

Programmer avec Python

Table des matières

Historique.....	6
À quoi peut servir Python ?.....	6
Caractéristiques de Python.....	6
Un langage interprété.....	6
Bibliothèque intégrés et externes.....	7
Intégration de composants.....	7
Internet des objets.....	7
Version de Python.....	7
Installation de Python.....	7
Procédure pour Windows.....	7
Procédure pour Linux.....	8
Un premier exemple.....	8
Les éléments de base de Python.....	9
Les commentaires.....	9
Les variables.....	9
Les opérations arithmétiques.....	11
Entrées et sorties.....	11
Entrée au clavier.....	11
Sortie à l'écran.....	11
Méthode conseillée pour la sortie à l'écran.....	11
Transformez dans un autre type.....	13
Les alternatives.....	13
Les boucles.....	14
Boucle while.....	14
Boucle for.....	15
Boucle for pour compter.....	15
Boucle pour parcourir les éléments d'une liste.....	15
Structures de données.....	16
Les listes.....	16
Création d'une liste en Python.....	16
Exemple.....	16
Accès aux éléments d'une liste.....	16
Exemple.....	16
Changer la valeur d'un élément de la liste.....	16
Exemple.....	16
Parcourir les éléments d'une liste Python.....	17
Exemple.....	17
Longueur d'une liste Python.....	17
Exemple.....	17

Ajouter ou supprimer des éléments à la liste.....	17
Ajouter un un élément à une liste Python.....	17
Exemple.....	17
Exemple.....	17
Retirer un élément d'une liste Python.....	18
Exemple.....	18
Exemple.....	18
Exemple.....	18
Exemple.....	18
Exemple.....	19
Les différente méthodes destinées aux listes Python.....	19
Les tuples.....	20
Création d'un Tuple et syntaxe de base.....	20
Accès aux Éléments.....	21
Slicing.....	21
Opérations Courantes sur les Tuples.....	21
Longueur du Tuple.....	21
Parcourir les éléments d'un tuple.....	21
Affectation Multiple ou déballage (Unpacking en anglais).....	22
Pourquoi utiliser les Tuples ?.....	22
Les dictionnaires.....	23
Définir un dictionnaire en Python.....	23
Syntaxe :.....	23
Exemple : Annuaire téléphonique.....	23
Parcourir les valeurs et les clés d'un dictionnaire Python.....	23
Exemple : parcourt des valeurs d'un dictionnaire.....	23
Exemple : parcourt des clés d'un dictionnaire.....	24
Exemple: parcourt des clés et des valeurs.....	24
Mettre à jour, ajouter ou supprimer des éléments d'un dictionnaire.....	24
Mettre à jour un élément du dictionnaire.....	24
Exemple: gestionnaire d'un stock.....	24
Ajouter un élément au dictionnaire.....	25
Exemple: Ajouter un élément au stock.....	25
Supprimer un élément du dictionnaire.....	25
Exemple: suppression d'un élément du dictionnaire.....	25
Exemple: Suppression du dernier élément.....	25
Vider un dictionnaire.....	26
Exemple: vider un dictionnaire.....	26
Comment faire pour chercher si une clé est présente dans un dictionnaire ?.....	26
Sommaire des méthodes associées à un dictionnaire.....	26
Les fonctions en Python.....	28
Définition d'une fonction - def.....	28
Fonction sans paramètres.....	28
Fonction avec paramètres.....	29
Utilisation d'une variable comme argument.....	30
Fonction avec plusieurs paramètres.....	30
Valeurs par défaut pour les paramètres.....	31

Travailler avec des fichiers sous Python.....	32
Mode d'ouverture d'un fichier.....	32
Ouverture et lecture d'un fichier.....	32
Lecture totale avec la méthode read().....	32
Exemple: ouverture et lecture d'un fichier existant.....	32
Lecture partielle avec la méthode read().....	33
Exemple. lecture des 20 premiers caractères.....	33
Exemple. lecture d'un fichier avec la boucle while.....	33
Lecture ligne par ligne avec les méthodes readline() et readlines().....	33
La méthode readline().....	33
Exemple. lecture du fichier ligne par ligne.....	33
Exemple. lecture de toutes les lignes avec readline().....	34
La méthode readlines() - Méthode suggérée.....	34
Exemple. lecture des lignes du fichier avec readlines().....	34
Exemple. lecture des lignes à l'aide de la boucle for.....	34
Lecture d'un fichier à une position précise avec la méthode readlines().....	35
Exemple. lecture d'un fichier depuis le caractère 10 jusqu'au caractère 20 de la troisième ligne.....	35
Méthode suggérée pour ouvrir un fichier (with ... as).....	35
Exemple : Ouvrir un fichier « myfile.txt » en lecture seulement et faire afficher chaque ligne à l'écran.....	35
Ouverture et écriture dans un fichier existant.....	35
Exemple: ouvrir un fichier et y ajouter du contenu:.....	36
Exemple: ouvrir le fichier « myFile.txt » avec écrasement du contenu existant:.....	36
Lecture de fichier CSV.....	36
Structure des fichiers CSV.....	36
Utilisation des fichiers CSV et module csv avec Python.....	37
Lecture d'un fichier CSV.....	37
Exemple avec csv.DictReader.....	37
Écriture de fichier CSV.....	38
Exemple de csv.DictWriter (Écriture).....	38
Création des fichiers.....	39
Exemple. Création d'un fichier nommé « myFile.txt »:.....	39
Ajouter des lignes à un fichier en Python avec la méthode writelines().....	39
Exemple. ajouter une liste des lignes à un fichier.....	39
Les modules.....	40
Qu'est ce qu'un module en Python ?.....	40
Types de modules en Python.....	40
Les modules natifs.....	40
Exemple :.....	40
Les modules externes (ou tiers).....	41
Les modules personnalisés.....	41
Importation de modules.....	41
Importation du module au complet : import module.....	41
Exemple :.....	41
Importation avec alias : import module as alias.....	41
Exemple :.....	41
Importation partielle : from module import élément.....	42

Importation de tout le contenu :.....	42
Importation conditionnelle.....	42
Importation depuis un package.....	43
Exemple :.....	43
Installation d'un module.....	43
Installation dans un environnement virtuel.....	44
Sous Windows Sous Linux :.....	44
Remarque :.....	44
Vous pouvez facilement savoir si vous êtes dans votre environnement virtuel ou non. L'invite de commande commencera par (venv).....	44
Installer un module dans l'environnement virtuel.....	44
Désactiver l'environnement virtuel.....	44
Installation d'un module dans le système.....	45
Module relié à la gestion et à l'administration du système.....	46
Le module OS (plus d'information à : https://pythongeeeks.org/python-os-module/).....	46
getcwd().....	46
chdir().....	46
listdir().....	47
mkdir() et makedirs().....	47
chmod().....	47
chown().....	47
remove() et removedirs().....	47
Le module psutil – Gestion des processus et des ressources systèmes.....	48
Pour plus d'information : https://psutil.readthedocs.io/en/latest/	48
Installation (dans un environnement virtuel).....	48
Quelques exemples (Plus d'information à https://psutil.readthedocs.io/en/latest/).....	48
Informations sur le CPU.....	48
Exemple :.....	48
Mémoire (RAM et swap).....	48
Exemple :.....	49
Gestion des processus.....	49
Lister tous les processus :.....	49
Accéder à un processus précis :.....	49
Terminer un processus :.....	49
Le module netifaces.....	50
Installation (dans un environnement virtuel).....	50
Installation (dans le système localement).....	50
Quelques exemples (plus de détail à : https://pypi.org/project/netifaces/).....	50
Lister les interfaces réseau.....	50
Obtenir les adresses IP d'une interface.....	50
Obtenir les passerelles par défaut.....	51
Créer son propre module.....	51
Étape 1 : Créer le fichier du module.....	51
Étape 2 : Utiliser le module dans un autre script.....	52
Créer un package.....	52

Historique

- Créé en 1991.
- Guido van Rossum est le fondateur.
- Langage de programmation interprété.
- Nom en hommage à « Monty Python » supposément



Guido avec sa Heineken :)

À quoi peut servir Python ?

Python est utilisé dans plusieurs domaines de l'informatique. Ce langage à la fois relativement simple à apprendre et riche en possibilités permet de développer dans les domaines aussi variés que :

- les jeux vidéos,
- l'intelligence artificielle,
- apprentissage automatique (machine learning),
- multimédia,
- interfaces graphiques,
- scripts,
- réseautique, etc...

Caractéristiques de Python

Un langage interprété

Python est un langage interprété, c'est-à-dire que les instructions que vous tapez sont traduites en langage machine au fur et à mesure de leur lecture à l'instar d'un langage compilé qui traduit le code en langage machine en une étape à part entière.

Ce langage fonctionne sur la plupart des systèmes d'exploitation sans aucune modification. Autrement dit, vous codez un programme en Python sur votre système Windows et ce programme roulera sans problème sur votre machine Linux.

Bibliothèque intégrés et externes

Python possède un grand nombre de bibliothèques pré-construites et portables. Vous pouvez les charger à tout moment pour utiliser les fonctionnalités souhaitées.

Intégration de composants

Python dispose de plusieurs moyens pour prendre en charge la communication inter-applications. Il permet des mécanismes tels que le chargement de bibliothèques C et C ++ ou inversement, l'intégration à des composants Java et DotNET, la communication à l'aide de COM / Silverlight et l'interfaçage avec des périphériques USB via des ports série. Il peut même échanger des données sur des réseaux utilisant des protocoles tels que SOAP, XML-RPC et CORBA.

Internet des objets

Python s'est fait une place de choix comme langage de programmation dans le domaine de l'Internet des Objets. On le retrouve dans plusieurs applications qui utilisent des systèmes embarqués, des capteurs ou tout autre dispositifs de prises d'information.

Version de Python

En date de l'écriture de ce document, la dernière version de Python est la version 3.7.4 disponible sur le site de Python à l'adresse <http://www.python.org>.

Devrais-je utiliser Python 2 ou Python 3 ?

Vous devriez utiliser les versions 3 de la famille Python. Les versions de la famille 2 ne seront plus officiellement supportées à partir de Janvier 2020.

Installation de Python

Procédure pour Windows

- Rendez-vous à l'adresse <http://www.python.org>
- Cliquez sur Download et Windows.

- Choisissez la dernière version pour Windows en 32 ou 64 bits

Procédure pour Linux

Python est pré-installé sur la plupart des distributions Linux. Cependant, il est possible que vous n'ayez pas la dernière version en date. Pour le vérifier, tapez dans un terminal la commande `python -V`. Cette commande vous renvoie la version de Python actuellement installée sur votre système.

Un premier exemple

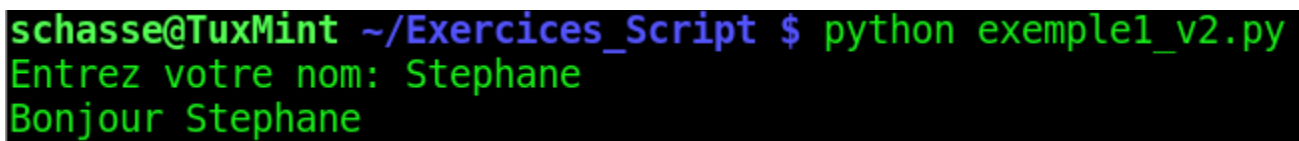
Entrez les lignes de code ci-dessous :

```
nom = input("Entrez votre nom: ")  
  
print("Bonjour {0}".format(nom))  
#ou semblable à C#  
print(f"Bonjour {nom}")
```

Sauvegardez le fichier en le nommant « exemple1.py » .

Exécutez en tapant : `python3 exemple1.py`

Vous obtenez :

A terminal window with a black background and green text. The prompt is 'schasse@TuxMint ~/Exercices_Script \$'. The command 'python exemple1_v2.py' has been entered. The output shows 'Entrez votre nom: Stephane' followed by 'Bonjour Stephane' on the next line.

```
schasse@TuxMint ~/Exercices_Script $ python exemple1_v2.py  
Entrez votre nom: Stephane  
Bonjour Stephane
```

Vous remarquez donc que :

- Pour entrer une donnée au clavier, on utilise la fonction « input » en donnant la phrase que l'on veut demander à l'utilisateur.

- La fonction « input » retourne une chaîne de caractère qui correspond à la donnée entrée au clavier.
- La fonction « print » fait afficher les données à l'écran.
- Les balises {0} sont remplacées par la variable qui est envoyée dans la fonction « format ».
- Dans les nouvelles versions, on peut directement faire afficher la variable en la balisant de { et } . N'oubliez pas le caractère « f » devant le premier guillemet pour utiliser cette possibilité.

Les éléments de base de Python

Nous allons dans cette section discuter des principales structures de base de programmation avec le langage Python. Nous verrons donc les variables et leur assignation, l'entrée et la sortie à l'écran ainsi que les alternatives, les boucles et les fonctions.

Les commentaires

Les commentaires dans un programme python peuvent être réalisés avec le caractère « # ».

Exemple :

```
#Voici mon premier programme
# Auteur : Stephane
# Date : Aujourd'hui

print("Bonjour! ") # Commentaire sur la même ligne
```

Les variables

Python est un langage dynamiquement typé. Ceci veut dire que python sélectionne le meilleur type pour représenter votre donnée selon ce que vous avez assigné à la variable.

Exemples :

```
nom = input("Entrez votre nom : ") # nom sera str. Le type « str » est ce qui représente une chaîne de caractère
```

v1 = 10 # Cette variable est entière

v2 = 11.97 # Cette variable sera « float »

v3 = True # Cette variable sera « bool ». Remarquez True pour la valeur vraie et False pour le faux.

#Faites attention aux majuscules!

v1 = 5.5 # La variable est de type « float »

v1 = "Bonjour" # La variable est maintenant de type « str »

Rappelez-vous :

Un nom de variable doit commencer par une lettre ou le caractère underscore.

Un nom de variable ne peut pas commencer par un nombre.

Un nom de variable ne peut contenir que des caractères et underscores (AZ, 0-9 et _).

Les noms de variables sont sensibles à la casse (age, Age sont deux variables différentes)

Les opérations arithmétiques

Comme dans tout langage de programmation, les opérations arithmétiques sont réalisées avec les opérateurs usuels : addition (+), soustraction (-), multiplication(*), division(/), modulo ainsi que 2 autres opérateurs. Il s'agit de la division entière (//) et de l'exposant (**).

addition	+	Ex. : print(10 + 10) donne 20
Soustraction	-	Ex. : v1 =0.1 print(1 - v1) donne 0.9
multiplication	*	Ex. : print(0.1 * 0.75) donne 0.07500000000000001
division	/	Ex. : print(1000 / 100) donne 10.0
Modulo (reste de la division entière)	%	Ex. : print(5 % 2) donne 1
Double division	//	Ex. : print (5 // 2) donne 2. Garde la portion entière seulement.
Exposant	**	Ex. : print (5 ** 2) donne 25

Entrées et sorties

Comme vous l'avez déjà expérimenté précédemment, l'entrée au clavier se fait grâce à la fonction « input ». Vous devez cependant transformer la valeur lue dans un autre type si ce n'est pas un string que vous désirez.

Entrée au clavier

L'entrée d'une donnée au clavier se fait avec la fonction « input ».

Exemples :

```
v = input("Entrez une valeur")
```

Sortie à l'écran

L'affichage à l'écran se fait avec la fonction « print ».

Exemples :

```
v = input("Entrez une valeur")  
print(v)
```

Méthode conseillée pour la sortie à l'écran

Depuis la version 3.x de python, on vous conseille maintenant d'utiliser l'interpolation de chaîne.

On utilise le caractère « f » dans l'instruction print et on entoure les variables des {}.

Exemple :

```
moyenneTemp = 23.2
```

```
maximumTemp = 26.4
```

```
print( f"La température maximum pour le mois a atteint : {maximumTemp} et la moyenne du mois a atteint : {moyenneTemp}")
```

On peut même faire afficher le nombre de virgule après le point décimal avec la syntaxe suivante :

```
print( f"La température maximum pour le mois a atteint : {maximumTemp :.2f} et la moyenne du mois a atteint : {moyenneTemp :.2f}")
```

.2 signifie le nombre de décimale après le point.

f signifie que c'est un nombre à virgule.

On peut même faire afficher dans un espacement déterminé. Très utile pour faire des colonnes alignées.

Exemple :

Maximum du mois	Moyenne du mois
5.30	-9.30
36.20	25.10

Code :

```
print("Maximum du mois" + " "*10 + "Moyenne du mois")

print(f"{maximumJanvierTemp:>10,.2f} {moyenneJanvierTemp:>20,.2f}")
print(f"{maximumJuilletTemp:>10,.2f} {moyenneJuilletTemp:>20,.2f}")
```

Le caractère > signifie que le contenu sera justifié à droite de la colonne.

Le nombre 10 ou 20, signifie la largeur de la colonne.

Transformez dans un autre type

Vous devrez convertir la chaîne lue vers le type que vous désirez. Il s'agit alors d'utiliser les instructions « int », « float » ou « str » pour les différentes conversions.

Exemples :

De str à float	<pre>montant = float(input("Entrez le montant : ")) montant = float("12.99")</pre>
De str à int	<pre>age = int(input("Entrez votre age :")) c = "12" # Ceci est une chaîne c = int(c) # La variable devient maintenant entière</pre>
De int ou float à str	<pre>v1 = 12 v2 = 13.79 v1 = str(v1) # v1 est maintenant une chaîne "12" v2 = str(v2) # v2 est maintenant une chaîne "13.79"</pre>

Les alternatives

Comme dans plusieurs langages de programmation, l'instruction « if » permet de créer une alternative en lui indiquant la condition à évaluer.

if condition :

instruction

instruction

elif condition :

instruction

instruction

else :

instruction

instruction

Remarques :

- Le elif est une contraction d'un équivalent else :

if condition :

- Le bloc elif n'est pas obligatoire. Par contre, si il est présent, il doit avoir une condition
- La partie else n'est pas obligatoire.

INDENTATION

Un mot important sur les indentations. Vous savez, les espaces que l'on met au début d'une ligne pour que la lecture du code soit meilleure. Hey bien, en python, les indentations ne sont pas juste pour faire beau mais sont OBLIGATOIRES.

Ainsi les 2 if suivants ne sont pas équivalents :

<pre>x = 2 if x % 2 == 0 : print("Votre nombre est pair") x = int(input("Entrez un autre nombre")) else : print("Votre nombre est impair")</pre>	<pre>x = 2 if x % 2 == 0 : print("Votre nombre est pair") x = int(input("Entrez un autre nombre")) else : print("Votre nombre est impair")</pre>
--	--

Manque l'indentation ici.
Python va se plaindre

Dans le cas de l'exemple de droite, vous obtiendrez le résultat suivant :

```
File "exemple1_if.py", line 5
    else:
    ^
SyntaxError: invalid syntax
```

Les boucles

Python contient une panoplie de boucles pour pouvoir répéter des instructions ou parcourir une liste d'éléments.

Boucle while

Les boucles de type « while » permettent de répéter un certain nombre d'instructions tant et aussi longtemps que la condition sera vraie. Encore une fois, on devra respecter l'indentation pour les instructions faisant parti de la boucle.

Exemple :

```
Nombre = int(input("Entrez votre nombre: "))
Compteur = 1
while Compteur <= Nombre:
    print(Compteur)
    Compteur += 1
```

Boucle for

La boucle « for » permet de répéter un nombre d'instructions en fonction d'un nombre déterminé ou d'une condition particulière. La boucle for est également très employée pour parcourir des collections. Nous verrons cet aspect plus tard.

Boucle for pour compter

Voici un exemple où nous voulons faire afficher les nombres de 1 à 9 avec la boucle for. La fonction range(x) permet de créer une liste de nombre de 1 à x -1. Donc range(10) provoque la séquence des nombres de 1 à 9.

```
for i in range(10):
```

```
    print("i = {0}".format(i))
```

ou depuis la version 3.7 +

```
    print(f"i = {i}")
```

Boucle pour parcourir les éléments d'une liste

C'est dans ce contexte que la boucle for est plus utile en python. Voici un exemple :

```
liste = [10,20,30,40,50]
```

```
for valeur in liste :
```

```
    print(valeur)
```

On peut même la parcourir en ordre inverse !

```
liste = [10,20,30,40,50]
```

```
liste.reverse()
```

```
for valeur in liste:
```

```
    print(valeur)
```

Structures de données

Les listes

Création d'une liste en Python

Une liste en Python est un type de données ordonnée et modifiable qui fait partie des collections . En Python, les listes sont écrites entre crochets.

Exemple

```
#Création d'une liste
myList = ["Python", "Java", "Javascript"]
// Affichage de la liste
print (myList)
```

Accès aux éléments d'une liste.

Vous accédez aux éléments d'une liste Python, en vous référant au numéro d'index:

Exemple

```
Imprimer le 3ème élément de la liste:
myList = ["Python", "Java", "PHP"]
print(myList[2]) # Affiche 'PHP'
```

Changer la valeur d'un élément de la liste

Pour modifier la valeur d'un élément spécifique, reportez-vous au numéro d'index:

Exemple

Changer le deuxième élément:

```
myList = ["Python", "Java", "Javascript"]
myList[1]="Oracle"
print(myList) #Affiche : ['Python','Oracle','Javascript']
```

Parcourir les éléments d'une liste Python

Vous pouvez parcourir les éléments d'une liste en Python, en utilisant une boucle for:

Exemple

```
#Imprimer tous les éléments de la liste, un par un:
myList = ["Python", "Java", "PHP"]
for x in myList:
    print(x) # Affiche tous les éléments de la liste un par un.
```

Longueur d'une liste Python

Pour déterminer le nombre d'éléments d'une liste, utilisez la méthode len():

Exemple

```
#Imprimer le nombre d'éléments de la liste:
myList = ["Python", "Java", "PHP"]
print ("La longueur de ma liste est {}".format(len (myList))) #Affiche : La
longueur de ma liste est 3
```

Ajouter ou supprimer des éléments à la liste

Ajouter un un élément à une liste Python

Pour ajouter un élément à la fin de la liste, utilisez la méthode append():

Exemple

```
#ajouter un élément à la liste avec la méthode append() :
myList = ["Python", "Java", "PHP"]
myList.append ("Oracle")
print (myList) #Affiche : ["Python", "Java", "PHP", "Oracle"]
```

Pour ajouter un élément à l'index spécifié, utilisez la méthode insert():

Exemple

```
#Insérer un élément en deuxième position:
myList = ["Python", "Java", "PHP"]
myList.insert (1, "C++")
print (myList) # Affiche : ["Python", "C++", "Java", "PHP"]
```

Retirer un élément d'une liste Python

Il existe plusieurs méthodes pour supprimer des éléments d'une liste:

- La méthode « `remove()` » supprime un élément spécifié.
- La méthode « `pop()` » supprime un élément en spécifiant son index (ou le dernier élément si aucun index n'est spécifié)
- Le mot clé « `del` » supprime l'élément à l'index spécifié(`del` permet également de supprimer complètement la liste)
- La méthode « `clear()` » vide la liste.

Exemple

```
#suppression d'un élément spécifié avec la méthode remove()
myList = ["Python", "Java", "PHP"]
myList.remove("Java")
print(myList) #Affiche: ["Python", "PHP"]
```

Exemple

```
#Suppression d'un élément d'index spécifié avec la méthode pop()
myList = ["Python", "Java", "PHP"]
myList.pop(0)
print(myList) #Affiche : ["Java", "PHP"]
```

Exemple

```
#suppression d'élément à un index spécifié avec la méthode del :
myList = ["Python", "Java", "PHP"]
del myList[1]
print(myList) #Affiche : ["Python", "PHP"]
```

Le mot clé « `del` » peut également supprimer complètement la liste:

Exemple

```
myList = ["Python", "Java", "PHP"]
del myList
print(myList) #cela causera une erreur car "myList" n'existe plus.
```

Exemple

```
myList = ["Python", "Java", "PHP"]
myList.clear ()
print (myList) #Affichera uniquement [] . C'est-à-dire, une liste qui existe
mais qui est vide!
```

Les différentes méthodes destinées aux listes Python

Python a un ensemble de méthodes intégrées que vous pouvez utiliser sur des listes.

append()	Ajoute un élément à la fin de la liste
clear()	Supprime tous les éléments de la liste
copy()	Retourne une copie de la liste
count()	Retourne le nombre d'éléments avec la valeur spécifiée
extend()	Ajoute les éléments d'une liste (ou de tout élément itérable) à la fin de la liste actuelle
index()	Retourne l'index du premier élément avec la valeur spécifiée.
insert()	Ajoute un élément à la position spécifiée
pop()	Supprime l'élément à la position spécifiée
remove()	Supprime l'élément avec la valeur spécifiée
reverse()	Inverse l'ordre de la liste
sort()	Trie la liste

Les tuples

Les tuples ressemblent aux listes, mais on ne peut pas les modifier une fois qu'ils ont été créés.

On dit qu'un tuple est immuable.

Un **tuple** est une collection ordonnée et **immuable** (non modifiable) d'éléments en Python. Ils sont similaires aux listes, mais leur principale distinction est qu'une fois qu'un tuple est créé, vous ne pouvez pas modifier, ajouter ou supprimer ses éléments.

Les tuples sont :

- **Ordonné** : Les éléments ont un ordre défini, qui ne change pas.
- **Immuable** : Une fois créé, le contenu d'un tuple ne peut pas être modifié. C'est la différence majeure avec les **listes**.
- **Indexé** : Les éléments sont accessibles par leur position (index), commençant à **0**.
- **Hétérogène** : Peut contenir des éléments de différents types de données.

Création d'un Tuple et syntaxe de base

Les tuples sont définis en plaçant les éléments entre parenthèses (), séparés par des virgules.

À sa plus simple expression, une variable tuples peut être créée en utilisant les parenthèses

```
t = () # Une variable tuple vide.
```

Tuple d'entiers	<code>nombre = (1, 2, 3, 4)</code>
Tuple de chaînes de caractères	<code>couleurs = ('rouge', 'vert', 'bleu')</code>
Tuple hétérogène (mélange de types)	<code>melange = ('texte', 10, True, 3.14)</code>

Remarques

Tuple à un seul élément

Pour créer un tuple contenant un seul élément, vous **devez** inclure une virgule après l'élément, sinon Python l'interprétera comme une simple valeur entre parenthèses.

C'est un tuple

```
tuple_un_element = (5,)
```

Ceci est juste un entier

```
pas_un_tuple = (5)
```

Accès aux Éléments

On a accès aux éléments d'un tuple en utilisant l'index de l'élément tout comme on le fait avec une liste. Ce qui est pratique et qui ne demande pas une syntaxe supplémentaire.

Exemple :

```
fruits = ("pomme", "banane", "cerise", "orange")
```

```
# Accéder au premier élément (index 0)
premier = fruits[0] # Résultat: 'pomme'
```

```
# Accéder au dernier élément (index -1)
dernier = fruits[-1] # Résultat: 'orange'
```

Slicing

Cette façon d'accéder à plusieurs éléments n'est pas unique aux tuples. On retrouve aussi cette syntaxe dans les listes et les dictionnaires.

```
# Récupérer les éléments de l'index 1 (inclus) à l'index 3 (exclu)
segment = fruits[1:3] # Résultat: ('banane', 'cerise')
```

```
# Récupérer du début jusqu'à l'index 2 (exclu)
debut = fruits[:2] # Résultat: ('pomme', 'banane')
```

```
# Récupérer de l'index 2 (inclus) à la fin
fin = fruits[2:] # Résultat: ('cerise', 'orange')
```

Opérations Courantes sur les Tuples

Longueur du Tuple

Utilisez la fonction `len()` pour connaître le nombre d'éléments.

```
len(fruits) # Résultat: 4
```

Parcourir les éléments d'un tuple

Vous pouvez parcourir les éléments d'un tuple avec une boucle `for` comme on le ferait pour une liste.

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

Affectation Multiple ou déballage (Unpacking en anglais)

C'est une fonctionnalité très utile où vous attribuez les éléments d'un tuple à plusieurs variables en une seule ligne.

```
point2D = (10, 20)
x, y = point2D

print(x) # Résultat: 10
print(y) # Résultat: 20
```

Pourquoi utiliser les Tuples ?

1. **Immuabilité** : Ils garantissent que les données ne seront pas accidentellement modifiées. Ceci les rend plus sûrs dans certains contextes et plus rapides que les listes.
2. **Sécurité** : Les tuples sont souvent utilisés pour les données qui ne devraient pas changer (exemples : coordonnées géographiques, dates, clés de dictionnaire).
3. **Performances** : Ils sont généralement plus rapides à itérer que les listes.
4. **Clés de Dictionnaire** : Seuls les types immuables (comme les tuples) peuvent être utilisés comme clés dans un dictionnaire.

Les dictionnaires

Définir un dictionnaire en Python

Un autre type de donnée très utile, natif dans Python, est le *dictionnaire*. Ces dictionnaires sont parfois présents dans d'autres langages sous le nom de "mémoires associatives" ou de "tableaux associatifs". À la différence des séquences, qui sont indexées par des nombres, les dictionnaires sont indexés par des *clés*, qui peuvent être de n'importe quel type immuable ; les chaînes de caractères et les nombres peuvent toujours être des clés.

On peut définir un dictionnaire en entourant des accolades { } une liste de paires clé-valeur séparées par des virgules.

Syntaxe :

```
dic = {key1: valeur1, key2: valeur2, key3: valeur3, ...}
```

Pour accéder à une valeur à partir du dictionnaire, on utilise le nom du dictionnaire suivi de la clé correspondante entre crochets:

```
dic = {key1: valeur1, key2: valeur2, key3: valeur3, ...}  
print(dic[key1]) # affiche valeur1
```

Exemple : Annuaire téléphonique

```
phoneBook = {"Majid": "0556683531", "Tomas": "0537773332", "Bernard": "0668793338",  
             "Hafid": "066445566"}  
print(phoneBook["Majid"]) # affiche 0556683531
```

Parcourir les valeurs et les clés d'un dictionnaire Python

Un dictionnaire en Python est doté d'une méthode nommée `values()` qui permet de parcourir ses valeurs, et d'une autre nommée `keys()` permettant de parcourir ses clés.

Exemple : parcourt des valeurs d'un dictionnaire

```
phoneBook={"Majid": "0556633558", "Tomas": "0587958414", "Bernard": "0669584758"}  
for valeur in phoneBook.values():  
    print(valeur)
```

Exemple : parcourt des clés d'un dictionnaire

```
phoneBook={"Majid":"0556633558","Tomas":"0587958414","Bernard":"0669584758"}
for key in phoneBook.keys():
    print(key)
```

Remarque :

On peut aussi parcourir les clés et les valeurs en même temps en utilisant la méthode items()

Exemple: parcourt des clés et des valeurs

```
phoneBook={"Majid":"0556633558","Tomas":"0587958414","Bernard":"0669584758"}
for key , valeur in phoneBook.items():
    print(key, valeur)
```

Mettre à jour, ajouter ou supprimer des éléments d'un dictionnaire

Mettre à jour un élément du dictionnaire

On peut mettre à jour un élément du dictionnaire directement en affectant une valeur à une clé:

Exemple: gestionnaire d'un stock

```
stock={"Laptop":15, "Imprimante":35,"Tablette":27}

#modification de la valeur associée à la clé "Imprimante"
stock["Imprimante"]=42
print(stock)
# affiche : {'Laptop': 15, 'Imprimante': 42, 'Tablette': 27}
```

Ajouter un élément au dictionnaire

Dans le cas d'une clé inexistante, la même méthode ci-dessus, permet d'ajouter des éléments au dictionnaire:

Exemple: Ajouter un élément au stock

```
stock={"Laptop":15, "Imprimante":35,"Tablette":27}

# Ajout de l'élément "Ipad":18
stock["Ipad"]=18
print(stock)
# affiche : {'Laptop': 15, 'Imprimante': 35, 'Tablette': 27, 'Ipad':18}
```

Supprimer un élément du dictionnaire

On peut supprimer un élément du dictionnaire en indiquant sa clé dans la méthode pop()

Exemple: suppression d'un élément du dictionnaire

```
stock={'Laptop': 15, 'Imprimante': 35, 'Tablette': 27, 'Ipad':22}

# Suppression de l'élément "Imprimante": 35
stock.pop("Imprimante")
print(stock)
# affiche : {'Laptop': 15, 'Tablette': 27, 'Ipad':22}
```

Un dictionnaire est doté d'une autre méthode : popitem() qui permet de supprimer le dernier élément et le retourne sous forme d'un tuple.

Exemple: Suppression du dernier élément

```
stock={'Laptop': 15, 'Imprimante': 35, 'Tablette': 27, 'Ipad':22}

# Suppression du dernier élément
stock.popitem()
print(stock)
# affiche : {'Laptop': 15, 'Imprimante': 35, 'Tablette': 27}
```

Vider un dictionnaire

Un dictionnaire Python peut être vider à l'aide de la méthode `clear()`

Exemple: vider un dictionnaire

```
stock={'Laptop': 15, 'Imprimante': 35, 'Tablette': 27, 'Ipad':22}

# vider le dictionnaire
stock.clear()
print(stock)
# affiche un dictionnaire vide : {}
```

Comment faire pour chercher si une clé est présente dans un dictionnaire ?

Vous pouvez utiliser le mot-clé « `in` » pour vérifier la présence d'une clé que vous cherchez:

```
stock={'Laptop': 15, 'Imprimante': 35, 'Tablette': 27, 'Ipad':22}

if "Tablette" in stock:
    print("Tablette est presente dans le dictionnaire")
else:
    print("Je n'ai pas trouvé le mot recherché")
```

Attention

Le mot recherché est une clé ou une valeur et doit être écrit de la même façon.

Si on veut rechercher autrement, comme par exemple, les noms qui commenceraient par I, il faut alors utiliser les expressions régulières.

Sommaire des méthodes associées à un dictionnaire

- `clear()` : supprime tous les éléments du dictionnaire.
- `copy()` : retourne une copie superficielle du dictionnaire.
- `fromkeys(seq [, v])` : retourne un nouveau dictionnaire avec les clés de `seq` et une valeur égale à `v` (la valeur par défaut est `None`).
- `get(key [, d])` : retourne la valeur de `key`. Si la clé ne quitte pas, retourne `d` (la valeur par défaut est `Aucune`).

- `items()` : retourne une nouvelle vue des éléments du dictionnaire (clé, valeur).
- `keys()` : retourne une nouvelle vue des clés du dictionnaire.
- `pop(key [, d])` : supprime l'élément avec `key` et renvoie sa valeur ou `d` si `key` n'est pas trouvé. Si `d` n'est pas fourni et que la clé est introuvable, soulève `KeyError`.
- `popitem()` : supprimer et retourner un élément arbitraire (clé, valeur). Lève `KeyError` si le dictionnaire est vide.
- `setdefault(key [, d])` : si `key` est dans le dictionnaire, retourne sa valeur. Sinon, insérez la clé avec la valeur `d` et renvoyez `d` (la valeur par défaut est `Aucune`).
- `update([other])` : met à jour le dictionnaire avec les paires clé / valeur des autres clés existantes.
- `values()` : retourne une nouvelle vue des valeurs du dictionnaire

Les fonctions en Python

Les fonctions en python ont la même utilité que dans les autres langages. Elles permettent de coder une série d'instructions qui peut être réutilisées dans le même programme ou dans d'autres programmes.

Définition d'une fonction - **def**

Syntaxe

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nom_fonction(liste de paramètres):  
    bloc d'instructions
```

Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots-clés réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, on utilise par convention des minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux classes).

Corps de la fonction

Comme les instructions **if**, **for** et **while**, l'instruction **def** est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un deux-points :, qui introduisent un bloc d'instructions qui est précisé grâce à l'indentation. Ce bloc d'instructions constitue le **corps de la fonction**.

Fonction sans paramètres

Exemple

```
def compteur3():  
    i = 0  
    while i < 3:  
        print(i)  
        i = i + 1  
  
print("bonjour")  
compteur3()  
compteur3()
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui compte jusqu'à 2. Notez bien les parenthèses, les deux-points, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Après la définition de la fonction, on trouve le programme principal qui débute par l'instruction `print("bonjour")`. Il y a ensuite au sein du programme principal, l'appel de la fonction grâce à `compteur3()`.

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons.

Fonction avec paramètres

Exemple

```
def compteur(stop):  
    i = 0  
    while i < stop:  
        print(i)  
        i = i + 1
```

```
compteur(4)  
compteur(2)
```

Affichage après exécution :

```
0  
1  
2  
3  
0  
1
```

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument.

Utilisation d'une variable comme argument

L'argument que nous utilisons dans l'appel d'une fonction peut être une variable.

Exemple

```
def compteur(stop):  
    i = 0  
    while i < stop:  
        print(i)  
        i = i + 1
```

```
a = 5  
compteur(a)
```

Affichage après exécution :

```
0  
  
1  
  
2  
  
3  
  
4
```

Fonction avec plusieurs paramètres

Exemple

La fonction suivante utilise trois paramètres : `start` qui contient la valeur de départ, `stop` la borne supérieure exclue comme dans l'exemple précédent et `step` le pas du compteur.

```
def compteur_avec_pas(debut, fin, pas):  
    i = debut  
    while i < fin:  
        print(i)  
        i = i + pas
```

```
compteur_complet(1, 7, 2)
```

Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus.

Exemples :

```
def calcultaxe(montant, taxe=5):  
    print("Le montant avec taxe s'élève à : {0} $".format(montant * taxe/100))  
  
calcultaxe(100)  
  
Le montant avec taxe s'élève à : 105 $  
  
calcultaxe(100, 10)  
  
Le montant avec taxe s'élève à : 110 $
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

Travailler avec des fichiers sous Python

En Python, il n'est pas nécessaire d'importer une bibliothèque externe pour lire et écrire des fichiers. Python fournit une fonction intégrée pour la création, l'écriture et la lecture de fichiers.

Mode d'ouverture d'un fichier

En langage Python, il n'est pas nécessaire d'importer une bibliothèque pour lire et écrire sur des fichiers. Il s'agit d'opérations gérées nativement par le langage. La première chose à faire est d'utiliser la fonction `open()` intégrée de Python pour obtenir un objet fichier (Python file object). La fonction `open()` ouvre un fichier d'une façon assez simple! Lorsque vous utilisez la fonction `open()`, elle renvoie un objet du type file object. Les objets file object, contiennent des méthodes et des attributs pouvant être utilisés pour collecter des informations sur le fichier que vous avez ouvert. Ils peuvent également être utilisés pour manipuler le dit fichier.

Un objet fichier crée par la méthode `open()`, est doté de certaines propriétés permettant de lire et écrire sur ce dernier. Sa syntaxe est:

```
f = open([nom du fichier], [mode ouverture])
```

[nom du fichier] est le nom du fichier qu'on souhaite ouvrir ou créer. Le mode d'ouverture comprend les paramètres suivants:

- Le mode **'r'** : ouverture d'un fichier existant en lecture seule,
- Le mode **'w'** : ouverture en écriture seule, écrasé s'il existe déjà et crée s'il n'existe pas,
- Le mode **'a'** : ouverture et écriture en fin du fichier avec conservation du contenu existant
- Le mode **'+'** : ouverture en lecture et écriture
- Le mode **'b'** : ouverture en mode binaire

Ouverture et lecture d'un fichier

Pour lire un fichier existant, plusieurs méthodes sont disponibles :

Lecture totale avec la méthode `read()`

La méthode `read()` permet de lire le contenu total ou partiel d'un fichier, après être ouvert avec la méthode `open()`.

Exemple: ouverture et lecture d'un fichier existant

```
f = open("myFile.txt", 'r')
contenu = f.read() # lecture du contenu
print(contenu) # impression du contenu
f.close() # fermeture du fichier
```

Lecture partielle avec la méthode read()

La méthode read() peut être également utilisée pour lire une partie du fichier seulement en indiquant le nombre de caractère à lire entre parenthèses :

Exemple. lecture des 20 premiers caractères

```
f = open("myFile.txt", 'r')
contenu = f.read(20) # lecture de 20 caractère du contenu du fichier
print(contenu) # impression du contenu
f.close() # fermeture du fichier
```

Remarque

Après exécution de la fonction read(n) (n = nombre de caractères à lire), le curseur se trouve à la position n+1, et donc si on exécute la fonction une 2ème fois, la lecture débutera depuis le (n+1)ème caractère.

Exemple. lecture d'un fichier avec la boucle while

```
f = open("myFile.txt", 'r')
sortie = False
s=""
while not sortie:
    c = f.read(1)
    if c == "":
        sortie = True
    s = s + c
print(s)
```

Lecture ligne par ligne avec les méthodes readline() et readlines()

La méthode readline()

La méthode readline() permet de lire un fichier ligne par ligne. Cette méthode pointe sur la première ligne lors de sa première exécution, ensuite sur la deuxième ligne lors de sa seconde exécution et ainsi à la n-ième exécution, elle pointe vers la n-ième ligne.

Exemple. lecture du fichier ligne par ligne

```
# -*- coding: utf-8 -*-
f = open("myFile.txt", 'r')
```

```
print(f.readline()) # affiche la ligne n°1
print(f.readline()) # affiche la ligne n°2
```

En combinant la méthode `readline()` avec la méthode `while()`, on peut lire la totalité des lignes d'un fichier :

Exemple. lecture de toutes les lignes avec `readline()`

```
f = open("myFile.txt", 'r')
sortie = False
s=""
while not sortie:
    ligne = f.readline()
    if (ligne == ""):
        sortie = True
    s = s + ligne
print(s) # impression de la totalité des lignes
```

La méthode `readlines()` - Méthode suggérée

La méthode `readlines()`, renvoie une liste dont les éléments sont les lignes du fichier

Exemple. lecture des lignes du fichier avec `readlines()`

```
f = open("myFile.txt", 'r')
content = f.readlines()
print(content[0]) # impression de la première ligne
print(content[1]) # impression de la deuxième ligne
```

Remarque: On peut aussi lire la totalité des lignes du fichier en appliquant la boucle `for`:

Exemple. lecture des lignes à l'aide de la boucle `for`

```
f = open("myFile.txt", 'r')
content = f.readlines()
for ligne in content:
    print(ligne)
```

Lecture d'un fichier à une position précise avec la méthode readlines()

La méthode readlines() nous permet aussi de lire un fichier à une position bien précise :

Exemple. lecture d'un fichier depuis le caractère 10 jusqu'au caractère 20 de la troisième ligne

```
f = open("myFile.txt", 'r')
content = f.readlines()[2] #récupération de la deuxième ligne
result = content[9:19] # extraction depuis le caractère position 10 jusqu'à 18
print (result)
```

Méthode suggérée pour ouvrir un fichier (with ... as)

L'instruction « with ... as ... » permet d'ouvrir un fichier, de travailler avec ce fichier dans le bloc d'instruction et de fermer le fichier automatiquement à la fin du bloc d'instruction.

Exemple : Ouvrir un fichier « myfile.txt » en lecture seulement et faire afficher chaque ligne à l'écran

```
With open("myfile.txt", "r") as fichier :
    for ligne in fichier.readlines() :
        print(ligne)
# ==> Fichier automatiquement fermé ici

→ Ici le fichier est automatiquement fermé (comme si vous aviez utilisé la commande
fichier.close() ).
```

Ouverture et écriture dans un fichier existant

Pour écrire dans un fichier existant, vous devez ajouter l'un des paramètres à la fonction open():

- « a » – Append – sera ajouté à la fin du fichier
- « w » – Write – écrasera tout contenu existant
- « r+ » Lecture et écriture sans écraser le contenu existant

On dira alors que le fichier est ouvert en mode écriture (write mode) Pour écrire dans fichier ouvert en mode écriture, on utilise la fonction write().

La syntaxe est:

```
file.write(contenu)
```

Exemple: ouvrir un fichier et y ajouter du contenu:

```
# ouverture avec conservation du contenu existant
f = open ("myFile.txt", "a")
f.write ("Voici un contenu qui va s'ajouter au fichier sans écraser le
contenu!")
f.close ()
# ouvrir et lire le fichier après l'ajout:
f = open ("myFile.txt", "r")
print (f.read())
```

Exemple: ouvrir le fichier « myFile.txt » avec écrasement du contenu existant:

```
# -*- coding: utf-8 -*-
# ouverture avec écrasement du contenu existant
f = open ("myFile.txt", "w")
f.write ("Désolé ! J'ai supprimé le contenu!")
f.close()
# ouvrir et lire le fichier après l'ajout:
f = open ("myFile.txt", "r")
print (f.read())
```

Lecture de fichier CSV

Les fichiers de type **CSV** (« Comma-Separated Values » en anglais), ou **valeurs séparées par des virgules** sont un format de fichier texte simple et couramment utilisé pour stocker des données sous forme de colonne. C'est un type de fichier fréquemment utilisé dans des applications de bureau comme les feuilles de calcul Excel par exemple.

Structure des fichiers CSV

Un fichier CSV est structuré de la manière suivante :

1. **Chaque ligne** du fichier correspond à un **enregistrement** ou une **ligne** de données (par exemple, une personne, une transaction, un produit).
2. Dans chaque ligne, les **champs** ou **colonnes** de données sont **séparés par un délimiteur**, qui est le plus souvent une **virgule**. Cependant, d'autres caractères comme le point-virgule (;), la tabulation (\t), ou autres caractères peuvent être utilisés.
3. La **première ligne** du fichier contient souvent les noms des colonnes.

Exemple de contenu CSV :

Extrait de code

```
Nom, Cours, Note  
Josée, Info1, 82  
Sylvio, Info2, 78
```

Dans cet exemple :

- Les enregistrements sont : Josée,Info1,82 et Sylvio,Info2,78
- Les champs sont séparés par la virgule.
- Les en-têtes de colonnes sont : Nom, Cours,Note

Utilisation des fichiers CSV et module csv avec Python

Il existe un module en Python qui permet de traiter relativement facilement la lecture et l'écriture des fichiers CSV.

Lecture d'un fichier CSV

C'est une excellente pratique d'utiliser **csv.DictReader** et **csv.DictWriter** en Python pour manipuler les fichiers CSV, car cela vous permet de travailler avec des **dictionnaires** (où les clés sont les noms des colonnes) plutôt qu'avec des listes d'indices. Cela rend le code plus lisible et moins sujet aux erreurs.

Voici un exemple complet montrant comment lire et modifier des données, puis les réécrire en utilisant ces classes.

Exemple avec csv.DictReader

Supposons que le fichier notes.csv contienne ceci :

```
Nom, Cours, Note  
Josée, Info1, 82  
Sylvio, Info2, 78
```

Les étapes sont habituellement les suivantes :

1. On importe le module csv
2. On ouvre le fichier csv avec la méthode with...as. On ouvre le fichier en mode lecture et en utilisant l'option newline=""
3. On crée l'objet DictReader
4. On parcourt l'objet DictReader dont chaque élément sera un dictionnaire dont la clé sera le nom de colonne et la valeur sera la donnée correspondante dans cette colonne.

Voici l'exemple :

```
fichier = 'notes.csv'
donnees_modifiees = []

with open(fichier, mode="r", newline="") as fichier_csv:
    # Création de l'objet DictReader
    lignes = csv.DictReader(fichier_csv)

    # Parcourir chaque ligne (qui est un dictionnaire)
    for ligne in lignes:
        print(ligne) #Voici le contenu d'une ligne
        print(f"{ligne['Nom']} et sa note : {ligne['Note']}")
        # La ligne ci-dessous affiche le nom et la note de l'étudiant.
        # Remarquez comment il est facile, grâce au dictionnaire, d'aller chercher

        # le nom de la personne. On utilise la clé qui est en fait le nom de la
        # colonne
        if ligne["Nom"] == "Sylvio":
            ligne["Note"] = 90
            donnees_modifiees.append(ligne)
```

Écriture de fichier CSV

Exemple de csv.DictWriter (Écriture)

Exemple pour l'écriture :

```
# Nous allons ici écrire la liste de dictionnaire précédemment modifiée dans un
fichier de type CSV nommé "notesfinales.csv".
```

```
# Définir explicitement les noms de colonnes (très important pour DictWriter)
```

```
nomColonnes = ["Nom", "Cours", "Note"]
```

```
with open("notesfinales.csv", mode="w", newline="") as fichier_csv:
```

```
    lignesAEcrire = csv.DictWriter(fichier_csv, nomColonnes)
```

```
    lignesAEcrire.writeheader()
```

```
    lignesAEcrire.writerows(donnees_modifiees)
```

```
#donnees_modifiees est une liste de dictionnaire de l'exemple précédent
```

Création des fichiers

Pour créer un nouveau fichier en Python, on utilise la méthode `open()`, avec l'un des paramètres suivants:

- «x» ce mode d'ouverture, crée un fichier s'il n'existe pas et renvoie une erreur si le fichier existe
- «a» – Append – créera un fichier si le fichier spécifié n'existe pas
- «w» – Write – créera un fichier si le fichier spécifié n'existe pas et si le fichier existe, il sera écrasé
- « r+ » – ouverture en mode lecture et écriture. Si le fichier n'existe pas, une erreur est renvoyée.

Exemple. Création d'un fichier nommé « myFile.txt »:

```
f = open ("myFile.txt", "x") # renvoie une erreur si le fichier existe.
```

Ajouter des lignes à un fichier en Python avec la méthode `writelines()`

La méthode `writelines()`, permet d'ajouter une liste de chaînes ou une liste de lignes à un fichier ouvert en mode écriture

Exemple. ajouter une liste des lignes à un fichier

```
f = open ("myFile.txt", "r+")
l = ["ligne1\n", "ligne2\n", "ligne3\n"]
f.writelines(l)
print(f.read())
f.close()
```

Ce qui affiche après exécution:

```
ligne1
ligne2
ligne3
```

Les modules

Qu'est ce qu'un module en Python ?

Un module en Python est simplement un fichier constitué de code Python qu'on peut appeler et utiliser son code sans avoir besoin de le recopier. Un module peut contenir des fonctions, des classes, des variables... Un module vous permet d'organiser logiquement votre code Python. Le regroupement de code associé dans un module rend le code plus facile à comprendre et à utiliser.

Les modules permettent de :

- **Structurer** le code en plusieurs fichiers logiques,
- **Réutiliser** des fonctions ou classes dans plusieurs projets,
- **Partager** facilement du code (via des bibliothèques),
- **Éviter les répétitions** et améliorer la **lisibilité**.

Types de modules en Python

Il existe principalement trois types de module en Python. Il s'agit des :

- Modules natifs Python
- Modules externes ou tiers
- Modules personnels

Nous allons voir ces trois types de modules. Il existe également des « packages » que nous regarderons vers la fin.

Les modules natifs

Ce sont les modules **fournis automatiquement** avec Python.

Ils sont écrits en C pour la plupart, et ne nécessitent **aucune installation**.

Exemples :

- `math` → Fonctions mathématiques
- `os` → Interactions avec le système d'exploitation
- `sys` → Accès aux variables et fonctions du système Python
- `datetime` → Manipulation de dates et d'heures
- `csv` → Lecture/écriture de fichiers csv
- `random` → Génération de nombres aléatoires

Exemple :

```
import math
print(math.sqrt(16))    # 4.0
print(math.pi)         # 3.141592653589793
```

Les modules externes (ou tiers)

Ce sont des modules **développés par la communauté** et installés via **pip** (le gestionnaire de paquets Python).

 Exemples :

- `numpy` → Calculs numériques et matrices
- `pandas` → Analyse de données
- `requests` → Requêtes HTTP
- `psutil` → Gestion et surveillance du système
- `netifaces` → Gestion des cartes réseaux
- `matplotlib` → Graphiques et visualisation

On verra comment les installer dans la section sur l'installation de module.

Les modules personnalisés

Ce sont les modules que **vous créez vous-même** pour organiser votre code.

Ils peuvent être utilisés dans le même projet ou partagés entre plusieurs projets.

Importation de modules

Importation du module au complet : `import module`

C'est la forme la plus courante.

Elle importe **tout le module** et oblige à utiliser le **nom du module** pour accéder à ses éléments.

Exemple :

```
import math

print(math.sqrt(16)) # On utilise math.sqrt
print(math.pi)
```

Avantage :

→ Lecture claire du code, car on sait d'où vient chaque fonction.

Inconvénient :

→ On doit toujours écrire le nom du module avant la fonction.

Importation avec alias : `import module as alias`

Permet de **renommer** un module pour l'utiliser plus facilement ou de façon plus court.

Exemple :

```
import numpy as np
import pandas as pd
```

```
# np et pd sont des alias pratiques
tableau = np.array([1, 2, 3])
print(pd.DataFrame(tableau))
```

Avantage :

- Raccourcit le code.
- Très utilisé avec les grosses bibliothèques (numpy, pandas, matplotlib, etc.).

Inconvénient :

- Peut rendre le code moins lisible si l'alias est peu clair.

Importation partielle : `from module import élément`

Permet d'importer **uniquement certaines fonctions, classes ou variables** d'un module.

Ainsi, on n'a **pas besoin du préfixe** du module.

Exemple :

```
from math import sqrt, pi

print(sqrt(9))
print(pi)
```

Avantage :

- Code plus court à écrire.

Inconvénient :

- Risque de conflit de noms si une autre variable du même nom existe dans le programme.

Importation de tout le contenu :

```
from module import *
```

Cette syntaxe importe **tous les éléments publics** d'un module **directement** dans l'espace de noms courant.

Exemple :

```
from math import *

print(sin(pi / 2)) # 1.0
```

Attention :

- Cette méthode est **fortement déconseillée**, car elle peut provoquer des **conflits de noms**.
- On ne sait plus quelle fonction vient de quel module.
- On perd en **lisibilité** et la maintenance de code peut devenir difficile.

Importation conditionnelle

On peut importer un module **seulement dans certaines conditions**, par exemple selon le système d'exploitation ou si une bibliothèque est installée.

Exemple :

```
import platform

if platform.system() == "Windows":
    import winsound
    winsound.Beep(1000, 500)
else:
    print("Ce module n'est disponible que sur Windows.")
```

Avantage :

→ Permet d'écrire du code multiplateforme.

Inconvénient :

→ Peut rendre le code plus difficile à maintenir.

Importation depuis un package

Quand un module fait partie d'un **package** (un dossier contenant plusieurs modules), on peut importer de manière hiérarchique.

Exemple :

Structure :

```
mon_package/
├── __init__.py
├── outils/
│   ├── __init__.py
│   └── math_utils.py
└── monFichier.py
```

Dans monFichier.py :

```
from mon_package.outils.math_utils import addition
```

En résumé

À privilégier :

- `import module`
- `import module as alias`
- `from module import fonction`

À éviter sauf cas spécial :

- `from module import *`

Installation d'un module

Il existe deux possibilités pour installer un module et la méthode privilégiée consiste à installer le module dans un environnement virtuel de façon à contrôler la gestion des modules dans un environnement isolé.

Installation dans un environnement virtuel

Créer un environnement virtuel

Exécuter la commande suivante pour créer l'environnement :

```
python -m venv dossier_voulu
```

➡ Cela crée un dossier « dossier_voulu » contenant une copie isolée de Python.

Activer l'environnement virtuel

Sous Windows

```
venv\Scripts\activate
```

Sous Linux :

```
source dossier_voulu/bin/activate
```

Remarque :

Vous pouvez facilement savoir si vous êtes dans votre environnement virtuel ou non. L'invite de commande commencera par (venv).

Installer un module dans l'environnement virtuel

On utilise la commande « pip » pour installer un module dans notre environnement virtuel.

```
pip install nom_du_module
```

Exemples :

```
pip install requests  
pip install numpy==1.26.0
```

Désactiver l'environnement virtuel

On désactive l'environnement virtuel en faisant la commande :

```
deactivate
```

Installation d'un module dans le système

L'autre possibilité pour l'installation d'un module consiste à installer le module sur le système de façon globale. Le module sera donc disponible pour quiconque veut bien l'utiliser. Ce n'est pas la méthode recommandée. Toutefois, dans un contexte où le système est une machine virtuelle, cette façon de faire est moins problématique.

La méthode ressemble beaucoup à celle avec la commande « pip » mais on utilise directement la commande « apt » pour l'installation d'un paquet.

Voici la commande générale :

```
apt install python3-nom_du_module
```

Exemple :

On veut installer le paquet numpy.

On fera :

```
apt install python3-numpy
```

Module relié à la gestion et à l'administration du système

Pour plus d'information : <https://pythongeeks.org/python-os-module/>

Nous allons discuter de trois modules Python (**psutil**, **netifaces** et **os**) qui sont souvent utilisés dans les scripts d'administration, de surveillance et d'automatisation système.

Le module OS (plus d'information à : <https://pythongeeks.org/python-os-module/>)

Le module OS est un module qui permet d'interagir avec le système d'exploitation. Il permet, entre autre, de faire la gestion des fichiers et des dossiers et plusieurs autres commandes du système.

Pour charger un module, on utilise l'instruction `import`.

Ainsi, pour charger le module `os`, on fera :

```
import os
```

Voici quelques commandes disponibles dans ce module

getcwd()

Cette méthode renvoie le répertoire de travail courant sous la forme d'une chaîne. Vous n'avez pas besoin de lui passer quoi que ce soit. C'est à peu près l'équivalent de **pwd**.

```
print(os.getcwd())
```

chdir()

chdir() est l'équivalent Python de **cd**. Appelez la méthode et transmettez-lui le répertoire vers lequel vous souhaitez changer sous forme de chaîne.

```
os.chdir('/accueil/utilisateur/Documents')
```

Il prend également en charge l'utilisation de chemins relatifs comme **cd**.

```
os.chdir('../Téléchargements')
```

listdir()

L'utilisation **listdir()** est très similaire à la commande « ls ». Il y a une grande différence, cependant, la valeur de retour.

Au lieu d'imprimer le contenu du répertoire, vous le recevez en retour sous forme de liste.

```
listeFichier = os.listdir("Desktop")
print(listeFichier)
```

Vous pouvez spécifier le dossier où le contenu sera affiché.

mkdir() et makedirs()

Ces deux-là sont exactement ce que vous attendez d'eux. **mkdir()** fonctionne de manière très similaire à la commande Linux que vous connaissez. Dans ce cas, il peut prendre un deuxième paramètre pour spécifier les autorisations.

```
os.mkdir('testdir', 0o755)
```

La méthode **makedirs()** fonctionne de la même manière, mais peut créer des répertoires de manière récursive les uns dans les autres, éliminant ainsi le besoin d'exécuter plusieurs fois la méthode **mkdir()**.

```
os.makedirs('testdir2/quelque chose/quelque chose d'autre', 0o755)
```

chmod()

C'est encore une fois un équivalent simple à son homologue traditionnel. **chmod()** pour Python prend deux arguments. Le premier est le répertoire à modifier et le second les autorisations de modification.

```
os.chmod('test', 0o774)
```

chown()

Vous pouvez également changer de propriétaire avec Python. **chown()** est la façon dont vous le faites. La méthode est très similaire à la commande Linux, mais elle nécessite des identifiants d'utilisateur et de groupe pour fonctionner.

```
os.chown('test', 1000, 1000)
```

Cette commande changera la propriété du répertoire "test" en l'utilisateur avec un identifiant de 1000 et le groupe avec un identifiant de 1000.

remove() et rmdir()

remove() supprime le fichier passé en paramètre. Il supprime un seul fichier. Vous pouvez également lui transmettre le chemin complet du fichier.

```
os.remove('/home/user/Downloads/somefile')
```

Pour supprimer une arborescence de dossier, utilisez la méthode **rmdir()**. Cette méthode fonctionne à la manière de la commande **rm** avec l'option « -r » pour supprimer un dossier et son contenu.

```
os.rmdir('test_dir')
```

Le module psutil – Gestion des processus et des ressources systèmes

Pour plus d'information : <https://psutil.readthedocs.io/en/latest/>

psutil (pour *process and system utilities*) est un module Python permettant d'obtenir des informations sur l'utilisation des **ressources système** (CPU, mémoire, disques, réseau, capteurs, etc.) et de **gérer les processus** de manière portable (Windows, Linux, macOS, etc.).

Il est largement utilisé pour :

- la **surveillance système**,
- la **gestion des processus**,
- les **tableaux de bord de performance**,
- les **outils d'administration** ou de **diagnostic**.

Installation (dans un environnement virtuel)

```
pip install psutil
```

Quelques exemples (Plus d'information à <https://psutil.readthedocs.io/en/latest/>)

Informations sur le CPU

psutil permet d'accéder à :

- l'utilisation du CPU,
- le nombre de cœurs,
- les temps d'utilisation,
- la fréquence du CPU.

Exemple :

```
import psutil

print("Nombre de coeurs physiques :", psutil.cpu_count(logical=False))

# Fréquence du CPU
freq = psutil.cpu_freq()
```

Mémoire (RAM et swap)

psutil fournit des fonctions pour surveiller :

- la mémoire virtuelle (RAM),

- la mémoire d'échange (swap).

Exemple :

```
import psutil

# Mémoire principale
mem = psutil.virtual_memory()
print(f"Mémoire totale : {mem.total / (1024 ** 3):.2f} Go")
```

Gestion des processus

psutil permet de :

- lister les processus,
- accéder à leurs informations,
- les tuer, suspendre, etc.

Lister tous les processus :

```
import psutil

for proc in psutil.process_iter(['pid', 'name', 'username']):
    print(proc.info)
```

Accéder à un processus précis :

```
p = psutil.Process(1) # PID 1
print("Nom :", p.name())
print("Statut :", p.status())
print("Utilisation CPU :", p.cpu_percent(interval=0.5))
print("Mémoire utilisée :", p.memory_info().rss / (1024 ** 2), "Mo")
```

Terminer un processus :

```
p = psutil.Process(1234) # Exemple PID
p.terminate()
p.wait(timeout=3)
```

Le module netifaces

Pour plus d'information: <https://pypi.org/project/netifaces/>

C'est un module Python non standard qui permet de récupérer les informations réseau des interfaces sur un système (Linux, Windows, macOS...).

Avec netifaces, on peut obtenir, entre autre :

- La liste des interfaces réseau (eth0, lo, wlan0, etc.)
- Les adresses IP associées à chaque interface (IPv4, IPv6)
- Les adresses MAC
- Les passerelles par défaut

C'est un outil pratique pour des scripts réseau ou des diagnostics automatisés.

Installation (dans un environnement virtuel)

```
pip install netifaces
```

Installation (dans le système localement)

```
apt install python3-netifaces
```

Quelques exemples (plus de détail à : <https://pypi.org/project/netifaces/>)

Lister les interfaces réseau

```
import netifaces

interfaces = netifaces.interfaces()
print(interfaces)
```

Exemple de sortie :

```
['lo', 'eth0', 'wlan0']
```

Obtenir les adresses IP d'une interface

```
iface = 'eth0'
addrs = netifaces.ifaddresses(iface)
print(addrs)
```

- `addrs[netifaces.AF_INET]` → liste des adresses IPv4
- `addrs[netifaces.AF_INET6]` → liste des adresses IPv6
- `addrs[netifaces.AF_LINK]` → adresse MAC

Exemple :

```

ipv4 = addrs[netifaces.AF_INET][0]['addr']
mac = addrs[netifaces.AF_LINK][0]['addr']
print("IPv4:", ipv4)
print("MAC :", mac)

```

Obtenir les passerelles par défaut

```

gateways = netifaces.gateways()
print(gateways)

```

Exemple de sortie :

```

{'default': {netifaces.AF_INET: '192.168.1.1'},
 netifaces.AF_INET: [('192.168.1.1', 'eth0', True)]}

```

- `gateways['default'][netifaces.AF_INET]` → passerelle IPv4 par défaut

Créer son propre module

Étape 1 : Créer le fichier du module

1. Crée un fichier, par exemple `mon_module.py`.
2. Le fichier contiendra les fonctions utiles que vous voulez y mettre.

Exemple :

```

# mon_module.py

def Addition (a, b):
    return a + b

def Multiplication(a, b):
    return a * b

def AireCercle(rayonCercle) :

    return math.PI * rayonCercle**2

```

Étape 2 : Utiliser le module dans un autre script

Crée un autre fichier, par exemple `test_module.py`, dans **le même dossier** :

```
import mon_module

print(mon_module.Addition(3, -5))      # -2
print(mon_module.AireCercle(10))      # 314.159
```

Créer un package

Utile lorsqu'on veut regrouper plusieurs modules dans un même dossier.

1. Créer un dossier `mon_package/`.
2. Mettre un fichier `__init__.py` (peut être vide).
3. Mettre les modules à l'intérieur du dossier.

Usage :

```
from mon_package.module1 import Addition
```