



# PYTHON

FONDAMENTAUX



DÉBUTANT



# SOMMAIRE

**Introduction**

**Mise en place de l'environnement**

**Affichage / Saisie**

**Les variables**

**Les conditions**

**Les boucles**

**Les listes**

**Les tuples**

**Les dictionnaires**

**Le système de gestion d'exception**

**Les fonctions**

**Manipuler des fichiers**

**Aller plus loin**





# INTRODUCTION

---

# PRÉSENTATION

Python est le langage le plus populaire, et est utilisé par de nombreuses très grandes entreprises telles que Google, Facebook, Dropbox... Python offre une syntaxe simple, une programmation orientée objet ainsi qu'une large collection de bibliothèques / modules.

## **Python est un langage interprété**

Comme à l'image du langage PHP, un programme Python ne nécessite pas d'étape de compilation en langage machine pour fonctionner. Le code est interprété à l'exécution.

## **Python est doté d'un typage dynamique fort**

Python effectue des vérifications de cohérence sur les types manipulés, et permet de transformer explicitement une variable d'un type à un autre.





# DOMAINES D'APPLICATION

ILS SONT NOMBREUX



**Intelligence Artificiel – IA**

<https://standardjs.com/>



**Data Science**

<https://github.com/airbnb/javascript>

**django**

**Développement Web et bien plus...**

<https://google.github.io/styleguide/jsguide.html>





# MISE EN PLACE DE L'ENVIRONNEMENT

---



# INTERPRÉTEUR PYTHON

## INSTALLATION

<https://www.python.org/downloads/>

1. Se rendre sur le lien ci contre pour télécharger le setup.

2. Choisir l'installation de Python 3.



Lors de l'installation, veillez à bien cocher la case :  
Add python to environment PATH



```
$ py / python / python3 --version  
Python 3.10.0
```

TERMINAL / CMD

VÉRIFICATION DE L'INSTALLATION

# CHOISIR UN IDE / ÉDITEUR DE TEXTE

## NOTRE ENVIRONNEMENT DE DÉVELOPPEMENT



**PyCharm**  
JetBrains



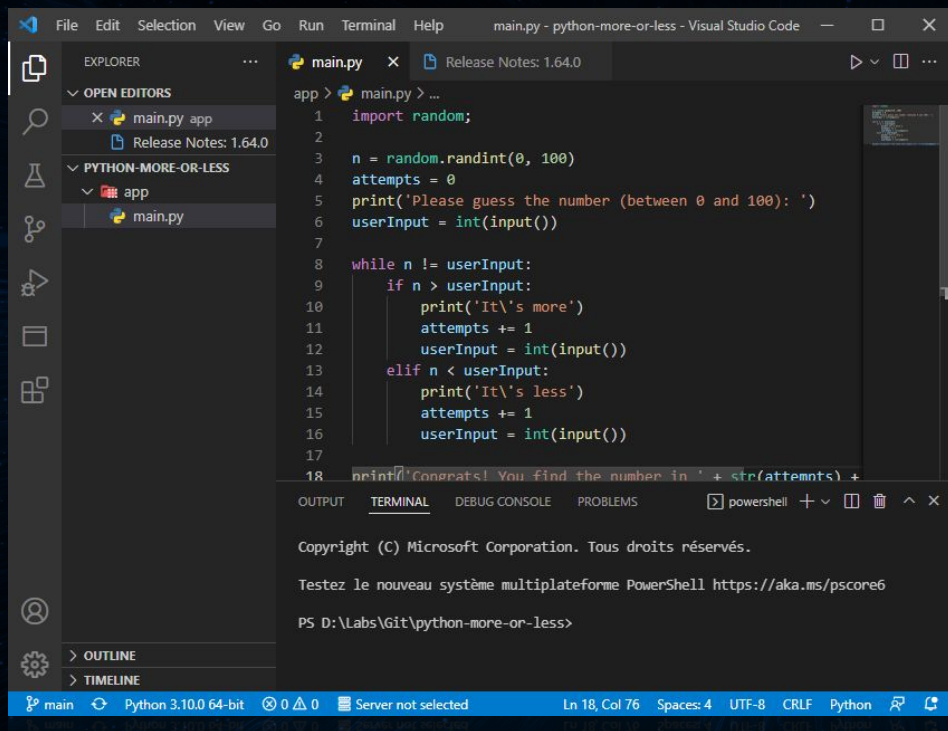
**Visual Studio Code**  
Microsoft

<https://code.visualstudio.com/>



# VISUAL STUDIO CODE

## INTERFACE & RACCOURCIS



**Gratuit**

**Système d'extensions**

**Connecté au système de versionning**

**Le plus populaire**

**Et bien plus...**



# EXTENSION PYTHON

UN INDISPENSABLE



**Python**  
Microsoft



**Notebooks**

**Snippets**

**Outils de debugging**

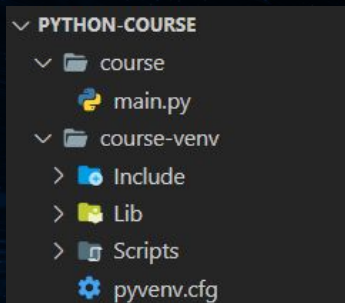
**Linter**

**Et bien plus...**



# CRÉATION DU PROJET

## KICK START PYTHON !



### ARBORESCENCE DU PROJET

1. Créer et ouvrir le dossier python-course à l'aide de VS. Code.

2. Créer un dossier course avec à l'intérieur le fichier main.py.

```
python -m venv course-venv
```

3. Créer l'environnement virtuel associé, course-venv.

```
py  
python3
```

```
course-venv\Scripts\Activate
```



4. Activer l'environnement virtuel.

# L'ENVIRONNEMENT VIRTUEL

VIRTUAL ENV - VENV

## CRÉATION DE L'ENVIRONNEMENT VIRTUEL

```
python -m venv <venv-name>
```

py  
python3

## ACTIVER L'ENVIRONNEMENT VIRTUEL

```
source <venv-name>/bin/Activate
```



```
<venv-name>\Scripts\Activate
```



L'environnement virtuel (venv) permet d'isoler les dépendances et l'exécution de python. La bonne pratique est d'utiliser un environnement virtuel par projet.



Attention, il faut être sur le cmd et non sur le powershell.





**AFFICHAGE / SAISIE**

---

# AFFICHAGE

## FONCTION PRINT

La fonction `print()` permet d'afficher un élément dans la sortie standard (console).



```
# Affiche dans la console "Hello World!"  
print('Hello World!')  
  
# Exemple avec des nombres  
print(5)  
print(5*5)
```

course/main.py



Hello World!

5  
25

SORTIE CONSOLE



# SAISIE

## FONCTION INPUT

La fonction `input()` permet d'établir une interaction avec l'utilisateur en lui demandant de saisir un texte.



```
my_input = input("Please enter smth...")  
  
# L'utilisateur saisie "I'm John Doe"  
print(my_input)
```

course/main.py



I'm John Doe

SORTIE CONSOLE

# COMMENTAIRES

#



```
# Commentaire monoligne
```

```
"""
```

```
Commentaire multiligne
```

```
Aussi appelé docstring
```

```
--
```

```
"""
```

course/main.py



Veuillez utiliser les commentaires avec parcimonie, un code bien écrit doit parler de lui-même.





# LES VARIABLES

---

# FONCTIONNEMENT

## MIEUX COMPRENDRE

La mémoire de l'ordinateur est constituée de cases. Chaque case a une adresse et un contenu, un nombre ou une chaîne de caractère par exemple.

Adresse	1	2	3	4	5
Contenu	32	25		"bonjour"	
Adresse	6	7	8	9	10
Contenu			37.5		



# FONCTIONNEMENT

## MIEUX COMPRENDRE

Les ordinateurs ont beaucoup de cases mémoires  
(on parle en milliards...).

C'est pourquoi les adresses sont très compliquées !



```
a = 25
```

```
course/main.py
```

Dans le code ci-dessus, nous mettons 5 dans une case que l'on appelle a. On ne connaît pas sa véritable adresse. (Ici 12015212)

C'est pourquoi les variables ont un nom afin de s'abstraire de la complexité des adresses et rendre notre code plus compréhensible.

Adresse	12015211	12015212	12015213	12015214	12015215
Contenu	32	25		"bonjour"	
Adresse	12015216	12015217	12015218	12015219	120152120
Contenu			37.5		



# RÈGLES À RESPECTER

## AVOIR UN CODE PROPRE

- ✗ Ne pas utiliser d'accent, ni de signe de ponctuation ou @.
- ✗ Les chiffres ne doivent pas être utilisés comme premier caractère.
- ✗ Ce qui se trouve à gauche du signe égal doit toujours être un nom de variable, et non une expression.
- ✓ Respecter la convention de nommage snake\_case.
- ✓ Privilégier l'écriture du code en anglais.

`m + 1 = b`



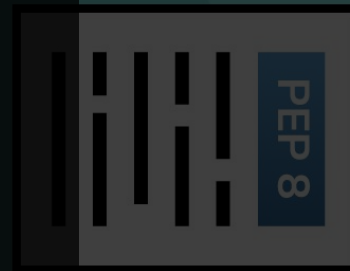


# CONVENTION PEP-8

## PYTHON STYLE GUIDE

PEP 8 (pour Python Extension Proposal) est un ensemble de règles qui permet d'homogénéiser le code et d'appliquer les bonnes pratiques.

L'exemple le plus connu est la guerre entre les développeurs à savoir s'il faut indenter son code avec des espaces ou avec des tabulations. La PEP 8 nous permet de trancher, ce sont les espaces qui gagnent, au nombre de 4. Fin du débat.



# TYPE DE DONNÉES

## LES TYPES SIMPLES

### INTEGER

```
age = 20
type(age)
# <class 'int'>
```

NOMBRE ENTIER

### FLOAT

```
mark = 15.5
type(mark)
# <class 'float'>
```

NOMBRE FLOTTANT

### STRING

```
sport = 'basket'
type(sport)
# <class 'str'>
```

CHAÎNE DE CARACTÈRES

### BOOLEAN

```
is_open = True
type(is_open)
# <class 'bool'>
```

VRAI OU FAUX



# OPÉRATIONS ENTRE LES NOMBRES

## MANIPULER LES TYPES NUMÉRIQUES

#	EXEMPLES	RÉSULTATS
Addition	$2 + 1$	3
Soustraction	$2 - 1$	1
Multiplication	$2 * 2$	4
Division	$17 / 4$	4.25
Division entière	$17 // 4$	4
Modulo	$17 \% 4$	1
Exposant	$5 ** 2$	25

# MANIPULER DES CHAÎNES DE CARACTÈRES

## FONCTIONS & MÉTHODES



```
hobby = 'Sport'

print(hobby.upper())
print(hobby.lower())
print(hobby.capitalize())
print(hobby.replace("r", "t"))
print(len(hobby))
```

course/main.py



```
SPORT
sport
Sport
Spott
5
```

SORTIE CONSOLE





# CONCATÉINATION

## ASSOCIER DEUX CHAÎNES DE CARACTÈRES



```
first = 'Hello'  
second = 'World!'  
  
print(first + ' ' + second)  
print(f'{first} {second}')
```

course/main.py



```
Hello World!  
Hello World!
```

SORTIE CONSOLE

Pour concaténer des chaînes de caractères, je recommande d'utiliser la seconde méthode, à savoir celle avec les accolades.

Elle est plus concise et plus lisible.

---



# CASTING

## CONVERSIONS DE TYPES



```
# Convertit le nombre entier 3 (int) en chaîne de caractère "3" (str)
str(3)

# Convertit la chaîne de caractères "3" (str) en nombre entier 3 (int)
int("3")

# Convertit la chaîne de caractères "5.6" (str) en nombre flottant 5.6 (float)
float("5.6")

# Convertit la chaîne de caractères "True" (str) en booléen True (bool)
bool("True")
```

course/main.py



Attention, on ne peut pas toujours caster, il faut rester logique...



```
>>> int("Bonjour")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: invalid literal for int()
with base 10: 'Bonjour'
```

SORTIE CONSOLE





# UTILITÉ DU CASTING

## EXEMPLE AVEC LA SAISIE

La fonction `input` retourne une chaîne de caractères, c'est pourquoi il est nécessaire de caster avant de réaliser l'addition des deux nombres.



```
a = input('Veuillez entrer un premier nombre : ')
b = input('Veuillez entrer un second nombre : ')
result = int(a) + int(b)

print(f'Le résultat de l\'addition est {result}')
```

course/main.py



```
>>> Veuillez entrer un premier nombre : 5
>>> Veuillez entrer un second nombre : 6

Le résultat de l'addition est 11
```

SORTIE CONSOLE



# TRAVAUX PRATIQUES

---



# ÉCHAUFFEMENT

## ÉCRITURE D'UN CONVERTISSEUR

```
>>> Veuillez entrer une vitesse en km/h : 50  
50.0 km/h = 31.08 m/h
```

SORTIE CONSOLE ATTENDUE

Écrire un programme qui convertit en m/h une vitesse donnée en km/h.  
Rappel : 1 mile = 1,609 km

Bonus : Arrondir le résultat au centième près.





# **BONUS : FIRESHIP**

---





# LES CONDITIONS

---

# IF... ELIF... ELSE...

## ÉCRITURE D'UNE STRUCTURE CONDITIONNELLE

L'instruction `if` exécute un bloc d'instruction si une condition donnée est vraie ou équivalente à vrai. Si la condition n'est pas vérifiée, il est possible d'utiliser une autre instruction.

Un bloc conditionnel sera toujours débuté par le mot clef `if` [si]. Celui-ci peut être suivi de plusieurs `elif` [sinon si] et finalisé par un `else` [sinon].



Pour un bloc conditionnel, une seule condition peut être validée. Les conditions sont évaluées dans l'ordre, les expressions suivantes ne seront donc pas évaluées.



```
points = 7

if points < 5:
    score = 0
elif points == 5:
    score = 10
elif 6 <= points <= 10:
    score = 50
else:
    score = 1000
```

course/main.py



# OPÉRATEURS DE COMPARAISON

--

OPÉRATEURS	DESCRIPTIONS
==	Égal à
!=	Différent de
>	Strictement supérieur à
>=	Supérieur ou égal à
<	Strictement inférieur à
<=	Inférieur ou égal à



# OPÉRATEURS LOGIQUES

## L'ALGÈBRE DE BOOLE

OPÉRATEURS	DESCRIPTIONS
not	NON Logique
and	ET Logique
or	OU Logique



```
age = 19
height = 1.68

if age >= 18 and height > 1.50:
    print('Chouette, vous avez accès au manège.')
else:
    print('Accès au manège refusé.')
```

course/main.py



Chouette, vous avez accès au manège.

SORTIE CONSOLE



# LES BOUCLES

---

# BOUCLES WHILE ET FOR

RÉPÉTER X FOIS DES INSTRUCTIONS



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

course/main.py



```
# Boucle : For
for i in range(0, 5):
    print(i)
```

course/main.py



```
0
1
2
3
4
```

SORTIE CONSOLE

Les boucles permettent de répéter des actions simplement et rapidement.

Une boucle peut être vue comme une version informatique de « copier N lignes » ou de « faire X fois quelque chose ».

---



# INSTRUCTION BREAK

## SORTIE PRÉMATURÉE D'UNE BOUCLE



```
# Boucle : For
for i in range(0, 100):
    if i > 5:
        print("Stop")
        break
    print(i)
```

course/main.py



```
0
1
2
3
4
5
Stop
```

SORTIE CONSOLE

L'instruction break peut s'avérer utile lorsqu'on souhaite sortir prématurément d'une boucle ou d'une condition switch [Python 3.10].

Elle ne fonctionne pas dans une condition if/else.

---

# INSTRUCTION CONTINUE

PASSER À L'ITÉRATION SUIVANTE



```
# Boucle : For
for i in range(0, 5):
    if i == 3:
        # Saut de la valeur 3
        print("-")
        continue
    print(i)
```

course/main.py



```
0
1
2
-
4
```

SORTIE CONSOLE

L'instruction continue arrête l'exécution des instructions pour l'itération de la boucle. L'exécution est reprise à l'itération suivante.



## BONUS : VS. CODE

---





# TRAVAUX PRATIQUES

---



# LE JEU DU PLUS OU MOINS

Le programme tire un nombre aléatoire entre 0 et 100. Ensuite, la personne doit trouver le nombre en recevant comme indication si le nombre est trop grand ou trop petit.

A la fin, le programme doit dire combien de coups ont été joués.



```
Entrer un nombre : 70
Le nombre est plus grand que 70
Entrer un nombre : 88
Le nombre est plus petit que 88
Entrer un nombre : 84
Le nombre est plus grand que 84
Entrer un nombre : 85
Bravo, vous avez trouvé en 4 essais
```

SORTIE CONSOLE ATTENDUE

## BONUS

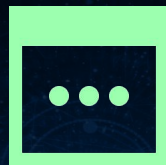
Faire un menu permettant de choisir une difficulté :

Facile : 0 à 10.

Moyen : 0 à 100.

Difficile : 0 à 1000.





# LES LISTES

---



# LISTES

## STOCKER PLUSIEURS VALEURS DANS UNE MÊME VARIABLE



L'index d'une liste est un nombre entier qui commence toujours à 0.



```
cities = ['Paris', 'Lyon', 'Marseille']  
print(cities[2])  
  
for city in cities:  
    print(city)
```

course/main.py



```
Marseille  
  
Paris  
Lyon  
Marseille
```

SORTIE CONSOLE

Une liste est une collection ordonnée, elle contient plusieurs éléments dans un ordre précis. L'index permet de récupérer précisément un élément d'une liste.

# MANIPULER DES LISTES

## AJOUT, RECHERCHE ET SUPPRESSION D'UN ÉLÉMENT



```
hobbies = ['Sport', 'Music']

# Ajouter un élément
# append() ajoute un élément en fin de liste.
hobbies.append('Movies')
# insert() ajoute un élément à l'index indiqué, ici 0.
hobbies.insert(0, 'Games')

# Rechercher un élément
# index() retourne l'index de l'élément indiqué.
hobbies.index('Music')

# Supprimer un élément
# del permet de supprimer un élément via l'index.
del hobbies[1]
# remove supprime l'élément via la valeur.
hobbies.remove('Movies')
```

course/main.py



```
['Sport', 'Music', 'Movies']

['Games', 'Sport', 'Music', 'Movies']

2

['Games', 'Music', 'Movies']

['Games', 'Music']
```

SORTIE CONSOLE - print(hobbies)

# MANIPULER DES LISTES

## CONCATÉNATION, CONDITION ET SEGMENT



```
hobbies = ['Sport', 'Music']

# Concaténation de listes
# extend() étend la liste.
hobbies.extend(['Travel', 'Writing'])
# Concaténation en créant une nouvelle liste.
new_hobbies = hobbies + ['Movies']

# Condition avec les listes
if 'Sport' in hobbies:
    input('Ah génial, quel sport pratiques-tu ?')

# Segment de liste
# Retourne une sous-liste entre l'index 1 et l'index 2.
hobbies[1:3]
```

course/main.py



```
['Sport', 'Music', 'Travel', 'Writing']

['Sport', 'Music', 'Travel', 'Writing', 'Movies']

'Ah génial, quel sport pratiques-tu ?'
>>> ...

['Music', 'Travel']
```

SORTIE CONSOLE - print(hobbies) / print(new\_hobbies)



# MANIPULER DES LISTES

## TRI ET FONCTIONS



```
marks = [6, 18, 12, 15.5, 9]

# Tri de liste
# sort() ordonne les éléments de la liste, cette méthode retourne None.
marks.sort()
# reverse() inverse l'ordre des éléments de la liste, cette méthode retourne None.
marks.reverse()
# sorted(...) génère une nouvelle liste avec les éléments ordonnés.
result = sorted(marks)

# Fonction applicable sur une liste
# Retourne la taille de la liste
result = len(marks)
# Retourne le plus petit élément de la liste
result = min(marks)
# Retourne le plus grand élément de la liste
result = max(marks)
# Retourne la somme des éléments de la liste
result = sum(marks)
```

course/main.py



```
[6, 9, 12, 15.5, 18]
```

```
[18, 15.5, 12, 9, 6]
```

```
[6, 9, 12, 15.5, 18]
```

```
5
```

```
6
```

```
18
```

```
60.5
```

SORTIE CONSOLE - print(marks) / print(result)

# LISTES ET CHÂÎNES DE CARACTÈRES

## SPLIT ET JOIN



```
# Chaîne de caractères --> Liste
# split(sep) sépare la chaîne de caractères via un séparateur.
email = 'john.doe@yahoo.fr'
result = email.split('.')

# Liste --> Chaîne de caractères
# join(list) joint les éléments de la liste via un caractère.
cities = ['Paris', 'Monaco', 'Limoges']
result = '-'.join(cities)
```

course/main.py
















```
['john', 'doe@yahoo', 'fr']
```

```
Paris-Monaco-Limoges
```

SORTIE CONSOLE - print(result)

# PYTHON : CHEAT SHEET

MÉTHODES DE LISTE

	<code>.append (🍏)</code>	➡	
	<code>.clear ()</code>	➡	
	<code>.copy ()</code>	➡	
	<code>.count (🍏)</code>	➡	<code>2</code>
	<code>.pop (1)</code>	➡	
	<code>.remove (🍏)</code>	➡	
	<code>.reverse ()</code>	➡	





# TRAVAUX PRATIQUES

---



# LES VALEURS UNIQUES

Écrire un programme qui créera une nouvelle liste en ne gardant que les valeurs uniques.

Cf. Exemple ci-dessous.

---



```
numbers = [8, 7, 11, 7, 2, 10, 5, 8]

# Votre programme...

result = [8, 7, 11, 2, 10, 5]
```

[course/unique.py](#)

L'utilisateur entre un nombre. Le programme stock les 10 premiers résultats de la table de multiplication choisie par l'utilisateur dans une liste puis affiche cette liste.

### BONUS

Signaler au passage, à l'aide d'une astérisque, ceux qui sont des multiples de 3



```
Quelle table voulez-vous afficher ? 5
```

```
[5,10,15,20,25,30,35,40,45,50]
```

```
[5,10,'15*',20,25,'30*',35,40,'45*',50]
```

BONUS

SORTIE CONSOLE ATTENDUE



## LA TABLE DE MULTIPLICATION





# LES TUPLES

---

# TUPLES

## ENSEMBLE DE VALEURS NON MODIFIABLES



```
# Tuple
fruits = ('banana', 'peach', 'apple')
print(type(fruits))

# Unpacking
mass, speed = (3, 12.7)
print(mass)
print(speed)
```

course/main.py



```
<class 'tuple'>
```

```
3
12.7
```

SORTIE CONSOLE

Les tuples sont des variables en lecture seule, on ne peut pas les modifier.

Les tuples sont souvent utilisés en retour de fonction car ils consomment moins d'espace mémoire que les listes.

---





# LES DICTIONNAIRES

---



# DICTIONNAIRES

## ENSEMBLE DE PAIRES CLEFS / VALEURS



```
# Dictionnaire
car = {
    "brand": "Kia",
    "model": "Rio",
    "version": "3",
    "doors_nbr": 3,
    "options": ['rims', 'ESC']
}
```

course/main.py

Les dictionnaires est un ensemble d'éléments non triés ou chacun est associé à une clef.

Nous pouvons faire l'analogie avec un dictionnaire de français où l'on peut à partir d'un mot accéder à sa définition.

---



# MANIPULER DES DICTIONNAIRES

## AJOUT, MODIFICATION ET SUPPRESSION D'UN ÉLÉMENT



```
# Dictionnaire
user = {
    "last_name": "Doe",
    "first_name": "John",
    "age": 20
}

# Ajout d'une paire clef/valeur
user["email"] = "john.doe@xyz.fr"
user.update({ "email": "john.doe@xyz.fr" })

# Modification d'une valeur
user["age"] = 22
user.update({ "age": 22 })

# Suppression d'une paire clef/valeur
del user["first_name"]
```

course/main.py

Dict.

# ITÉRER DANS UN DICTIONNAIRE

## PARCOURIR UN DICTIONNAIRE À L'AIDE D'UNE BOUCLE



```
user = {  
    "last_name": "Doe",  
    "first_name": "John",  
    "age": 20  
}  
  
# Retourne la liste des clefs du dictionnaire  
list(user)  
  
# Retourne un objet dict_keys  
# (Liste) des clefs du dictionnaire  
user.keys()  
  
# Retourne un objet dict_values  
# (Liste) des valeurs du dictionnaire  
user.values()  
  
# Retourne un objet dict_items  
# (Liste) de tuples (clef, valeur) du dictionnaire  
user.items()
```

course/main.py

Pour parcourir un dictionnaire proprement, il existe des méthodes adaptées [cf. ci-contre].

Ce sont celles-ci que l'on va venir utiliser en combinaison de la boucle for.



# ITÉRER DANS UN DICTIONNAIRE

## PARCOURIR UN DICTIONNAIRE À L'AIDE D'UNE BOUCLE



```
user = {  
    "last_name": "Doe",  
    "first_name": "John",  
    "age": 20  
}  
  
# Boucle, en ayant accès aux clefs  
for key in user.keys():  
    print(key)  
  
# Boucle, en ayant accès aux valeurs  
for value in user.values():  
    print(value)  
  
# Boucle, en ayant accès aux clefs/valeurs  
for key, value in user.items():  
    print(f'{key} - {value}')
```

course/main.py



```
last_name  
first_name  
age
```

```
Doe  
John  
20
```

```
last_name - Doe  
first_name - John  
age - 20
```

SORTIE CONSOLE

# DICTIONNAIRE ET CONDITION

```
user = {  
    "last_name": "Doe",  
    "first_name": "John"  
}  
  
if 'age' not in user.keys():  
    user['age'] = int(input('Entrer votre âge : '))
```

course/main.py

Entrer votre âge :

SORTIE CONSOLE

# LES ITÉRABLES

## CHAÎNE DE CARACTÈRES

```
greetings = 'Hi!'

for char in greetings:
    print(char)
```

## SORTIE CONSOLE

```
H
i
!
```

## LISTE

```
marks = [6, 18, 12]

for mark in marks:
    print(mark)
```

## SORTIE CONSOLE

```
6
18
12
```

## TUPLE

```
fruits = ('apple', 'cherry')

for fruit in fruits:
    print(fruit)
```

## SORTIE CONSOLE

```
apple
cherry
```

## DICTIONNAIRE

```
user = {
    "last_name": "Doe",
    "first_name": "John"
}

for key, value in user.items():
    print(f'{key} - {value}')
```

## SORTIE CONSOLE

```
last_name - Doe
first_name - John
```

# PARCOURIR





# TRAVAUX PRATIQUES

---

# ÉCHAUFFEMENT

## Client 1

Nom : Doe

Prénom : John

Age : 21

Email : john.doe@xyz.com

Hobbies : Karaté, Tennis

## Client 2

Nom : Stewart

Prénom : Jane

Age : 26

Email : [Email non renseigné]

Hobbies : Danse, Peinture, Chant

## Client 3

Nom : Tardieu

Prénom : Olivier

Age : 32

Email : olivier.tardieu@xyz.com

Hobbies : [Hobbies non renseignés]

Créer un nouveau fichier `customers.py`.

A l'aide de listes et de dictionnaires, trouver le meilleur moyen de stocker toutes les données ci-contre dans une seule et unique variable.

---



# ÉPELER UN NUMÉRO

Écrire un programme qui épellera un numéro de téléphone en toute lettre.

---

Entrer un numéro de téléphone :

>>> 0265987412

Zero Two Six Five Nine Eight Seven Four One Two

SORTIE CONSOLE ATTENDUE



Faire une analyse statistique d'un texte, celle-ci affichera un dictionnaire avec le décompte de chaque lettre.



```
text = 'Hello World!'

# Votre programme...

# Le résultat va de a à z.
# Par souci de lisibilité, l'exemple s'arrête à la lettre d.
result_dict = { 'a': 0, 'b': 0, 'c': 0, 'd': 1 }
```

course/count\_letters.py

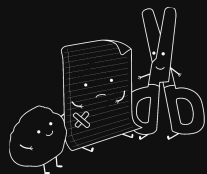
### Exemple de texte

Two roads diverged in a yellow wood,  
And sorry I could not travel both  
And be one traveler, long I stood  
And looked down one as far as I could  
To where it bent in the undergrowth.



## COMPTER LES LETTRES





# PIERRE / FEUILLE / CISEAUX

Écrire dans un nouveau fichier `shifumi.py`, un programme qui permettra de jouer à Pierre / Feuille / Ciseaux contre l'ordinateur en 2 manches gagnantes [BO3].





# LE SYSTÈME DE GESTION D'EXCEPTION

---



# LE MESSAGE D'ERREUR

## LIRE ET COMPRENDRE UN MESSAGE D'ERREUR

Le système de gestion d'exception est mis en place pour éviter que le programme "crash". Le but est d'anticiper les erreurs potentielles et de programmer une réaction à celles-ci. En Python quand une erreur survient, le programme lève un Traceback avant de s'arrêter.

Le Traceback est composé de 3 éléments :

- La **Stack Trace**, liste de l'ensemble des appels qui ont mené à la ligne en erreur.
- Le **type** d'erreur.
- Le **message** d'erreur.

```
>>> a = 5 / 0
Traceback (most recent call last)
  File "d:\Labs\Git\python-course\course\main.py", line 1, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
```

SORTIE CONSOLE [Exemple d'erreur]

# TRY – EXCEPT

## LIRE ET COMPRENDRE UN MESSAGE D'ERREUR



```
number = input("Entrer un nombre")

# Dans le bloc try, on écrit les lignes qui peuvent lever une erreur.
try:
    number = int(number)
# Dans le bloc except, on implémente une réaction en cas d'erreur.
except ValueError:
    print("Vous n'avez pas entré un nombre...")
```

course/main.py



Il est important de préciser le type d'erreur potentiel [ici, ValueError] afin d'éviter les erreurs silencieuses.



# TRY – EXCEPT

## UTILISATION DE PLUSIEURS BLOCS EXCEPT



```
number = input("Entrer un nombre")
try:
    number = int(number)
    total = 10 / number
# ValueError sera levée si le casting n'est pas possible.
except ValueError:
    print("Vous n'avez pas entré un nombre...")
# ZeroDivisionError sera levée s'il y a une division par 0.
except ZeroDivisionError:
    print("La division par 0 est impossible...")
```

course/main.py



# EXCEPTION

## GÉRER UNE ERREUR NON RÉPERTORIÉE



```
number = input("Entrer un nombre")
try:
    number = int(number)
    total = 10 / number
# ValueError sera levée si le casting n'est pas possible.
except ValueError:
    print("Vous n'avez pas entré un nombre...")
# ZeroDivisionError sera levée s'il y a une division par 0.
except ZeroDivisionError:
    print("La division par 0 est impossible...")
except Exception:
    print("Une erreur est survenue...")
```

course/main.py

L'erreur de type Exception englobe tous les types d'erreurs possibles. Il est souvent utilisé en tant que "fallback".

# ELSE

SI TRY SE DÉROULE CORRECTEMENT, LE BLOC ELSE SERA EXÉCUTÉ



```
number = input("Entrer un nombre")
try:
    number = int(number)
    total = 10 / number
# ValueError sera levée si le casting n'est pas possible.
except ValueError:
    print("Vous n'avez pas entré un nombre...")
# ZeroDivisionError sera levée s'il y a une division par 0.
except ZeroDivisionError:
    print("La division par 0 est impossible...")
# Si aucune exception n'est levée, else sera exécuté.
else:
    print(f"Total : {total} - Tout s'est bien passé !")
```

course/main.py

Il est également possible d'utiliser le mot clef else dans un système de gestion d'exception.

Le bloc else permet d'exécuter une portion du code seulement si le bloc try s'est correctement déroulé et qu'aucune exception n'a été levée.

---



# FINALLY

QUOI QU'IL ARRIVE LE BLOC FINALLY SERA EXÉCUTÉ



```
number = input("Entrer un nombre : ")
try:
    number = int(number)
    total = 10 / number
# ValueError sera levée si le casting n'est pas possible.
except ValueError:
    print("Vous n'avez pas entré un nombre...")
# ZeroDivisionError sera levée s'il y a une division par 0.
except ZeroDivisionError:
    print("La division par 0 est impossible...")
# Si aucune exception n'est levée, else sera exécuté.
else:
    print(f"Total : {total} - Tout s'est bien passé !")
# Erreur ou pas, finally sera exécuté.
finally:
    print("La gestion des erreurs / exceptions est terminée.")
```

course/main.py

Enfin il est possible de clôturer le système de gestion d'exception avec le mot clef finally.

Ce bloc permet d'exécuter des instructions que le bloc try passe ou non.

\_\_\_\_\_



# RAISE

## LEVER UNE EXCEPTION



```
x = -3
if x < 0:
    raise ValueError("x should not have a negative value")
```

course/main.py

L'instruction raise permet de lever une exception. Pour la plupart, il est possible de passer en paramètre un message pour décrire le cas d'erreur.

## QUELQUES TYPES D'ERREURS RENCONTRÉS

**TypeError**: unsupported operand type(s) for int and str

Signale que le type de la donnée n'est pas correct pour l'instruction à exécuter.

**IndexError**: string index out of range

Index invalide lors de l'accès à un élément dans une liste.

**KeyError**: 'first\_name'

Signale que la clé n'existe pas dans un dictionnaire.

**NotImplementedError**: 'get\_user'

La méthode ou la fonction n'est pas implémentée.



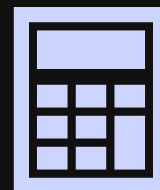
# TRAVAUX PRATIQUES

---

Écrire un programme qui demande en continue à l'utilisateur d'entrer des notes d'élèves. La boucle se termine seulement si l'utilisateur entre une valeur négative. Lorsque cela arrive, afficher le nombre de notes entrées, et calculer la moyenne de toutes les notes.

Si l'utilisateur entre autre chose qu'un nombre, l'erreur doit être traitée.

Si l'utilisateur ne rentre aucune note et quitte le programme immédiatement, l'erreur doit également être traitée.



## LA MOYENNE DES NOTES

```
Enter un nombre positif : 18
Enter un nombre positif : H
Ceci n'est pas un nombre...
Enter un nombre positif : 10
Enter un nombre positif : 7
Enter un nombre positif : 12
Enter un nombre positif : -6
La moyenne des 4 notes est : 11.75

--
Enter un nombre positif : -5
Il y a : 0 note(s)
Vous n'avez pas saisi de note...
```

SORTIE CONSOLE ATTENDUE



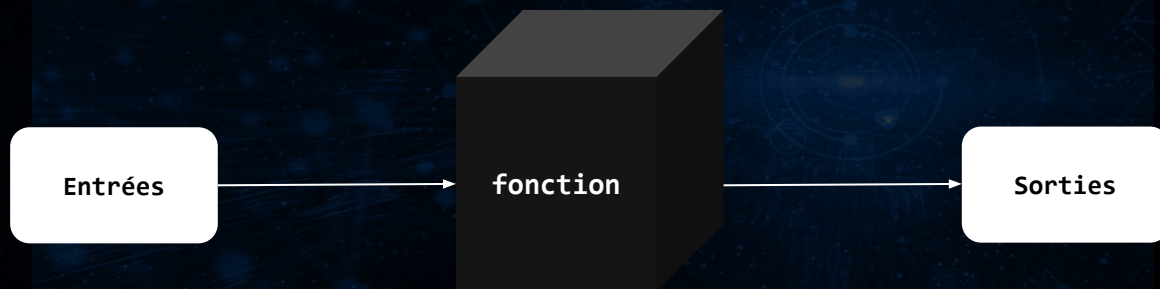


# LES FONCTIONS

---

# KÉSAKO ?

## LA MÉTAPHORE DE LA BOÎTE NOIRE



Une fonction est une série d'instructions qui peut prendre des données en entrée et retourner zéro, une ou plusieurs valeurs en sortie.

Elle constitue un premier niveau d'organisation au sein de notre code.

---

# DÉFINITION & APPEL

## MIEUX COMPRENDRE



# Définition de la fonction d'addition

```
def addition():  
    print(5 + 2)
```

# Appel de la fonction d'addition

```
addition()
```

course/main.py



Les fonctions permettent d'implémenter un comportement une seule fois puis de le réutiliser autant de fois que nécessaire. Elles apportent un premier niveau d'organisation et facilitent la maintenance et l'évolution des programmes.

La définition de la fonction ne sera pas exécutée par votre programme, c'est lors de l'appel à cette fonction que celui-ci réalisera son exécution.

7

SORTIE CONSOLE



# RETOURNER UNE VALEUR

## L'INSTRUCTION RETURN



```
# Définition de la fonction d'addition
def addition():
    return 5 + 2

# Appel de la fonction d'addition
# Affectation du retour de la fonction dans la variable result
result = addition()

print(result)
```

course/main.py



Suite au traitement réalisé par une fonction, il est possible d'en récupérer le résultat en dehors de celle-ci.

Ce résultat sera la valeur de retour de la fonction marquée en Python par le mot clef return.

7

SORTIE CONSOLE

# RETOURNER PLUSIEURS VALEURS

## LES TUPLES À LA RESCousse



```
# Définition de la fonction get_user
def get_user():
    return 'John', 'Doe', 24

# Appel de la fonction get_user
# Affichage du retour de la fonction get_user
print(get_user())
```

course/main.py



```
('John', 'Doe', 24)
```

SORTIE CONSOLE

Pour qu'une fonction puisse retourner plusieurs valeurs, nous utiliserons des tuples, elles prennent moins d'espace mémoire qu'une liste.

# ARGUMENTS

## LES DONNÉES EN ENTRÉE



```
# Définition de la fonction d'addition
def addition(x, y):
    return x + y

# Appel de la fonction d'addition
# Affectation du retour de la fonction dans la variable result
result = addition(5, 3)

print(result)
```

course/main.py



Afin que notre fonction d'addition soit plus générique, il est possible de passer des paramètres en entrée.

Ces paramètres devront être définis lors de l'appel de la fonction.

8

SORTIE CONSOLE



# ARGUMENTS

## POSITIONNELS & NOMMÉS



```
def addition(x, y):  
    return x + y  
  
addition(2, 3)
```

### ARGUMENTS POSITIONNELS

Les arguments positionnels sont obligatoires. Ils doivent être passés dans l'ordre défini lors de la déclaration de la fonction.



```
def addition(a = 0, b = 0):  
    return a + b  
  
addition()  
addition(1, 3)  
addition(a = 2, b = 3)  
addition(b = 1)
```

### ARGUMENTS NOMMÉS

Les arguments nommés sont facultatifs, s'ils ne sont pas renseignés lors de l'appel à la fonction, ils seront remplacés par leur valeur par défaut.

Il existe deux types d'arguments, les arguments positionnels et les arguments nommés. Ils sont différenciables lors de la déclaration de la fonction.

# PASS

## UNE INSTRUCTION UTILE POUR STRUCTURER SA RÉFLEXION ET ÉVITER LES ERREURS



```
expr = 1

# Condition
if expr:
    pass

# Boucle
for i in range(0, 10):
    pass

# Fonction
def get_cars():
    pass

# Classe (POO)
class User:
    pass
```

course/main.py

Supposons que nous ayons une boucle ou une fonction qui ne soit pas encore implémentée, mais que nous souhaitons l'implémenter à l'avenir.

L'interpréteur lève une erreur lorsque le bloc est vide. C'est là que l'instruction `pass` permet de définir un bloc vide sans lever d'erreur.

---

# \*ARGS

## ARGUMENTS POSITIONNELS MULTIPLES



# Définition de la fonction d'addition

```
def addition(*args):
```

```
    result = 0
```

```
    for arg in args:
```

```
        result += arg
```

```
    return result
```

# Appel de la fonction d'addition

```
result = addition(1, 5, 17, 9)
```

```
print(result)
```

course/main.py



32

SORTIE CONSOLE

Le paramètre `*args` (tuple) permet définir une fonction, qui lors de son appel peut prendre un nombre illimité d'arguments positionnels.

Dans l'exemple ci-contre, la fonction d'addition retournera la somme de tous les arguments en entrées.



# \*\*KWARGS

## ARGUMENTS NOMMÉES MULTIPLES



```
# Définition de la fonction show_user_details
def show_user_details(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} - {value}')

# Appel de la fonction show_user_details
show_user_details(first_name = 'John',
                  last_name = 'Doe',
                  age = 22,
                  size = 1.78)
```

course/main.py



```
first_name - John
last_name - Doe
age - 22
size - 1.78
```

SORTIE CONSOLE

Le paramètre `**kwargs` (dict.) permet définir une fonction, qui lors de son appel peut prendre un nombre illimité d'arguments nommés.

Dans l'exemple ci-contre, la fonction `show_user_details` affichera les détails d'un utilisateur.

---

# AIDE AU DÉVELOPPEUR

## BIEN ÉCRIRE UNE FONCTION – TYPE HINTS / INDICATIONS DE TYPE



# Définition de la fonction d'addition

```
def addition(x:int, y:int) -> int:  
    return x + y
```

# Définition de la fonction greetings

```
def greetings(user:dict[str, str]) -> None:  
    print(f'Hello {user["first_name"]}!')
```

# Définition de la fonction get\_max

# A vous de l'implémenter :)

```
def get_max(numbers:list[float]) -> float:  
    pass
```

course/main.py

Lors de l'écriture d'une fonction, il est possible de préciser le type des paramètres en entrées ainsi que le type de la valeur de retour.

Cela permet de structurer notre pensée lors de l'implémentation et fait office de documentation.

Ne vous en privez pas !

---



# FUNCTION HELP

## DOCUMENTATION



```
def get_users():  
    """  
        Return a list of all users.  
        A user is defined by their first name, last name and age.  
    """  
    return [  
        {  
            "first_name": "John",  
            "last_name": "Doe",  
            "age": 22  
        },  
        {  
            "first_name": "Jane",  
            "last_name": "Doe",  
            "age": 20  
        }  
    ]  
  
help(get_users)
```

course/main.py



Help on function get\_users in module \_\_main\_\_:

```
get_users()  
    Return a list of all the users.  
    A user is defined by their first name, last name and age.
```

SORTIE CONSOLE







# EXPRESSION LAMBDA / FONCTION ANONYME

UNE FONCTION SANS NOM



```
# Exemple d'une expression lambda additionnant x et y
lambda x, y: x + y
print(lambda x, y: x + y)

# Nous pouvons lui donner un nom en l'affectant à une variable
# Ainsi nous pouvons l'appeler
addition = lambda x, y: x + y
print(addition(5, 8))
```

course/main.py



```
<function <lambda> at 0x0000020F40C63E20>
```

```
13
```

SORTIE CONSOLE





# APPLICATION CONCRÈTE

## C'EST MIGNON TOUT ÇA, MAIS À QUOI ÇA SERT ?

SIGNATURE DE LA FONCTION SORT(...)

sort(\*, key: `lambda`, reverse: `bool`) -> `None`



```
# Objectif : Trier la liste des auteurs par leurs noms.
authors = [
    'Frank Herbert',
    'Isaac Asimov',
    'Arthur C. Clarke',
    'Douglas Adams',
]

authors.sort(key = lambda elem: elem.split(' ')[-1])
print(authors)
```

course/main.py



```
[
    'Douglas Adams',
    'Isaac Asimov',
    'Arthur C. Clarke',
    'Frank Herbert'
]
```

SORTIE CONSOLE



# APPLICATION CONCRÈTE : MULTIPLICATOR

## UNE LAMBDA EN RETOUR DE FONCTION



```
from typing import Callable

def multiplicator(number:int) -> Callable:
    return lambda x: x * number

double = multiplicator(2)
triple = multiplicator(3)

print(double(5))
print(triple(10))
```

course/main.py



```
10
30
```

SORTIE CONSOLE

Le fait qu'on puisse retourner une lambda à la fin d'une fonction permet de créer un modèle de fonction, ici mutiplicator.

Dans l'exemple ci-contre, à partir de ce modèle, nous avons pu créer les fonctions double et triple.

---





# GÉNÉRATEURS

## L'INSTRUCTION YIELD



```
from typing import Generator

def square(max:int) -> Generator:
    for i in range(1, max + 1):
        yield i**2

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

course/main.py



```
1
4
9
16
25
```

SORTIE CONSOLE

Les générateurs sont des fonctions un peu spéciales qui peuvent être appelés un nombre limité de fois. Ils sont souvent utilisés avec la boucle for.

---



# GÉNÉRATEURS

LA FONCTION NEXT POUR BIEN COMPRENDRE



```
--
```

course/main.py



```
1
4
9
16
25
```

SORTIE CONSOLE

La fonction next permet de passer à la valeur suivante d'un générateur jusqu'à épuisement de celui-ci.

---





# LISTES OU GÉNÉRATEURS ?

## LES DIFFÉRENCES

### LISTES

Prennent plus de place en mémoire

Toutes les valeurs existent ensemble

Occupent un espace fini

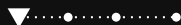
### GÉNÉRATEURS

Prennent moins de place en mémoire

Seulement une valeur existe à un instant t

Peuvent occuper un espace infini

Générateur



[15.5, 6, 8, 9, 10]



Liste

.....



PYTHON  
simplified

## BONUS : PYTHON SIMPLIFIED

---



# TRAVAUX PRATIQUES

---



# FIZZ BUZZ

Le Fizz Buzz est un exercice courant en entretien d'embauche.

Pour chaque multiple de 3 :	Afficher Fizz
Pour chaque multiple de 5 :	Afficher Buzz
Pour les multiples de 3 et 5 :	Afficher Fizz Buzz
Pour tous les autres :	Afficher le nombre.

```
fizzbuzz(16)
```

```
1  
2  
Fizz  
4  
Buzz  
Fizz  
7  
8  
Fizz  
Buzz  
11  
Fizz  
13  
14  
Fizz Buzz  
16
```

course/fizzbuzz.py

SORTIE CONSOLE ATTENDUE



A l'aide de vos recherches, écrire une fonction permettant de générer une couleur [RGB] aléatoire.

### BONUS

Écrire une seconde fonction s'appuyant sur la première, celle-ci permettra d'écrire un message d'une couleur aléatoire dans la console.

### AIDES

```
import random
from sty import fg
```

MODULE À IMPORTER

```
pip install sty
py -m pip install sty
```

CONSOLE



```
print(color, "Hello")
print(color, "Hello")
```

course/random\_color.py



```
Hello
Hello
```

SORTIE CONSOLE ATTENDUE



# COULEUR ALÉATOIRE

PYTHON SIMPLIFIED - RGB COLOR MODE





# LE CHIFFRE DE CÉSAR

## CRYPTOGRAPHIE

ABCDEFGHIJKLMNOPQRSTUVWXYZ

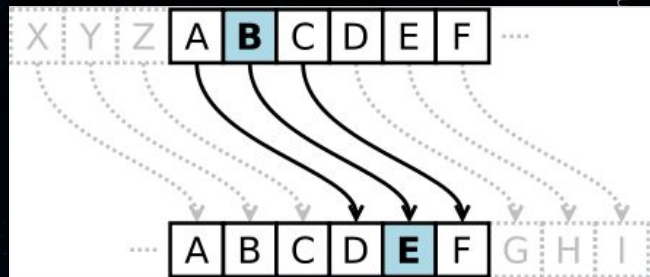
def caesar\_cipher(...):

DEFGHIJKLMNOPQRSTUVWXYZABC

EXEMPLE POUR UN DÉCALAGE DE 3

Écrire la fonction qui permet de réaliser le chiffre de César avec un décalage de 1 dans un premier temps.

Dans un second temps, ajouter le décalage en paramètre de la fonction.



Le chiffre de César fonctionne par décalage des lettres de l'alphabet.

Par exemple, dans l'image ci-dessus, il y a une distance de 3 caractères, donc B devient E dans le texte codé.

### Exemple de texte à chiffrer

Two roads diverged in a yellow wood,  
And sorry I could not travel both  
And be one traveler, long I stood  
And looked down one as far as I could  
To where it bent in the undergrowth.





# MANIPULER DES FICHIERS

---





# LES TYPES DE FICHIERS

## LES FICHIERS PLATS

### FICHIERS TEXTUELS

Texte en clair

CSV

JSON

Code source

### FICHIERS BINAIRES

Code compilé

Image

Audio

Vidéo

Il existe différents types de fichiers, nous allons nous concentrer sur la manipulation de fichiers textuels.

# CRÉATION D'UN FICHIER TEXTE

POEM.TXT

```
mkdir data
```



1.

Créer un dossier data

```
ni data/poem.txt
```



2.

À l'intérieur du dossier data, créer un fichier poem.txt

3.

Copier et coller le contenu suivant dans le fichier poem.txt :

Two roads diverged in a yellow wood,  
And sorry I could not travel both  
And be one traveler, long I stood  
And looked down one as far as I could  
To where it bent in the undergrowth.

---



# LA FONCTION OPEN

## OUVRIR UN FICHIER

**SIGNATURE DE LA FONCTION** `open( ... )`

`open(file:str, mode='r': str, ...) -> None`

MODE	SIGNIFICATION
'r'	Ouvre en lecture [ <b>Par défaut</b> ]
'w'	Ouvre en écriture, en effaçant le contenu du fichier
'x'	Ouvre pour une création exclusive, échouant si le fichier existe déjà
'a'	Ouvre en écriture, ajout en fin de fichier
'b'	Mode binaire
't'	Mode texte [ <b>Par défaut</b> ]
'+'	Ouvre en modification [ <b>Lecture &amp; Écriture</b> ]





# LIRE LE FICHIER POEM.TXT

LA FONCTION OPEN EN ACTION !



```
f = open('./data/poem.txt')  
poem = f.read()  
f.close()  
  
print(poem)
```

course/main.py



```
Two roads diverged in a yellow wood,  
And sorry I could not travel both  
And be one traveler, long I stood  
And looked down one as far as I could  
To where it bent in the undergrowth.
```

SORTIE CONSOLE

Python propose une façon simple de lire un fichier textuel, avec la manière ci-contre, il ne faut pas oublier de "close" le fichier au risque de créer une fuite mémoire dans notre programme.

Il existe une bien meilleure façon de faire, rendez-vous sur la slide suivante !

\_\_\_\_\_

# LIRE LE FICHIER POEM.TXT

INSTRUCTION WITH - GESTIONNAIRE DE CONTEXTE



```
with open('./data/poem.txt') as f:
    poem = f.read()

print(poem)
```

course/main.py



```
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth.
```

SORTIE CONSOLE

Ici, l'instruction with est ce que l'on appelle un gestionnaire de contexte, il va s'occuper pour nous d'éviter les fuite mémoire de notre programme.

Dans l'exemple ci-contre, nous remarquons que l'on a plus besoin de "close" notre fichier car l'instruction with s'en occupe.

Pour optimiser votre programme, je vous recommande vivement d'utiliser "with" lorsque vous manipulez une ressource externe.



# FICHIER NON TROUVÉ

QUE SE PASSE T-IL SI ON ESSAIE D'OUVRIR UN FICHIER INEXISTANT ?



```
with open('../data/file.txt') as f:  
    file = f.read()  
  
print(file)
```

course/main.py



```
Traceback (most recent call last):  
--  
    with open('../data/file.txt') as f  
FileNotFoundError: [Errno 2] No su...
```

SORTIE CONSOLE

Lorsque le chemin vers le fichier spécifié ne correspond à aucun fichier existant alors la fonction open lève l'exception FileNotFoundError.



# FICHER NON TROUVÉ

## GESTION DE L'EXCEPTION



```
try:
    with open('../data/file.txt') as f:
        file = f.read()
except FileNotFoundError:
    file = None

print(file)
```

course/main.py



None

SORTIE CONSOLE

Afin de gérer l'exception du fichier non trouvé, il est possible d'englober le code de lecture du fichier dans un bloc try/except.

Dans l'exemple ci-contre, si le fichier n'existe pas alors la variable file sera égale à None.

---

# ÉCRIRE DANS UN FICHER TEXTE

OCEANS.TXT - MODE 'w'



```
oceans = [  
    "Pacific",  
    "Atlantic",  
    "Indian",  
    "Southern",  
    "Arctic"  
]  
  
with open("oceans.txt", "w") as f:  
    for ocean in oceans:  
        f.write(ocean)  
        f.write("\n")  
  
    # Autre possibilité  
    # print(ocean, file=f)
```

course/main.py

```
Pacific  
Atlantic  
Indian  
Southern  
Arctic
```

OCEANS.TXT

Dans l'exemple ci-contre, la méthode `write` permet d'écrire dans un fichier. Elle prend en paramètre une chaîne de caractère et `"\n"` représente un saut de ligne.



Attention au mode écriture `'w'`, celui-ci écrase le contenu déjà existant et si le fichier n'existe pas, il sera alors automatiquement créé.

# ÉCRIRE DANS UN FICHIER TEXTE

OCEANS.TXT - MODE 'a'

AVANT

Pacific  
Atlantic  
Indian

OCEANS.TXT



```
oceans = [  
    "Southern",  
    "Arctic"  
]  
  
with open("oceans.txt", "a") as f:  
    for ocean in oceans:  
        f.write(ocean)  
        f.write("\n")  
  
# Autre possibilité  
# print(ocean, file=f)
```

course/main.py

APRÈS

Pacific  
Atlantic  
Indian  
Southern  
Arctic

OCEANS.TXT

Le mode 'a' pour "append" en anglais, ce mode ajoute le texte en fin de fichier, il ne présente aucun risque de perte de données.

---





**ALLER PLUS LOIN**

---

# ALLER PLUS LOIN

CONTINUER D'APPRENDRE

Modules et packages en Python

La POO en Python

**django**

Le développement Web avec Django



**MERCI DE VOTRE ATTENTION**