du langage de programmation **Python**.
- Etre en **mesure d'exploiter** les librairies de Data Science **Numpy**, **Pandas** et **Matploplib**.



Pré-requis

Des connaissances de base en informatique.



Planning

3 jours de 9h00 à 12h00 et de 13h00 à 17h00.

L'histoire de Python

1989 Création du langage **Python** par **Guido Van Rossum**.

2000 Sortie de la **version 2.0** de **Python**.

2001 Association du langage **Python** à la **Python Software Foundation**.

2008 Sortie de la **version 3.0** de **Python**.



Les versions de Python

Version 2

N'est **plus supportée** depuis le **1^{er} janvier 2020**.

En revanche, elle est toujours présente dans les systèmes existants.

Version 3

Nous sommes actuellement à la version **3.12** de **Python**.

Il est **recommandé** d'utiliser la **version 3** de python pour les nouveaux développements.

Téléchargement et installation de Python

Vous pouvez vous référer à l'excellente [documentation officielle](#) de Python.

Environnement de développement

Les **trois principaux IDE** pour développer en **Python** sont :

- **Visual Studio Code** : IDE gratuit de Microsoft.
- **PyCharm** : IDE gratuit avec une version payante de JetBrains.
- **Spyder** : IDE gratuit et open source orienté pour la Data Science.

Les caractéristiques de Python

Python est un **langage** :

- **Interprété** et **compilé à la volée**, avec les modules **C**.
- **Typage dynamique fort**, ainsi il n'est **pas nécessaire** de **spécifier** le **type** des **variables**.
- **Orienté objet** (mais pas seulement).
- **Portable** car **compatible** avec **toutes les plateformes** actuelles.
- **Flexible**, il est utilisé de l'administration système au développement web.
- **Populaire**, il est dans le **Top 5** des **langages** les **plus utilisés** depuis des années.

L'indentation

L'**indentation** désigne les **espaces ou tabulations** situés au **début** d'une **ligne** de code.

Alors que dans d'autres langages de programmation, l'**indentation** du code ne sert qu'à **faciliter** la **lecture**, l'indentation en Python est très importante. **Python** utilise l'**indentation** pour **délimiter** les **blocs de code**.

Les avantages et inconvénients de Python

Forces

Stable

Cross-plateforme

Facile à apprendre

Grande communauté

Grand nombre de module

Faiblesses

Pas entièrement compilé, donc plus lent

L'optimisation est complexe à apprendre

Les plateformes

Il existe **différents interpréteurs** pour **Python** :

- CPython/Pypy \Rightarrow C/C++
- Jython \Rightarrow JVM
- IronPython \Rightarrow .Net

Les domaines d'exploitation de Python

Les **domaines d'application** de **Python** :

- **Sciences** : Data mining, Machine Learning, Physiques, Mathématiques, ...
- **OS** : Linux, Raspberry Pi, scripting pour l'administration système.
- **Education** : Introduction à la programmation.
- **Web** : Django, Flask, ...
- **3D CAD** : FreeCAD, pythonCAD, ...
- **Multimédia** : Kodi, ...



**Avez-vous des
questions ?**

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, while others are smaller and solid. The lines connecting them are thin and grey, creating a mesh-like structure.

2.

Les fondamentaux de Python

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being larger and having concentric circles, and others being smaller and solid. The lines are thin and grey.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark gray, while others are hollow with a light gray outline. The lines connecting them are thin and light gray, creating a mesh-like structure.

2.1

Les variables et les types de bases

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines. The nodes are a mix of solid dark gray circles and hollow light gray circles, with thin light gray lines connecting them in a non-uniform, organic pattern.

Les variables

Une variable permet de **stocker en mémoire**, le temps que le programme s'exécute, des **données**.

Pour **stocker** en mémoire une **valeur** dans une **variable**, on utilise le **signe =**.

Exemple :

```
nomVariable = 1
```

Le nommage des variables

Il y a deux règles à respecter en ce qui concerne le nommage des variables :

- Les noms doivent commencer par une lettre minuscule, une lettre majuscule ou un underscore.
- Les noms doivent contenir uniquement les éléments précédents ainsi que des chiffres.

ATTENTION, il est donc **interdit** d'**utiliser** des **espaces**.

Les types en Python

Les types de base en Python :

- **int** : Un entier
- **float** : Un réel
- **str** : Une chaîne de caractères
- **bool** : Un booléen

Exemples :

- `compteur = 0`
- `pourcentageReduction = 3.5`
- `motATrouver = 'code'`
- `motEstTrouvé = True`

La fonction `type`

La fonction `type` permet de **connaître** le **type** d'une **variable**.

Exemple :

```
question = "Quelle est la réponse à l'univers ?"  
answer = 42  
type(question) ← <class 'str'>  
type(answer) ← <class 'int'>
```


La conversion de type

Nous pouvons utiliser les classes des types de base pour **convertir** des **valeurs** en **leur type**.

Exemple :

`int(42.5)` ← 42

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark grey, while others are hollow with a light grey outline. The lines connecting them are thin and light grey, creating a dense, organic structure that tapers off towards the right.

2.2

Les opérateurs

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being solid dark grey and others being hollow with a light grey outline. The lines are thin and light grey, forming a complex, interconnected web that tapers off towards the left.

Les opérateurs arithmétiques

`nomVariable = élément1 opérateur élément2`

Les opérateurs arithmétiques en Python :

- `+` : Addition
- `-` : Soustraction
- `/` : Division
- `//` : Division Euclidienne
- `*` : Multiplication
- `**` : Puissance
- `%` : Modulo

Exemples :

- `compteur = compteur + 1`
- `prime = salaire / 10`
- `resultat = var1 * 8 + var2 - var3`

Notation raccourcie

`nomVariable` opérateur= élément1

Lorsqu'on affecte le résultat de notre opération sur la même variable, on peut utiliser une notation raccourcie.

Exemples :

- `a += 8` est équivalent à `a = a+8`

- `c %= 2` est équivalent à `c = c%2`

Les opérateurs de comparaison

élément1 opérateur élément2

Les opérateurs de comparaison :

- **>** : Strictement supérieur
- **>=** : Supérieur ou égal
- **<** : Strictement inférieur
- **<=** : Inférieur ou égal
- **==** : Egal
- **!=** : Différent

Exemples :

- | | |
|-----------|----------|
| - 9 >= 10 | est faux |
| - 5 > 5 | est faux |
| - 5 != 10 | est vrai |

Les opérateurs logiques

ET (and)

A	B	A and B
1	1	1
1	0	0
0	1	0
0	0	0

OU (or)

A	B	A or B
1	1	1
1	0	1
0	1	1
0	0	0

NON (not)

A	not A
1	0
0	1

Exemples :

- not(5 <= 10) and (3 == 3) est faux
- (10 != 10) or (5 > 3) est vrai
- (10 != 10) and (5 > 3) est faux
- not(10 != 10 and 5 > 3) est vrai



2.3

Les structures conditionnelles



If

if condition:
instructions

Les **instructions** du **bloc if** seront **exécutées** uniquement si la **condition** est **vraie**. Dans le cas contraire, les instructions du bloc ne seront pas exécutées.

Exemple :

```
if age < 18:  
    print("Vous êtes mineur.")
```


If ... else

```
if condition:  
    instructions  
else:  
    instructions
```

Les **instructions** du **bloc if** seront **exécutées** uniquement si la **condition** est **vraie**. Dans le cas contraire, les instructions du **bloc else** seront **exécutées**.

Exemple :

```
if age < 18:  
    print("Vous êtes mineur.")  
else:  
    print("Vous êtes majeur.")
```

Imbrication des instructions if ... else

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

Les **instructions** du **bloc if** seront **exécutées** uniquement si la **condition1** est **vraie**. Dans le cas contraire, si la **condition2** est **vraie**, les **instructions** du **bloc elif** seront **exécutées**. Sinon, les instructions du **bloc else** seront **exécutées**.

Exemple :

```
if note < 8:  
    print("Vous avez rate votre BAC.")  
elif note < 10:  
    print("Vous devez passer les rattrapages.")  
else:  
    print("Vous avez votre BAC.")
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The lines connecting them are thin and grey, creating a dense, organic structure.

2.4

Les entrées/sorties

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being solid grey and others hollow with a grey outline. The overall structure is a complex, interconnected web.

Afficher une valeur

Pour afficher un message, on utilise la fonction **print**.

```
print("texte", variable)
```

Exemple :

```
print("Age")  
print(age)  
print("Vous avez", age, "ans")
```

Les arguments de la fonction **print**

- **sep** : Caractères affichés entre chaque argument. Par défaut **' '**.
- **end** : Caractères affichés après le dernier argument. Par défaut **'\n'**.
- **file** : Flux de sortie. Par défaut **sys.stdout**.
- **flush** : Doit-on vider le tampon ? Par défaut **False**.

Exemple :

```
print("Hello", "World", sep=', ', end="!\n")
```



Hello, World!

Les strings formatées

Il est possible de construire des **strings complexes** à l'aide de la fonction **format**:

Exemple :

```
"My name is {lastname}, {firstname} {lastname}.".format(  
    firstname="James",  
    lastname="Bond")
```

└───────────> My name is Bond, James Bond.

Les strings formatées


La fonction **format** dispose d'un raccourci pratique :

Exemple :

```
firstname="James"
```

```
lastname="Bond"
```

```
f"My name is {lastname}, {firstname} {lastname}."
```



My name is Bond, James Bond.

Récupérer une valeur

Pour récupérer une valeur, on utilise la fonction **input**.

```
variable = input("texte")
```

Exemple :

```
print("Donner votre nom : ")  
nom = input()
```


Les séquences d'échappement

Les sequences d'échappement sont :

- `\n` : Nouvelle ligne
- `\r` : Retour chariot
- `\f` : Nouvelle page
- `\b` : Retour arrière
- `\t` : Tabulation horizontale
- `\v` : Tabulation vertical
- `\a` : Bip machine

Récupérer une valeur

La fonction **input**, renvoie par défaut un type **str**. Ce comportement est un soucis lorsqu'on demande à l'utilisateur un nombre. Pour obtenir le type que l'on souhaite, il est necessaire d'utiliser la fonction **eval**.

```
variable = eval(input("texte"))
```

Exemple :

```
age = eval(input("Donner votre age : "))
```



**Avez-vous des
questions ?**





Exercices

Les variables

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are larger and have concentric circles, while others are smaller and solid. The edges are thin lines connecting these nodes, creating a dense, branching structure.

2.4

Les structures séquentielles

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are larger and have concentric circles, while others are smaller and solid. The edges are thin lines connecting these nodes, creating a dense, branching structure.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark gray, while others are hollow with a light gray outline. The lines connecting them are thin and light gray, creating a dense, organic structure.

2.4.1

Les séquences





Définition


Une **séquence** est un **regroupement** au sein d'une même **variable** de **plusieurs valeurs**. Ces **valeurs** seront **accessibles** par leur **position**.





Objectif

Une **séquence** a pour **objectif** d'**optimiser** certaines **opérations** tel que la recherche d'un élément, le tri de ces valeurs, le calcul de leur maximum.



Accès à un élément

L'**accès** à un **élément** d'une **séquence** se fait à l'aide de la **position** de cet élément et de l'**opérateur crochets** `[]`.

`maSéquence[position]`

ATTENTION, la position commence à **0**.

Les 3 principaux types de séquences

- Les **listes** dont les éléments sont **quelconques** et **modifiables**.
- Les **t-uples** dont les éléments sont **quelconques** et **non modifiables**.
- Les **chaînes de caractères** dont les éléments sont des **caractères** et **non modifiables**.

Les opérations communes aux séquences

Opération	Résultat
$x \text{ in } s$	Teste si x appartient à s
$x \text{ not in } s$	Teste si x n'appartient pas à s
$s + t$	Concaténation de s et t
$s * n$ ou $n * s$	Concaténation de n copies de s
$\text{len}(s)$	Nombre d'éléments de s
$\text{min}(s)$	Plus petit élément de s
$\text{max}(s)$	Plus grand élément de s
$s.\text{count}(x)$	Nombre d'occurrences de x dans s
$s.\text{index}(x)$	Indice de x dans s



2.4.2

Les listes



Déclaration d'une liste

```
maListeVide = []
```

```
maListeVide = list()
```

```
maListeAvecUnElement = [valeur]
```

```
maListe = [valeur1, valeur2, valeur3]
```

Les listes sont **muables**

- Les **listes** sont **modifiables**, ainsi il est **possible** de **modifier**, **supprimer** ou **ajouter** des éléments.
- Une **fonction** qui a en **paramètre** une **liste**, sera en mesure de la **modifier**.

Les opérations propres aux listes

Opération	Résultat
<code>list(s)</code>	Transforme une séquence s en une liste
<code>s.append(x)</code>	Ajoute l'élément x à la fin de s
<code>s.extend(t)</code>	Étend s avec la séquence t
<code>s.insert(i,x)</code>	Insère l'élément x à la position i
<code>s.clear()</code>	Supprime tous les éléments de s
<code>s.remove(x)</code>	Retire l'élément x de s
<code>s.pop(i)</code>	Renvoie l'élément d'indice i et le supprime
<code>s.reverse()</code>	Inverse l'ordre des éléments de s
<code>s.sort()</code>	Trie les éléments de s par ordre croissant

Les opérations propres aux listes

```
maListe = [1, 3, 5]
```

```
maListe.append(7) → [1, 3, 5, 7]
```

```
maListe.extend((8, 11)) → [1, 3, 5, 7, 8, 11]
```

```
maListe.remove(8) → [1, 3, 5, 7, 11]
```

```
maListe.insert(4, 9) → [1, 3, 5, 7, 9, 11]
```


Les operations propres aux listes

```
maListe = list(range(0, 11, 2))
```

↳ [0, 2, 4, 6, 8, 10]

```
taListe = list('Wakanda Forever')
```

↳ ['W', 'a', 'k', 'a', 'n', 'd', 'a', ' ', 'F', 'o', 'r', 'e', 'v', 'e', 'r']



2.4.3

Les t-uples

Déclaration d'un t-uples

```
monTupleVide = ()
```

```
monTupleVide = tuple()
```

```
monTupleAvecUnElement = (valeur)
```

```
monTuple = (valeur1, valeur2, valeur3)
```

Les t-uples sont **immuables**

- Les **t-uples** ne sont **pas modifiables**, ainsi il est **impossible** de **modifier**, **supprimer** ou **ajouter** des éléments.
- Une **fonction** qui a en **paramètre** un **t-uple**, ne sera **pas** en mesure de le **modifier**.

Les opérations propres aux t-uples

Opération	Résultat
<code>tuple(s)</code>	Transforme une séquence s en un t-uple

```
monTuple = tuple(range(0, 11, 2))  
↳ (0, 2, 4, 6, 8, 10)
```

Les intérêts des t-uples

- Si l'on souhaite définir une **séquence non modifiable**, utiliser un **t-uple sécurise** votre code (par exemple, définir la largeur et longueur de votre fenêtre).
- **Itérer** sur les éléments d'un **t-uple** est **plus rapide** que sur ceux d'une **liste**.
- Une **fonction** qui **retourne** « **plusieurs valeurs** », retourne en fait un **t-uple**.



2.4.4

Le slicing



L'accès aux éléments (slicing)

Le **slicing** peut être appliqué sur **toutes** les **séquences**.

Opération	Résultat
<code>s[i]</code>	i -ème élément de s
<code>s[i:j]</code>	Sous-séquence de s constituée des éléments entre le i -ème (inclus) et le j -ème (exclus)
<code>s[i:j:k]</code>	Sous-séquence de s constituée des éléments entre le i -ème (inclus) et le j -ème (exclus) pris avec un pas de k

L'accès aux éléments (slicing)

`monTuple = (10, 20, 30, 40, 50, 60)`

`monTuple[3:]` \longrightarrow `(40, 50, 60)`

`monTuple[1:4]` \longrightarrow `(20, 30, 40)`

`monTuple[1::2]` \longrightarrow `(20, 40, 60)`

Modification à l'aide du “slicing”

La **modification** à l'aide du **slicing** peut être appliqué **uniquement** sur les **listes**.

Opération	Résultat
$s[i] = x$	Remplacement de l'élément $s[i]$ par x
$s[i:j] = t$	Remplacement des éléments de $s[i:j]$ par ceux de la séquence t
$\text{del}(s[i:j])$	Suppression des éléments de $s[i:j]$
$s[i:j:k] = t$	Remplacement des éléments de $s[i:j:k]$ par ceux de la séquence t
$\text{del}(s[i:j:k])$	Suppression des éléments de $s[i:j:k]$

Modification à l'aide du "slicing"

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

```
maListe[2:4] = (6, 'x', 7) → [1, 2, 6, 'x', 7, 5]
```

```
maListe[1:6:2] = 'tes' → [1, 't', 6, 'e', 7, 's']
```



2.4.5

Les sets

Déclaration d'un set

```
monEnsembleVide = set()
```

```
monEnsembleAvecUnElement = {valeur}
```

```
monEnsemble = {valeur1, valeur2, valeur3}
```

Les sets

- Les **ensembles** sont des **collections non ordonnées** et **sans répétitions**.
- Les **ensembles** sont **modifiables**, en revanche il est possible d'utiliser les **frozenset** pour les rendre **non modifiables**.

Les opérations des sets

Opération	Résultat
<code>set(s)</code>	Transforme une séquence s en un ensemble.
<code>s.add(x)</code>	Ajoute l'élément x à l'ensemble s .
<code>s.update(t)</code>	Étend s avec la séquence t .
<code>s.discard(x)</code>	Supprime l'élément x à l'ensemble s .
<code>s.remove(x)</code>	Supprime l'élément x à l'ensemble s , en levant une exception si x n'est pas présent dans s .

Les opérations des sets (suite)

Opération	Résultat
<code>s1.union(s2)</code> ou <code>s1 s2</code>	Créer un set avec les éléments de s1 et s2 .
<code>s1.intersection(s2)</code> ou <code>s1 & s2</code>	Créer un set avec les éléments communs de s1 et s2 .
<code>s1.difference(s2)</code> ou <code>s1 - s2</code>	Créer un set avec les éléments de s1 non compris dans s2 .
<code>s1.symmetric_difference(s2)</code> ou <code>s1 ^ s2</code>	Créer un set avec les éléments de s1 et s2 , mais qui ne sont pas dans les deux à la fois.
<code>s1.issubset(s2)</code> ou <code>s1 <= s2</code>	Renvoie true si s1 est un sous-ensemble de s2 .
<code>s1.issuperset(s2)</code> ou <code>s1 >= s2</code>	Renvoie true si s2 est un sous-ensemble de s1 .

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark gray, while others are hollow with a light gray outline. The lines connecting them are thin and light gray, creating a dense, organic structure that tapers off towards the right.

2.4.6

Les dictionnaires



Déclaration d'un dictionnaire

```
monDictionnaireVide = {}
```

```
monDictionnaireVide = dict()
```

```
monDictionnaireAvecUnElement = {clé:valeur}
```

```
monDictionnaire = {clé1:valeur1, clé2:valeur2}
```

Les dictionnaires

- Les **dictionnaires** sont des **collections** d'objets **non-ordonnées**.
- Un **dictionnaire** est composé d'**éléments** et chaque **élément** se **compose** d'une **paire clé: valeur**.
- Les **dictionnaires** sont des **objets modifiables**.
- Un **dictionnaire** peut contenir des **objets** de **tous les types**, mais les **clés** doivent **être uniques**.

Les opérations des dictionnaires

Opération	Résultat
<code>dict(s)</code>	Transforme une séquence s de paire clé-valeur en un dictionnaire.
<code>d.get(k)</code>	Retourne la valeur v se trouvant à la clé k du dictionnaire d . Renvoie None si la clé k n'existe pas.
<code>d.pop(k)</code>	Supprime l'élément qui possède la clé k , tout en renvoyant sa valeur v .
<code>d.popitem()</code>	Supprime le dernier élément, tout en renvoyant un tuple contenant sa clé et sa valeur.
<code>d.clear()</code>	Vide le dictionnaire d .
<code>del d</code>	Supprime le dictionnaire d .



Avez-vous des questions ?



Exercices

Les collections

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by circles of varying sizes, some with solid centers and others with dashed outlines. The lines are thin and gray, creating a dense, organic structure that tapers off towards the right.

2.5

Les structures itératives

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of interconnected nodes and lines, with nodes represented by circles of different sizes and line styles (solid and dashed). The overall pattern is a complex, branching network that fades into the background.

Boucles bornées et non bornées

Boucle bornée

Quand on **sait combien** de **fois** doit avoir lieu la **répétition**, on utilise généralement une boucle **for**.

Boucle non bornée

Si on **ne connaît pas** à l'avance le nombre de **répétitions**, on choisit une boucle **while**.

La boucle **for**

```
for compteur in range:  
    instructions
```

Les **instructions** du **bloc for** seront **exécutées** autant de fois que la **range** le permet. On dit qu'on réalise une **itération**, à **chaque fois** que les **instructions** de la boucle sont **exécutées**.

Exemple :

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

Parcours d'une séquence avec un **for**.

```
for x in maSequence:  
    instructions
```

```
for i, x in enumerate(maSequence):  
    instructions
```

```
for i in range(len(maSequence)):  
    instructions
```

```
for x, y in zip(maSequence1, maSequence2):  
    instructions
```

Parcours d'un dictionnaire avec un **for**.

```
for key in monDictionnaire:  
    instructions
```

```
for key, value in monDictionnaire.items():  
    instructions
```

La boucle **while**

`while condition:`
`instructions`

Les **instructions** du **bloc while** seront **exécutées** **tant que** la **condition** est **vraie**.

ATTENTION à la boucle infinie !!!

Exemple :

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Break

L'instruction **break** permet de « **casser** » l'**exécution** d'une **boucle** (**while** ou **for**). Elle fait **sortir** de la **boucle** et passer à l'instruction suivante.

Continue

L'instruction **continue** permet de **passer prématurément** au **tour** de **boucle suivant**(**while** ou **for**). Elle fait **continuer** sur la **prochaine itération** de la boucle.

L'instruction **else** après une boucle

```
for compteur in range:  
    instructions  
else:  
    instructions
```

Les **instructions** du **bloc else** seront **exécutées** uniquement si la **boucle arrive à son terme** « **normalement** » (pas de **break**).

Exemple :

```
for i in range(5):  
    print(i)  
else:  
    print("end")
```

La syntaxe pour définir une liste en compréhension

Le **but** est de **construire** une **liste** à partir d'une **séquence** déjà **existant**.

```
[expression for x in maSequence if conditions]
```


La syntaxe pour définir une liste en compréhension

```
t = tuple(range(5))
```

```
maListe1 = [x ** 2 for x in t]
```

```
print(maListe1) → [0, 1, 4, 9, 16]
```

```
maListe2 = [x for x in t if x%2 == 0]
```

```
print(maListe2) → [0, 2, 4]
```



**Avez-vous des
questions ?**



Exercices

Les structures itératives



A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark grey, while others are hollow with a light grey outline. The lines connecting them are thin and light grey, creating a dense, organic structure that tapers off towards the right.

2.6

Les fonctions

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being solid dark grey and others being hollow with a light grey outline. The lines are thin and light grey, forming a complex, interconnected web that tapers off towards the left.



2.6.1

Les généralités

Le principe

Une **fonction** est un **bloc d'instructions réalisant une certaine tâche**.

Elle possède un **nom** et est **exécutée** lorsqu'on **l'appelle**.

Un **programme bien structuré** contiendra une **fonction** dite « **principale** », et **plusieurs fonctions** dédiées à des fonctionnalités spécifiques.

Quand une **fonction** dite « **principale** » fait appel à une **autre fonction**, elle **suspend** son **déroulement**, et **exécute l'autre fonction**, puis **reprend** ensuite son **fonctionnement**.

Les avantages

L'utilisation de **fonction** possède **3 avantages** :

- **Eviter** la **duplication** de code.
- **Favoriser** la **réutilisation**.
- **Améliorer** la **conception** (en réduisant la complexité).

Les paramètres

Une **fonction** sert donc à effectuer un **traitement générique**.

Ce **traitement** porte sur des **données**, dont la **valeur** pourra ainsi **changer** d'un **appel à l'autre** de la **fonction**.

Ces **données** sont **appelées paramètre**.

Lors de l'**implémentation** d'une **fonction**, on va donc préciser la **liste de tous les paramètres** qu'elle va utiliser.

Les paramètres

Lors de l'**implémentation** d'une **fonction**, on va donc préciser la **liste de tous les paramètres** qu'elle va utiliser.

Lors de l'**utilisation** d'une **fonction**, on va alors **préciser** la **valeur** de **chacun** des **paramètres** qu'elle possède.

Les paramètres par défaut

Les **paramètres** d'une **fonction** peuvent comporter des **valeurs** par **défaut**.

Lorsqu'on **appelle** une **fonction**, on a **deux cas possibles** :

- On **ne précise pas** de **valeurs** pour les **paramètres** et la **fonction** utilise celles par **défaut**.
- On **précise** des **valeurs** ce sont celles-ci qui sont **utilisées**.

ATTENTION !

Les **paramètres** par **défaut** sont **obligatoirement** positionnés à **droite** des **paramètres**.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark grey, while others are hollow with a light grey outline. The lines connecting them are thin and light grey, creating a dense, organic structure that tapers off towards the right.

2.6.2

La portée des variables



Les variables locales

Pour réaliser sa **tâche**, une **fonction** aura besoin de ses **propres variables**. On parle alors de « **variables locales** ».

Ces **variables** ne sont **accessibles** qu'**au sein** de la **fonction** qui les définit.

Les variables globales

Une **fonction** reçoit donc des **données** à traiter, les **paramètres**, et pour ce faire peut **avoir besoin** de **variables locales**.

Une **fonction** peut également **manipuler directement** des **variables** définies par le **programme principal**. On parle alors de « **variables globales** ».

ATTENTION !

Il s'agit souvent d'une **mauvaise pratique** car cela **limite** les **performances** et la **réutilisabilité** du code.



2.6.3

L'implémentation

Syntaxe pour déclarer une fonction

Il existe **deux types** de **fonction** : Celles qui **retournent** une **valeur** et celles qui ne **retournent rien**.

Exemples :

```
def maFonction(param1, param2 = 0):  
    instructions  
    return monResultat
```

```
def maFonction(param1, param2):  
    instructions
```

Syntaxe pour appeler une fonction

Comme une instruction prédéfinie du langage. On appelle la **fonction** par son **nom**, en lui **passant** autant de **paramètres** qu'elle en **possède**.

Exemples :

```
maFonction(42)
```

```
a, b = 5, 10
```

```
resultat = maFonction(a, b)
```


Une particularité de Python

Une **fonction** peut **retourner plusieurs valeurs**, il suffit de **séparer** celles-ci par des **virgules**.

Exemples :

```
def maFonction(param1, param2):  
    instructions  
    return monResultat1, monResultat2
```

```
a, b = 5, 10  
longueur, largeur = maFonction(a, b)
```



Avez-vous des questions ?



Exercices

Les fonctions



2.7

Les fichiers

La fonction **open**

```
open(chemin, mode, encoding="utf8")
```

La fonction **open** permet d'ouvrir un fichier. Celle-ci attend **deux arguments** : un **chemin d'accès** vers un fichier et un **mode** qui détermine le **type** (texte ou binaire) et la **nature** des opérations qui seront réalisées sur le fichier (lecture, écriture ou les deux). Elle **retourne** une variable qui contient le contenu du fichier.

Le mode d'ouverture

Le **mode** est une **chaîne de caractères** composés d'une ou plusieurs lettres qui **décrit** le **type** du **flux** et la **nature** des **opérations** qu'il doit **réaliser**.

Mode	Type(s) d'opération(s)	Effets
r	Lecture	Rien
r+	Lecture et écriture	
w	Ecriture	Si le fichier n'existe pas, il est créé. Si le fichier existe, son contenu est effacé.
w+	Lecture et écriture	
a	Ecriture	Si le fichier n'existe pas, il est créé. Si le fichier existe, on écrit à la suite.
a+	Lecture et écriture	

La fonction **close**

close()

La fonction **close** permet de fermer un fichier.

Exemple :

```
fichier = open(file, "r")  
instructions  
fichier.close()
```

Lire le contenu d'un fichier

`read(size)`

La fonction **read** permet de **lire** le contenu d'un **fichier**.

L'argument **size** permet de **définir** le **nombre** de **caractère** à **lire**, il est **facultatif**.

Par défaut, la fonction **read** lit l'**intégralité** du **fichier**.

`readline()`

La fonction **readline** permet de **lire** le contenu d'un **fichier ligne par ligne**.

Ecrire dans un fichier

`write(variable)`

La fonction **write** permet d'**écrire** le contenu d'une **variable** dans un **fichier**.

`writelines(list)`

La fonction **writelines** permet d'**écrire** le contenu d'une **liste** dans un **fichier**.

Le mot clé **with**

Exemples :

```
fichier = open(file, "a+")  
texte = fichier.read()  
fichier.write(texte)  
fichier.close()
```

```
with open(file, "a+") as file:  
    texte = file.read()  
    file.write(texte)
```



**Avez-vous des
questions ?**





Exercices

Les fichiers

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

3.

La P00

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being larger and having concentric circles, all rendered in a light gray color.


Définition

La **POO** repose sur le **concept** de **classe** qui sont des **entités** qui vont pouvoir **posséder** un ensemble d'**attributs** et de **méthodes** qui leur sont propres.



Objectif

La **P00** a pour **objectif** de **rendre** nos **scripts** plus **clairs**, **mieux structurés**, **plus modulable** et **plus facile à maintenir** et à **débugger**.



La notion de classe et d'objet

Une **classe** est un “**moule**” qui va nous **permettre** de **créer** des **objets**. Chaque **objet** aura les **attributs** de **sa classe**, mais nous pourrons les **personnaliser**.

Les **classes** sont la **base** de la **POO**, car elles permettent de mettre en place les **trois concepts fondamentaux** de cette dernière, à savoir :

- L'**encapsulation**
- L'**heritage**
- Le **polymorphisme**

Création d'une classe

```
class CompteBancaire:  
    id = 1  
    solde = 126
```



Création de la classe

```
    def setSolde(self, n):  
        self.solde = n
```

```
monCompte = CompteBancaire()
```



Création de l'objet /
Instanciation de la classe

L'opérateur .

L'**accès** à un **attribut** d'une **classe** se réalise à l'aide de l'**opérateur .** suivi du **nom** de l'**attribut**. Ce dernier s'utilise comme une variable classique.

Le fonctionnement est le même pour l'accès aux **méthodes**.

Exemple :

```
monCompte.setSolde(5000)  
monCompte.id ← 1
```

Les constructeurs

Il existe une méthode particulière qui permet « **d'initialiser** » nos **objets**. On appelle cette **méthode** un **constructeur** et elle se code `__init__()`.

La méthode `__init__()` va être **automatiquement exécutée** au moment de l'**instanciation** d'une **classe**. Cette fonction va pouvoir **recevoir** des **arguments** pour « **personnaliser** » nos **objets**.

Exemple :

```
class CompteBancaire:
    def __init__(self, id, prenom, solde):
        self.id = id
        self.prenom = prenom
        self.solde = solde

monCompte = CompteBancaire(125, "Florent", 550)
```

Les méthodes “magiques”

Les **méthodes** “**magiques**” sont les méthodes **prédéfinies** par **python**, à l'image de la méthode **`__init()`**. Elles sont appelées **automatiquement** par l'**interpréteur** et elles sont toujours **définies** avec **`__`**.

Voici un exemple de **méthode magiques** :

Méthode	Fonctionnement
<code>__str__()</code>	Définir la représentation de l' objet sous forme de string .
<code>__len__()</code>	Définir la longueur de l' objet .
<code>__getitem__()</code>	Accéder à un élément de l'objet à l'aide de l' opérateur <code>[]</code> .
<code>__add__()</code>	Ajouter deux objets ensemble.

Les attributs de classes

Les **attributs de classe** sont des **attributs liés** à la **classe** directement et non à l'objet. Ainsi, l'**attribut de classe** est **accessible** à partir de l'**instance de la classe**.

Exemple :

```
class CompteBancaire:
    numeroCompte = 0

    def __init__(self, prenom, solde):
        self.__class__.numeroCompte += 1
        self.numeroCompte = self.__class__.numeroCompte
        self.prenom = prenom
        self.solde = solde

    def __str__(self):
        return "Le compte " + str(self.numeroCompte) + " a un solde de " + str(self.solde)

monCompte, compteLaurent = CompteBancaire("Florent", 550), CompteBancaire("Laurent", 1550)
print(compteLaurent)
```



Le compte 2 à un solde de 1550

L'encapsulation

L'**encapsulation** décrit l'idée « d'**enfermer** » les **attributs** et les **méthodes** au **sein** d'une **classe**. Cela **limite l'accès aux données de la classe** en dehors de cette dernière, dans le but de les **protéger**.

La visibilité des données

La majorité des langages de programmation ont **3 types** de **visibilité** pour les **données de classe** : **private**, **protected** et **public**.

Attention, la notion de visibilité n'existe pas en Python.

Exemple :

```
class CompteBancaire:
    def __init__(self, solde):
        self.solde = solde

    def getSolde(self):
        return self.solde

    def setSolde(self, n):
        self.solde = n
```

L'héritage

En **POO**, « **hériter** » signifie « **avoir également accès à** ».

La notion d'**héritage** va être particulièrement intéressante lorsqu'on va l'**implémenter** entre **deux classes**. En **POO**, nous allons pouvoir créer des **classes « enfants »** à partir de **classes de base** ou « **classes parentes** ».

Exemple :

```
class CompteBancaire:
    numeroCompte = 0

    def __init__(self, solde):
        self.__class__.numeroCompte += 1
        self.solde = solde

class CompteEpargne(CompteBancaire):
    estEpargne = True
```


La surcharge des méthodes de classe

« **Surcharger** » une **méthode** signifie la **redéfinir** d'une façon différente. En Python, les **classes filles** vont pouvoir **surcharger** les **méthodes héritées** de leur **classe parent**.

Souvent lors de la **redéfinition** d'une **méthode**, nous souhaitons **utiliser** la **méthode de base**. Pour se faire, nous allons l'**appeler directement** avec la **syntaxe** suivante : **NomClasseDeBase.nomMethode()**.

Exemple de surcharge

```
class CompteBancaire:
    def __init__(self, solde):
        self.solde = solde

    def __str__(self):
        return "Le compte " + str(self.numeroCompte) + " a un solde de " + str(self.solde)

class CompteEpargne(CompteBancaire):
    estEpargne = True

    def __init__(self, solde, taux):
        CompteBancaire.__init__(self, solde)
        self.taux = taux

    def __str__(self):
        return "Le compte d'épargne " + str(self.numeroCompte) + " a un solde de " + str(self.solde)
```

Le polymorphisme

« **Polymorphisme** » signifie littéralement « **plusieurs formes** ». En **POO**, le **polymorphisme** est un concept qui fait référence à la **capacité** d'un **attribut**, d'une **méthode** ou d'un **objet** à prendre plusieurs formes. Autrement dit, à sa **capacité** de **posséder plusieurs définitions** différentes.

Exemple de polymorphisme

```
class CompteBancaire:
    def __init__(self, solde):
        self.solde = solde
    def connaitrePlafond(self):
        pass

class CompteEnfant(CompteBancaire):
    def __init__(self, solde):
        CompteBancaire.__init__(self, solde)
        self.plafond = 50
    def connaitrePlafond(self):
        print("Vous êtes limité à", self.plafond, "€.")

class CompteEtudiant(CompteBancaire):
    def __init__(self, solde):
        CompteBancaire.__init__(self, solde)
        self.plafond = 500
    def connaitrePlafond(self):
        print("Vous êtes limité à", self.plafond, "€.")
```

Le duck typing

Il est possible d'**appliquer** le **polymorphisme** à des **types de base** en Python.
On nomme cela le **duck typing**.

Exemple :

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

```
resultat1 = ajouter(5, 10)           ← 15  
resultat2 = ajouter("Hello", " World") ← Hello World  
resultat3 = ajouter([0, 2, 4], [1, 3, 5]) ← [0, 2, 4, 1, 3, 5]
```



Avez-vous des questions ?



Exercices

La POO

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

4.

Debugging et Tests

A decorative network diagram in the bottom-right corner, similar to the one in the top-left, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

4.1

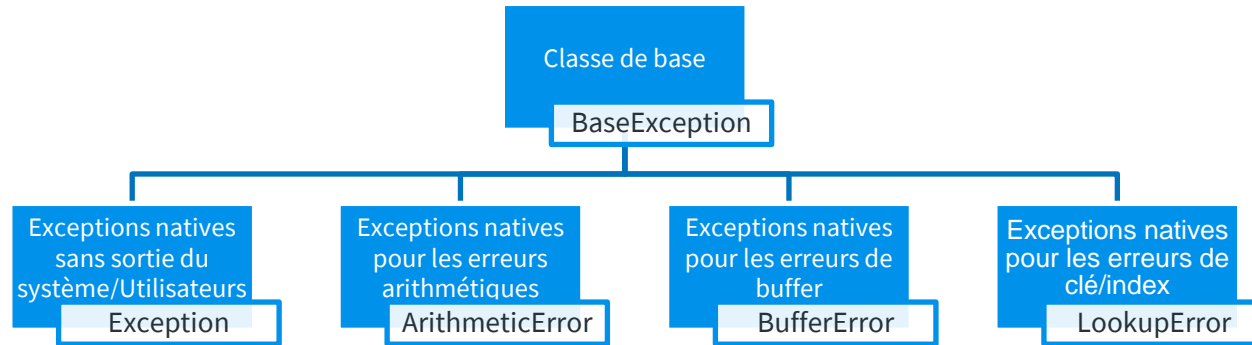
La gestion des erreurs

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of nodes connected by lines, with some nodes being larger and more prominent than others. The overall style is minimalist and technical.

Les classes exception

Il est important que votre **programme** soit en mesure de **gérer** les **erreurs d'environnement**.

Python gère les **erreurs** à l'aide de la **classe exception**, selon la **hiérarchie** suivante :



L'instruction `try ... except`

Les **clauses** `try` et `except` fonctionnent **ensemble**. Elles permettent de **tester** (`try`) un **code** qui peut **potentiellement poser problème** et de **définir** les **actions à prendre** si une **exception** est effectivement **rencontrée** (`except`).

Exemple :

```
try:
    retrait = int(input("Entrez un montant de retrait."))
    assert retrait > 0
except ValueError:
    print("Erreur: la valeur saisie n'est pas un entier")
except AssertionError:
    print("Erreur: vous ne pouvez pas rentrer une valeur négative")
```

La clause else

La **clause else** est **positionnée après** l'instruction **try... except**. Le **code contenu** dans cette **clause else** sera **exécuté** dans le **cas où aucune exception** n'a été **levée** par la **clause try**.

Exemple :

```
try:
    retrait = int(input("Entrez un montant de retrait."))
    assert retrait > 0
except ValueError:
    print("Erreur: la valeur saisie n'est pas un entier")
except AssertionError:
    print("Erreur: vous ne pouvez pas rentrer une valeur négative")
else:
    print("Vous avez effectué un retrait de", retrait, "€.")
```

La clause **finally**

La **clause **finally**** est **positionnée après** la clause **else**. Le **code contenu** dans cette **clause **finally**** sera **exécuté** dans **tous les cas**.

Exemple :

```
try:
    retrait = int(input("Entrez un montant de retrait."))
    assert retrait > 0
except ValueError:
    print("Erreur: la valeur saisie n'est pas un entier")
except AssertionError:
    print("Erreur: vous ne pouvez pas rentrer une valeur négative")
else:
    print("Vous avez effectué un retrait de", retrait, "€.")
finally:
    print("Vous êtes déconnecté de l'ATM.")
```

Le mot clé raise

La **mot clé** **raise** permet de **lever** une **exception**. Vous pouvez **définir** le **type d'erreur** à **soulever** et le **texte** à **afficher** pour l'utilisateur.

Exemple :

```
x = -1
if x < 0:
    raise Exception("Désolé, Pas de nombre inférieur à zéro.")

x = "hello"
if not type(x) is int:
    raise TypeError("Seulement les entiers sont acceptés.")
```



Avez-vous des questions ?



Exercices

Les Exceptions

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark grey, while others are hollow with a light grey outline. The lines connecting them are thin and light grey, creating a dense, organic structure.

4.2

Le debugging

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It consists of a cluster of nodes and connecting lines. The nodes are small circles, some solid dark grey and some hollow with light grey outlines. The lines are thin and light grey, forming a complex, interconnected pattern.

```
streamlitHelper.py > {} alt
1 import altair as alt
2 import polars as pl
3 import streamlit as st
4
5 @st.cache_data(show_spinner=False, experimental_allow_widgets=True)
6 def loadDataframe():
7     return pl.read_csv(source='Data/valeursFoncieres.csv')
8
9 def drawTop15(top15Years, default = True, new = True):
10     st.write("")
11     st.write("Evolution of the price per m² of the 15 largest cities in France")
12
13     # Rechargement de la page sans aucun changement
14     if new:
15         # Rechargement de la page avec les settings par défaut
16         if default:
17             st.session_state['top15_df'] = pl.read_csv(source='./Data/top15m')
18             # Chargement de la page avec un changement, avec des settings différents
19         else:
20             st.session_state['top15_df'] = pl.read_csv(source='./Data/top15m')
```

Démonstration

Les points d'arrêt.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are solid dark grey, while others are hollow with a light grey outline. The lines connecting them are thin and light grey, creating a mesh-like structure that extends from the top-left towards the center of the slide.

4.3

Les Tests en Python

A decorative network diagram in the bottom-right corner, mirroring the style of the top-left diagram. It consists of a cluster of interconnected nodes and lines. The nodes are small circles, some solid dark grey and some hollow with light grey outlines, connected by thin, light grey lines. This diagram is positioned in the bottom-right corner, extending from the right edge towards the center.

Les objectifs

- **Assurer** la **qualité** du logiciel.
- **Mesurer** les **régressions** à chaque mise à jour.
- **Refactoriser** en toute **confiance**.
- **Documenter** le **code** avec des **exemples d'utilisation**.

Test-Driven Development (TDD)

Le **TDD** est une **approche de développement** basé sur le **cycle** suivant :

1. **Créer** un **test unitaire** qui vise à s'assurer que les **exigences fonctionnelles** sont **satisfaites**.
2. **Vérifier** que le **test échoue**.
3. **Créer** le **code minimal** qui **résout le test**.
4. **Vérifier** que **tous** les **tests** sont **réussis**.

Les différents types de Test

Les Tests fonctionnels	
Unitaire	Vérification d'une petite unité de code (fonction ou classe).
Intégration	La combinaison d'unités de code de taille moyenne .
Régression	S'assurer que le niveau de qualité ne diminue pas .
Fumée	Vérifier les fonctionnalités critiques (et de base).
Les Tests non fonctionnels	
Stress	Mesurer la résistance du système sous une charge élevée .
Reprise	Mesurer la capacité de récupération du système .
Sécurité	Vérifier les vulnérabilités .

Des techniques de Test

Mocking	Remplacer les modules ou les fonctions par des mocks .
Génération	Fournir de nombreuses entrées aléatoires à un test pour détecter les bugs .
Mutation	Muter le programme , et la suite de tests doit échouer après la mutation .

Des techniques de Test - Mocking

- **Remplacer** une **classe**, une **méthode** ou un **attribut de module** par un **faux objet**.
- **Permet** de **tester** des **systèmes** qui ne sont **pas accessibles** pendant les **tests**.
- **Implémenter** par **unittest.mock** ou **pytest.monkeypatching**.

Des techniques de Test - Génération

- **Souvent couplé** à des **tests basés sur les propriétés**.
- Les **cas de Test** sont **automatisé**.
- **Réduit** les **cas d'erreur** à des **cas simples**.

Des techniques de Test - Mutation

- **Effectue** des **mutations** subtiles.
- **Compte** les **tests échoués** et **inachevés**.

Les bonnes pratiques

- **Suivez** le **cycle** suivant : **mise en place**, **exécution**, **validation**, **nettoyage**.
- **Créer** des **tests indépendants**.
- **Testez uniquement** l'API publique, **jamais** les **éléments** de l'implémentation.
- **Garder** les **tests rapides**.
- **Utiliser** des **tests de mutation** ou le **fuzzing**.

Les librairies Python

<u>unittest</u>	Le framework de test inclus dans la librairie standard.
<u>pytest</u>	Le framework de test le plus populaire.
<u>hypothesis</u>	Librairie pour la création de test unitaire.
<u>mutmut</u>	Librairie pour la création de test de mutation.
<u>deal</u>	Permet de réaliser du Design by contract (DbC).



Avez-vous des questions ?



Exercices

Les tests



5.

Analyse de données avec Numpy et Pandas





5.1

Introduction



Home

Environments

Learning

Community

Anaconda Notebooks

Cloud notebooks with hundreds of packages ready to code.

Learn More

A Full Python IDE directly from the

Documentation

Anaconda Blog



All applications

on

Streamlit

Channels



DataSpell

DataSpell is an IDE for exploratory data analysis and prototyping machine learning models. It combines the interactivity of Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendly environment.

Install



CMD.exe Prompt

0.1.1

Run a cmd.exe terminal with your current environment from Navigator activated

Launch



Jupyter

6.5.4

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Launch



VS Code

1.83.1

Streamlined code editor with support for development operations like debugging, task running and version control.

Launch



Datalore

Kick-start your data science projects in seconds in a pre-configured environment. Enjoy coding assistance for Python, SQL, and R in Jupyter notebooks and benefit from no-code automations. Use Datalore online for free.

Launch



Deepnote

Deepnote is a notebook built for collaboration. Create notebooks in your browser, spin up your conda environment in seconds and share with a link.

Launch



IBM Watson Studio Cloud

IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

Launch

ORACLE
Cloud Infrastructure

Oracle Data Science Service

OCI Data Science offers a machine learning platform to build, train, manage, and deploy your machine learning models on the cloud with your favorite open-source tools

Launch



PyCharm Professional

A full-fledged IDE by JetBrains for both Scientific and Web Python development. Supports HTML, JS, and SQL.

Install

Anaconda

Un environnement
pour la Data Science

L'écosystème de la Data Science

Les **librairies “standard”** pour la Data Science sont :

- **numpy** : Les **tableaux multidimensionnels** et les **calculs scientifiques**.
- **pandas** : La **manipulation** des **données**.
- **matplotlib** : L'**affichage** des **données**.
- **scipy** : Des **structures avancées** et diverses **fonctionnalités scientifiques**.



5.2

numpy



Introduction

numpy est une **bibliothèque centrée** sur un **concept** : le **tableau multidimensionnel**.

La **manipulation** des **tableaux numpy** est très **rapide**. On peut observer une **amélioration** entre **100** et **500 fois plus rapide** par rapport aux **listes Python**.

La **façon standard** d'importer **numpy** est de lui associer l'**alias np**.

Exemple :

```
import numpy as np
```

Création des tableaux

```
import numpy as np

np.empty((2, 3))      # Crée un tableau rempli de valeur aléatoire.
np.zeros((2, 3))      # Crée un tableau rempli de 0.
np.ones((2, 3))       # Crée un tableau rempli de 1.

a = np.empty((2, 3))
a.fill(8)
a                     # Un tableau rempli de 8.

np.array([[1, 2, 3], [4, 5, 6]]) # Convertis les séquences en tableau.
```

Création des tableaux

```
import numpy as np

# Crée un tableau rempli avec les valeurs de 0 à 9.
np.arange(10)

# Crée un tableau de 5 valeurs allant de -1 à 1 au pas de 0,5.
np.linspace(-1, 1, num=5)

# Crée un tableau de 9 valeurs avec les puissances de 2.
np.geomspace(1, 256, num=9)

# Crée un tableau de avec les valeurs suivantes 12, 22, 32, 42.
np.logspace(1, 4, num=4 , base=2)
```

Informations sur les tableaux

```
import numpy as np

a = np.zeros((2, 5))

# Renvoie le nombre de dimension.
a.ndim ← 2

# Renvoie la taille des dimensions.
a.shape ← (2, 5)

# Renvoie le nombre d'élément du tableau.
a.size ← 18

# Renvoie le type des éléments du tableau.
a.dtype ← dtype('float64')
```

Opérations sur les tableaux

```
import numpy as np
```

```
a, b = np.array([1, 2]), np.array([3, 4])
```

```
a + 2, a - 2, a * 2, a ** 2   ← (array([3, 4]), array([-1, 0]), array([2, 4]), array([1, 4]))
```

```
a + b, a - b, a * b, a @ b, a ** b   ← (array([4, 6]), array([-2, -2]), array([3, 8]), 11, array([ 1, 16]))
```

```
a.min(), a.max(), a.std(), a.mean(), a.sum()   ← (1, 2, 0.5, 1.5, 3)
```


Les opérations et la mémoire

Deux types de **gestion de la mémoire** peuvent être choisis pour une **opération** :

- elle est **stockée** dans un **nouveau tableau** (utilisation d'une nouvelle mémoire).
- elle vient **remplacer** les **valeurs précédentes d'un tableau** (opération sur place).

Dans les **fonctions numpy**, l'**argument out** permet souvent de **choisir** entre le **stockage sur place** et la **création d'un tableau**.

Il est également possible d'utiliser les **opérations de Python** tel que **+=**.

Les opérations d'axe sur les tableaux

```
a = np.array([[1, 2, 3], [4, 5, 6]])
```

```
a.mean(axis=0) ← array([2.5, 3.5, 4.5])
```

```
a.mean(axis=1) ← array([2., 5.])
```

```
a.transpose() ← array([[1, 4], [2, 5], [3, 6]])
```

```
a.swapaxes(0, 1) ← array([[1, 4], [2, 5], [3, 6]])
```

```
a.reshape(1, 6) ← array([[1, 2, 3, 4, 5, 6]])
```

```
a[-1, 1] ← 5
```

Le slicing

```
a = np.array([1, 2, 3, 4, 5, 6])
```

```
a[0:4] ← array([1, 2, 3, 4])
```

```
a[:4] ← array([1, 2, 3, 4])
```

```
a[:4:2] ← array([1, 3])
```

```
a[::-1] ← array([6, 5, 4, 3, 2, 1])
```

```
a = a.reshape(2, 3)
```

```
a[:, :2] ← array([[1, 2], [4, 5]])
```

Les tableaux d'indice et les masques

```
a = np.arange(4) * 10  ← array([0, 10, 20, 30])  
indice = np.array([0, 1, 3])  
a[indice]              ← array([0, 10, 30])
```

#On peut créer facilement des tableaux de booléens.

```
b = np.arange(4)  
mask = b >= 2  ← array([False, False, True, True])
```

```
a[mask]  ← array([20, 30])
```

```
a[a < 30] += 10  ← array([10, 20, 30, 30])
```

Sauvegarder et charger des tableaux

```
a, b = np.arange(4), np.arange(10)
```

```
np.savez_compressed("tableaux.npz", nom_a = a, nom_b = b)
```

```
tableaux = np.load("tableaux.npz")
```

```
tableaux["nom_b"] ← array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```



**Avez-vous des
questions ?**



Exercices

Numpy



5.3

pandas



Introduction

pandas permet de **manipuler facilement** des **données**, car il est possible de :

- **Charger** de **nombreux formats de données** dans une **structure facile à manipuler**.
- **Filtrer, grouper, séparer, réarranger, combiner** des **données**.
- **Résumer, agréger, observer**.
- **Gérer** les **valeurs manquantes**.

La **façon standard** d'importer **pandas** est de lui associer l'**alias pd**.

Exemple :

```
import pandas as pd
```

Les structures de données

La **bibliothèque** s'**articule** autour de **deux structures de données** :

- **Series** : Une **structure** d'**une dimension** qui **représente plusieurs échantillons** de la **même variable**.
- **Dataframe** : Une **structure** de **deux dimensions** qui est **composé** d'une **série par colonne**.

Exemple :

	prix_m ²	surface	nombrePieces
count	212354.000000	212354.000000	212354.000000
mean	5747.244005	61.975492	2.862324
std	4662.019548	24.854384	0.859005
min	200.000000	7.090000	2.000000
25%	2550.655000	44.060000	2.000000
50%	4039.705000	59.130000	3.000000
75%	8928.570000	74.950000	3.000000
max	209232.230000	209.630000	5.000000

Dataframe

Series

Le concept d'index

Un **DataFrame** possède **deux indexes** :

- **df.index** qui **correspond** aux **lignes**.
- **df.columns** qui **correspond** aux **colonnes**.

Exemple :

```
import pandas as pd
```

```
df = pd.DataFrame([(23, 30, 32, 25), (20, 29, 34, 27)],  
                  columns=["Juin", "Jui.", "Aout", "Sept."],  
                  index=["2021", "2022"])
```

	Juin	Jui.	Aout	Sept.
2021	23	30	32	25
2022	20	29	34	27

```
df.index      ==> Index(['2021', '2022'], dtype='object')  
df.columns    ==> Index(['Juin', 'Jui.', 'Aout', 'Sept.'], dtype='object')
```

Importer des données

pandas peut **importer** des **données** de **nombreux formats** avec les méthodes **read_*** : **read_csv**, **read_excel**, **read_json**, **read_xml**, **read_parquet**, ...

Exemple :

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')  
df
```

	Date	Description	Deposits	Withdrawls	Balance
0	20-Aug-2020	NEFT	23,237.00	00.00	37,243.31
1	20-Aug-2020	NEFT	00.00	3,724.33	33,518.98
2	20-Aug-2020	Commission	245.00	00.00	33,763.98
3	20-Aug-2020	NEFT	12,480.00	00.00	46,243.98
4	20-Aug-2020	RTGS	00.00	11,561.00	34,682.98
...

Afficher vos données

Pour afficher nos données, nous pouvons utiliser la fonction **print()**, mais nous préférons souvent les **méthodes** **head()** ou **tail()**.

Exemple :

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')
```

```
df.head(3)
```



	Date	Description	Deposits	Withdrawls	Balance
0	20-Aug-2020	NEFT	23,237.00	00.00	37,243.31
1	20-Aug-2020	NEFT	00.00	3,724.33	33,518.98
2	20-Aug-2020	Commission	245.00	00.00	33,763.98

```
df.tail(6)
```



	Date	Description	Deposits	Withdrawls	Balance
94	21-Aug-2020	Purchase	00.00	51,020.50	51,020.50
95	21-Aug-2020	Debit Card	02.78	00.00	51,023.28
96	21-Aug-2020	Commission	300,660.00	00.00	351,683.28
97	21-Aug-2020	IMPS	00.00	87,920.82	263,762.46
98	21-Aug-2020	ATM	00.00	26,376.25	237,386.21
99	21-Aug-2020	Transfer	193,326.38	00.00	430,712.59

Obtenir des informations sur les DataFrames

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')  
df.shape  
(100, 5)
```

```
df.dtypes
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 100 entries, 0 to 99
```

```
Data columns (total 5 columns):
```

```
# Column Non-Null Count Dtype
```

```
-----
```

```
0 Date 100 non-null object
```

```
1 Description 100 non-null object
```

```
dtypes: object(5)
```

```
memory usage: 4.0+ KB
```

Date object
Description object
Deposits object
Withdrawals object
Balance object dtype: object

Obtenir des informations numériques sur les DataFrames

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')  
df.describe()
```

	Date	Description	Deposits	Withdrawls	Balance
count	100	100	100	100	100
unique	2	15	59	41	95
top	21-Aug-2020	Cheque	00.00	00.00	00.00
freq	70	10	42	60	4

Obtenir des informations sur les valeurs

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')
```

```
# Obtient l'ensemble des valeurs d'une colonne.
```

```
df.Description.unique()
```



```
array(['NEFT', 'Commission', 'RTGS',  
      'Miscellaneous', 'Cheque', 'Purchase', 'Cash',  
      'IMPS', 'Transfer', 'ATM', 'Interest', 'Reversal',  
      'Bill', 'Debit Card', 'Tax'], dtype=object)
```

```
df.Date.value_counts()
```

```
21-Aug-2020    70
```

```
20-Aug-2020    30
```

```
Name: Date, dtype: int64
```

Les operations avec Pandas

```
import pandas as pd
```


```
df = pd.read_csv("Data/records.csv", sep=',', thousands=',')
```

```
df["Balance"], df["Deposits"] = df["Balance"].astype(float), df["Deposits"].astype(float)
```

```
mask = df.Deposits > 0  
dfWithChange = df[mask]
```

```
df["OldBalance"] = df["Balance"] - df["Deposits"]
```

```
dfNumeric = df.select_dtypes(include=["number"])
```



	Deposits	Withdrawals	Balance	OldBalance
0	23237.00	0.00	37243.31	14006.31
1	0.00	3724.33	33518.98	33518.98
2	245.00	0.00	33763.98	33518.98
3	12480.00	0.00	46243.98	33763.98
4	0.00	11561.00	34682.98	34682.98
...

	Date	Description	Deposits	Withdrawals	Balance	OldBalance
0	20-Aug-2020	NEFT	23237.00	0.00	37243.31	14006.31
1	20-Aug-2020	NEFT	0.00	3724.33	33518.98	33518.98
2	20-Aug-2020	Commission	245.00	0.00	33763.98	33518.98
3	20-Aug-2020	NEFT	12480.00	0.00	46243.98	33763.98
4	20-Aug-2020	RTGS	0.00	11561.00	34682.98	34682.98
...

L'indexing

Il existe **deux méthodes** d'**indexing** :


- **loc** qui est basé sur la **valeur**.
- **iloc** qui est basé sur la **position**.

Exemple :


```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')  
df.loc[df["Date"] == "20-Aug-2020"]
```

```
df.iloc[-6:]
```



	Date	Description	Deposits	Withdrawls	Balance
94	21-Aug-2020	Purchase	0.00	51020.50	51020.50
95	21-Aug-2020	Debit Card	2.78	0.00	51023.28
96	21-Aug-2020	Commission	300660.00	0.00	351683.28
97	21-Aug-2020	IMPS	0.00	87920.82	263762.46
98	21-Aug-2020	ATM	0.00	26376.25	237386.21
99	21-Aug-2020	Transfer	193326.38	0.00	430712.59



	Date	Description	Deposits	Withdrawls	Balance	OldBalance
0	20-Aug-2020	NEFT	23237.00	0.00	37243.31	14006.31
1	20-Aug-2020	NEFT	0.00	3724.33	33518.98	33518.98
2	20-Aug-2020	Commission	245.00	0.00	33763.98	33518.98
3	20-Aug-2020	NEFT	12480.00	0.00	46243.98	33763.98
4	20-Aug-2020	RTGS	0.00	11561.00	34682.98	34682.98
5	20-Aug-2020	Miscellaneous	88736.00	0.00	123418.98	34682.98
6	20-Aug-2020	Cheque	0.00	15427.37	107991.61	107991.61

La gestion des valeurs manquantes

```
mask = df.Deposits > 0  
df["OldBalance"] = df[mask]["Balance"] - df["Deposits"]
```

	Date	Description	Deposits	Withdrawls	Balance	OldBalance
0	20-Aug-2020	NEFT	23237.00	0.00	37243.31	14006.31
1	20-Aug-2020	NEFT	0.00	3724.33	33518.98	NaN
2	20-Aug-2020	Commission	245.00	0.00	33763.98	33518.98
3	20-Aug-2020	NEFT	12480.00	0.00	46243.98	33763.98
4	20-Aug-2020	RTGS	0.00	11561.00	34682.98	NaN
...

```
df.isna()
```

	Date	Description	Deposits	Withdrawls	Balance	OldBalance
0	False	False	False	False	False	False
1	False	False	False	False	False	True
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	True
...

```
df.isna().sum().sum() #Retoune le nombre total de valeur NA/NaN, ici 42.
```

```
df.fillna(0, inplace=True) #Remplace les valeurs NA/NaN par 0 dans ce cas.
```

Le grouping

Il est possible de **travailler** sur des **sous-ensembles** de **données** avec la **fonction df.groupby** :

Exemple :

```
import pandas as pd
```

```
df = pd.read_csv("Data/records.csv", sep=',')  
df.groupby(df.Description == "ATM").mean()
```

	Deposits	Withdrawls	Balance
Description			
False	92203.256489	86637.223511	769363.237021
True	81461.526667	99211.673333	347340.095000

L'interopabilité avec numpy

Il est possible de **passer** des **DataFrame** à des **fonctions numpy** qui vont s'**appliquer** à **tous** les **éléments**.

Un **tableau numpy** peut être **extraît** d'une **série** ou d'un **DataFrame** avec l'**attribut values**.

Exemple :

```
np.around(df.Balance)
```

```
0    37243.0  
1    33519.0  
2    33764.0  
3    46244.0  
4    34683.0  
...
```

`df.Date.values` #Retourne un tableau numpy avec toutes les dates.



**Avez-vous des
questions ?**





Exercices

Pandas



5.4

matplotlib



Introduction

[matplotlib](#) est la **bibliothèque graphique** de **référence** pour la **visualisation 2D**.

Un **grand nombre de bibliothèques** gravitent autour de **matplotlib** pour répondre à différents besoins et domaines.

Exemple :

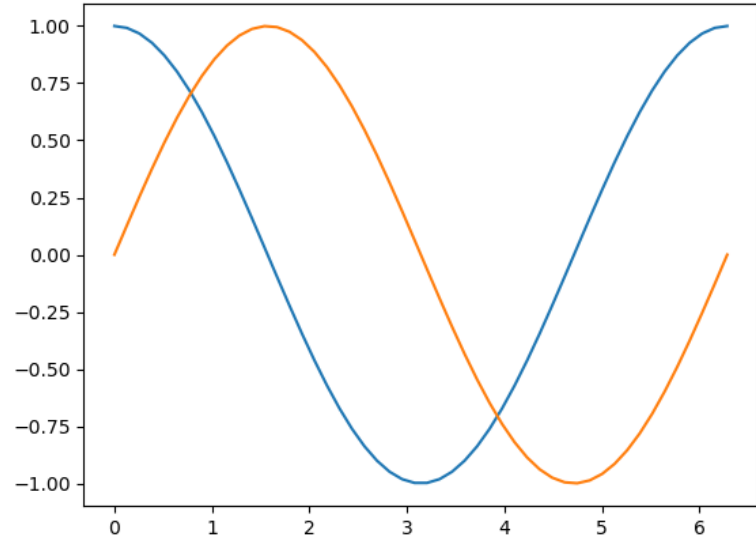
```
import matplotlib.pyplot as plt
```

Le premier graphe avec plot(x, y)

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
x = np.linspace(0, np.pi * 2)  
y1 = np.cos(x)  
y2 = np.sin(x)
```

```
plt.plot(x, y1)  
plt.plot(x, y2)  
plt.show()
```



scatter(x, y)

```
import matplotlib.pyplot as plt  
import numpy as np
```

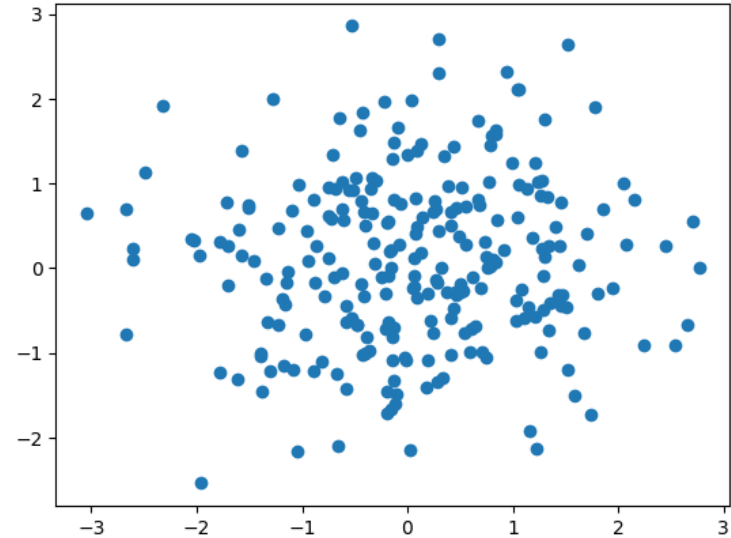
```
rng = np.random.default_rng()
```

```
x = rng.normal(size=250)
```

```
y = rng.normal(size=250)
```

```
plt.scatter(x, y)
```

```
plt.show()
```

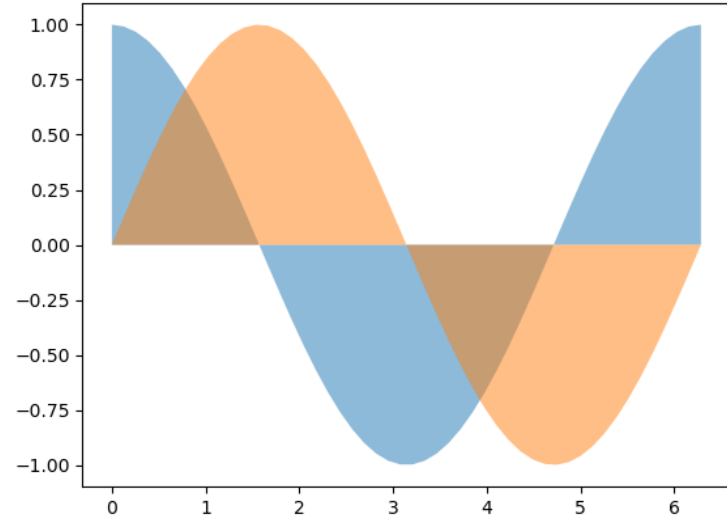


fill_between(x, y, y1)

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
x = np.linspace(0, np.pi * 2)  
y1 = np.cos(x)  
y2 = np.sin(x)
```

```
plt.fill_between(x, y1, 0, alpha=0.5)  
plt.fill_between(x, y2, 0, alpha=0.5)  
plt.show()
```



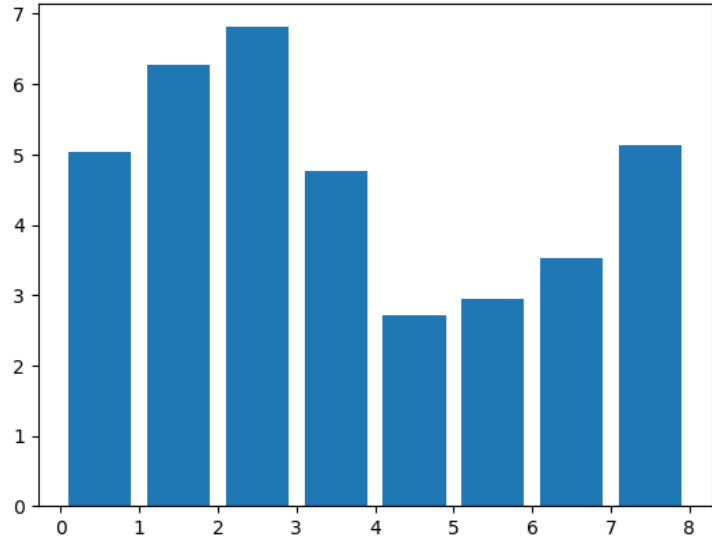
bar(x, y)

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
rng = np.random.default_rng()
```

```
x = 0.5 + np.arange(8)  
y = rng.uniform(2, 7, len(x))
```

```
plt.bar(x, y)  
plt.show()
```

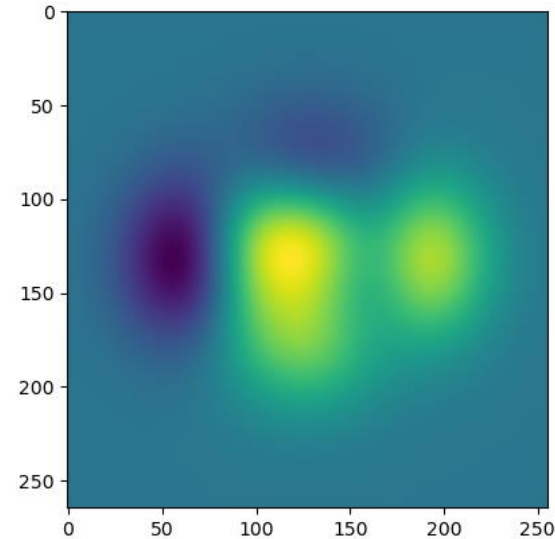


imshow(Z)

```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.meshgrid(
    np.linspace(-3, 3, 256),
    np.linspace(-3, 3, 256))
Z = (1 - X / 2 + X ** 5 + Y ** 3)
    * np.exp(-(X ** 2) - Y ** 2)

plt.imshow(z)
plt.show()
```

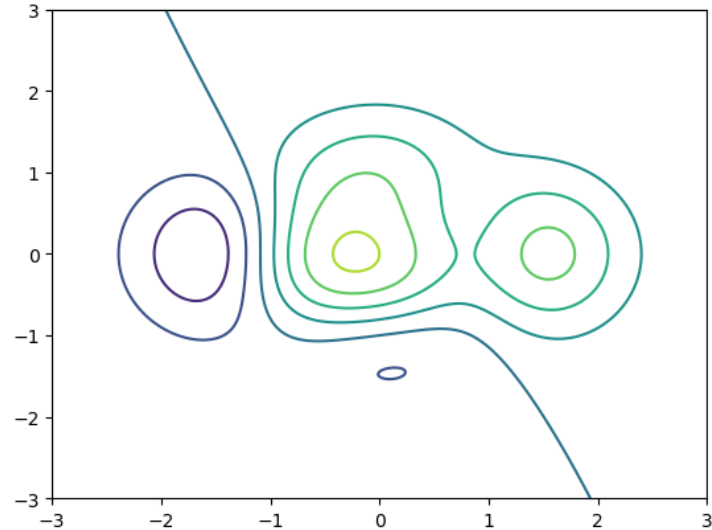


contour(X, Y, Z)

```
import matplotlib.pyplot as plt
import numpy as np

X, Y = np.meshgrid(
    np.linspace(-3, 3, 256),
    np.linspace(-3, 3, 256))
Z = (1 - X / 2 + X ** 5 + Y ** 3)
    * np.exp(-(X ** 2) - Y ** 2)

plt.contour(X, Y, Z)
plt.show()
```



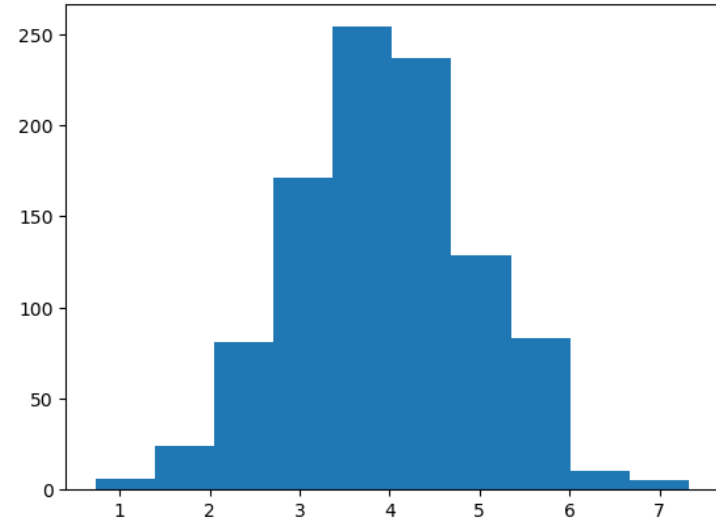
hist(x)

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
rng = np.random.default_rng()
```

```
x = rng.normal(4, size=1000)
```

```
plt.hist(x, bins=10)  
plt.show()
```

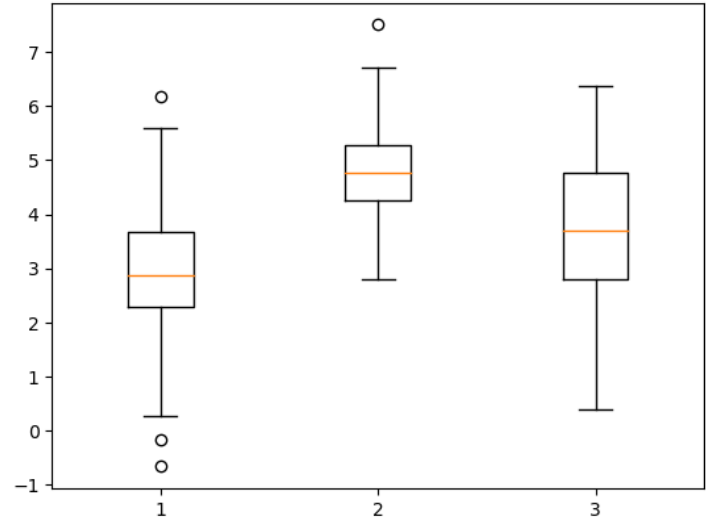


boxplot(X)

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
rng = np.random.default_rng(10)  
D = rng.normal(  
    (3, 5, 4),  
    (1.25, 1.00, 1.25),  
    (100, 3)  
)
```

```
plt.boxplot(D)  
plt.show()
```



Matplotlib une simple librairie ?

Ces premiers exemples en quelques lignes suggèrent une bibliothèque facile d'accès et d'utilisation pour la création de graphique.

Ce n'est pas si simple !

Une librairie riche mais complexe

Avantage

Matplotlib est une **bibliothèque hautement configurable**. C'est ce qui permet à de nombreuses **autres bibliothèques** de s'appuyer sur elle.

Inconvenient

Mais cela **implique** nécessairement un **coût** pour **réussir** à faire des **schémas esthétiques**.

Et malheureusement, la **documentation** est **loin d'être parfaite**.

L'anatomie d'un graphique

Figure	La structure qui contient tous les éléments à afficher.
Axes	Élément qui contient un graphique. Une figure contient un ou plusieurs axes
Spine	Encadrement d'un axe.
Legend	Légende associée à un axe.
Title	Titre associé à un axe.
Suptitle	Titre associé à un graphique.
Label	Description de l'axe.
Tick	Marqueurs sur l'encadrement d'un axe pour représenter l'échelle des valeurs.
Grid	Grille affichée sur la visualisation.

Paramétrisation

Un grand nombre d'**éléments** peuvent être **modifiés** pour **changer** les **graphiques**.

Taille et résolution

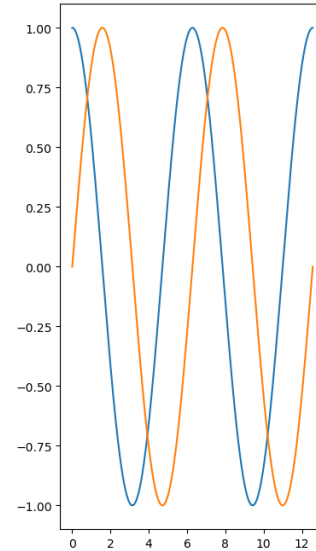
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots(
    figsize=(4, 8),
    dpi=100)

plt.plot(x, y1)
plt.plot(x, y2)

plt.show()
```



Les titres et légendes

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

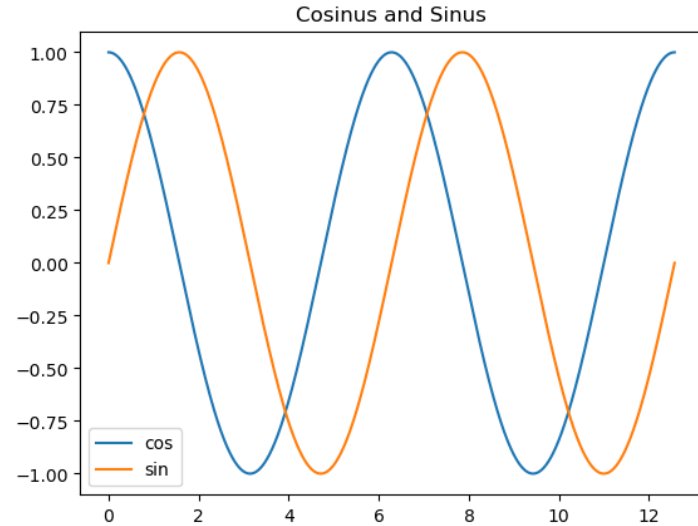
fig, ax = plt.subplots()

ax.plot(x, y1, label="cos")
ax.plot(x, y2, label="sin")

ax.legend()

ax.set_title("Cosinus and Sinus")

fig.show()
```



Les limites

```
import matplotlib.pyplot as plt
import numpy as np

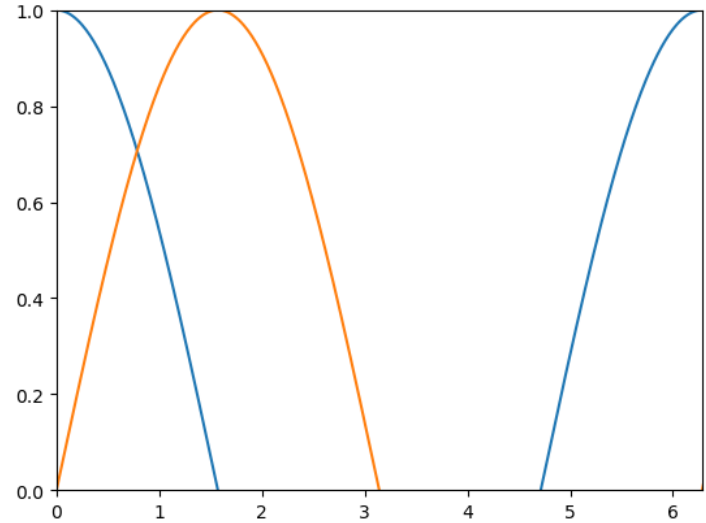
x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots()

ax.set_ylim(0, 1)
ax.set_xlim(0, np.pi * 2)

ax.plot(x, y1)
ax.plot(x, y2)

fig.show()
```



Modifier l'encadrement

```
import matplotlib.pyplot as plt
import numpy as np

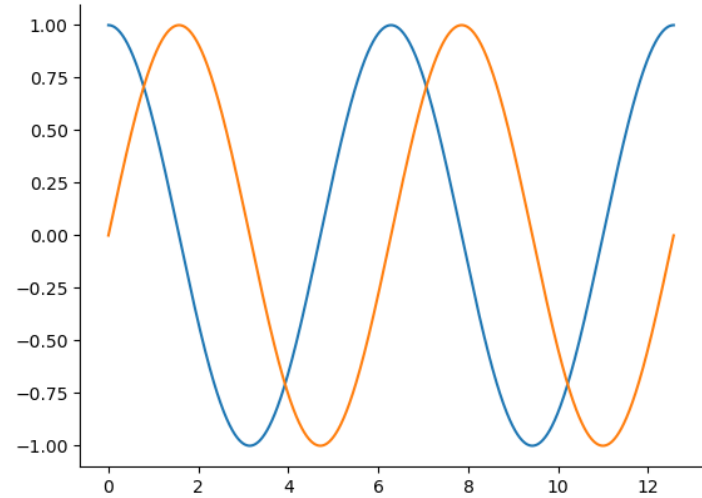
x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots()

ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)

ax.plot(x, y1)
ax.plot(x, y2)

fig.show()
```



Ajouter des labels

```
import matplotlib.pyplot as plt
import numpy as np

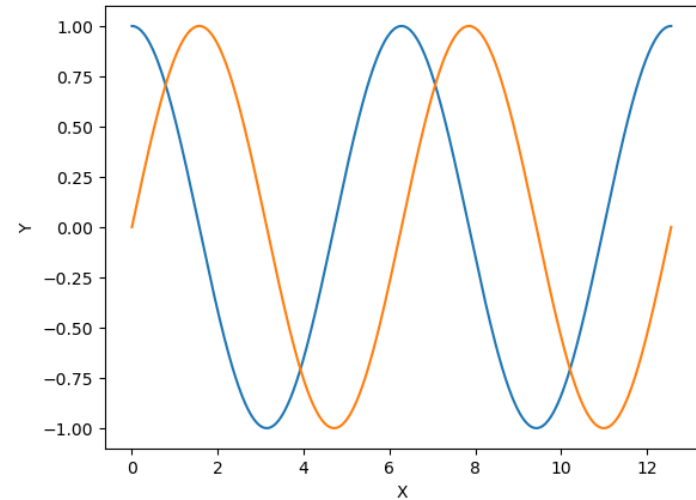
x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots()

ax.set_xlabel("X")
ax.set_ylabel("Y")

ax.plot(x, y1)
ax.plot(x, y2)

fig.show()
```



Changer les marqueurs d'échelle

```
import matplotlib.pyplot as plt
import numpy as np

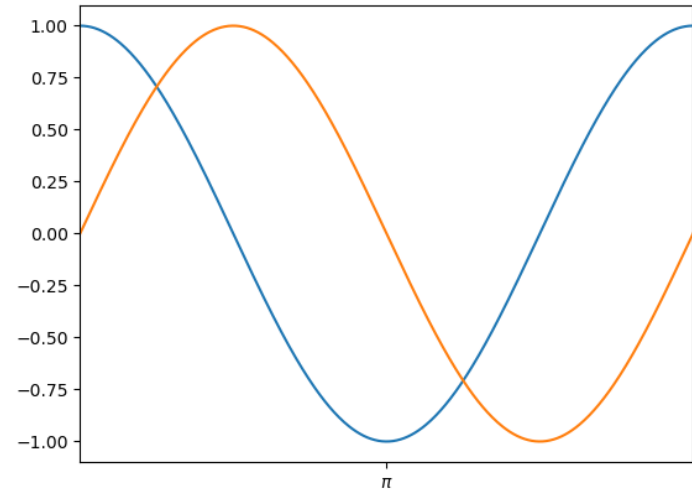
x = np.linspace(0, np.pi * 4, 1000)
y1 = np.cos(x)
y2 = np.sin(x)

fig, ax = plt.subplots()
ax.set_xlim(0, np.pi * 2)

ax.set_xticks([np.pi])
ax.set_xticklabels([r"$\pi$"])

ax.plot(x, y1)
ax.plot(x, y2)

fig.show()
```



Afficher plusieurs graphiques

```
import matplotlib.pyplot as plt
# ----- subplots -----
# 6 graphiques séparés en 2 lignes et 3 colonnes
fig, ax = plt.subplots(2, 3)
ax[0, 0].scatter(x1, y1)
ax[1, 0].scatter(x2, y2)
ax[0, 1].scatter(x3, y3)
# ...
# ----- ajouter des subplots -----
fig = plt.figure()
ax = fig.add_subplot(2, 3, 1)
ax.scatter(x, y)
ax = fig.add_subplot(2, 3, 2)
ax.scatter(x, y)
ax = fig.add_subplot(2, 3, 3)
ax.scatter(x, y)
# ...
```

Sauvegarder un graphique

```
fig = plt.figure()

# ...

fig.savefig(
    "monGraphique",
    # -- Optionel --
    dpi=150,
    format="png",
    transparent=True,
)
```



**Avez-vous des
questions ?**





Exercices

Matplotlib

