# Python

## Commentaires et documentation

• **Écrire un commentaire :** # commentaire

• **Écrire de la documentation (docstring) :** (extension autoDocstring sur VS Code)

**"""**\_summary\_ : description

Args:

paramètre1: description du paramètre1

paramètre2: description du paramètre2

Returns:

\_type\_: valeur de retour

Raises:

TypeError: condition de l’erreur

**"""**

Remarque : il faut éviter les accents, même dans les commentaires.

## Types de variable

• **Entier :** int, Integer

• **Flottant** (décimal) **:** float

• **Chaine de caractères :** str

• **Booléen** (vrai, faux) **:** bool (true, false)

Remarque : Python gère la déclaration à la volée. Les types sont donc implicites.

## Opérations mathématiques

• **Opérateurs de comparaison :** ==, !=, >, >=, <, <=

• **Opérateurs arithmétiques :** +, -, \*, \*\* (puissance), /, // (pour garder uniquement par partie entière de la division), % (modulo = division entière)

• **Opérateurs logiques :** not, or, and

Remarque : pour utiliser toutes les opérations mathématiques usuelles, importer le module « math »:

from math import sin, pi **OU :** from math import \*

## Opérations sur les variables

• **Déclarer une variable :** variable = valeur

Remarque : on ne peut pas nommer une variable avec un mot réservé. *Liste des 32 mots réservés : and, as, assert, async, await, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield.*

• **Affecter une valeur à une variable :** variable = valeur

• **Affecter différentes variables simultanément :** variable1, variable2 = valeur1, valeur2

• **Permuter les valeurs de deux variables :** variable1, variable2 = variable2, variable1

• **Incrémenter une valeur :** variable = variable + 1 **OU :** variable+=1 (pareil pour les autres opérateurs arithmétiques)

• **Décrémenter une valeur :** variable = variable - 1

## Transtypage (cast)

• **Convertir une variable en chaine de caractères :** str(variable)

• **Convertir une variable en entier :** int(variable)

• **Convertir une variable en décimal :** float(variable)

• **Convertir une variable en booléen :** bool(variable)

• **Convertir une chaine de caractère/un tuple en liste :** list(chaine/tuple)

• **Convertir une liste en tuple :** tuple(liste)

## Entrées et sorties

• **Récupérer la saisie de l’utilisateur :** variable = input("texte\_à\_afficher\_à\_l’utilisateur")

• **Afficher une variable :** print(variable)

## Opérations sur les chaines de caractères

• **Obtenir la longueur d’une chaine :** len('chaine' )

• **Concaténer deux chaines :** chaine3 = chaine1 + chaine2

• **Répéter une chaine :** chaine2 = chaine1 \* nombre

• **Scinder une chaine en liste de mots :** 'chaine'.split('séparateur')

• **Concaténer une liste en une chaine :** 'séparateur'.join(liste)

• **Donner la position d’une sous-chaine dans une chaine :** 'chaine'.find('sous-chaine') (le premier indice vaut 0 ; -1 sera retourné si la sous-chaine n’est pas trouvé)

• **Donner le nombre de sous-chaines dans une chaine :** 'chaine'.count('sous-chaine')

• **Convertir une chaine en minuscules :** 'chaine'.lower()

• **Convertir une chaine en majuscules :** 'chaine'.upper()

• **Convertir la première lettre d’une chaine en majuscule :** 'chaine'.capitalize()

• **Convertir la première lettre de tous les mots en majuscule :** 'chaine'.title()

• **Intervertir les casses d’une chaine :** 'chaine'.swapcase()

• **Supprimer les blancs en début et en fin de chaine :** 'chaine'.strip()

• **Remplacer une sous-chaine par une autre :** 'chaine'.replace('sous-chaine1', 'sous‑chaine2')

• **Récupérer un caractère d’une chaine :** chaine[position] (commence à 0)

• **Récupérer les caractères d’une chaine de la position1 à la position2 :** chaine[position1:position2-1] (il faut bien enlever 1 à la deuxième position)

• **Récupérer les caractères d’une chaine jusqu’à une position :** chaine[:position-1] (idem)

• **Récupérer les caractères d’une chaine à partir d’une position :** chaine[position:] (idem)

• **Récupérer le dernier caractère d’une chaine :** chaine[-1]

• **Insérer une variable dans une chaine :** chaine = f"{variable}" ou chaine = "{formatage}".format(variable). Ex : {:3d} → 3 entiers, {:02.0f} → 2 chiffres après la virgule, la variable est un nombre flottant

## Structures de données

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Liste** | **Tuple** | **Set** | **Dictionnaire** |
| **Mutable** (différents types) | X |  | X | X |
| **Ordonné** | X | X |  | X (Python 3.7 et +) |
| **Doublons** | X |  |  |  |
| **Rapide** | Lent | Plus rapide que les listes | Plus lent pour l’insertion | • Rapide  • Plus lent en insertion et suppression |
| **Modifiable** | X |  |  | X |
| **Accessibilité des éléments** | Index | Index | Index | Clé |

### Listes

• **Déclarer une liste vide :** liste = []

• **Déclarer une liste :** liste = [élément1, élément2]

• **Créer une liste de listes :** liste3 = [liste1, liste2]

• **Créer une liste jusqu’à un nombre :** range(nombre+1)

• **Créer une liste d’un nombre à un autre nombre :** range(nombre1, nombre2+1)

•  **Ajouter un élément à la fin d’une liste :** liste.append(élément)

• **Trier une liste :** liste.sort()

• **Trier une liste mais à l’inverse :** liste.reverse()

• **Trier une liste par l’élément numéro 1 de sa sous-liste imbriquée :** liste.sort(key=lambda i: i[1])

• **Rechercher l’index d’un élément dans une liste :** liste.index(élément)

• **Supprimer un élément d’une liste :** liste.remove(élément)

• **Supprimer le contenu d’une liste** : liste.clear()

• **Récupérer un élément d’une liste :** liste[position] (commence à 0)

• **Récupérer les éléments d’une liste de la position1 à la position2 :** liste[position1:position2-1] (il faut bien enlever 1 à la deuxième position)

• **Récupérer les éléments d’une liste jusqu’à une position :** liste[:position-1] (idem)

• **Récupérer le dernier élément d’une liste :** liste[-1]

• **Vérifier si un élément appartient à une liste :** élément in liste

• **Répéter un(des) élément(s) dans une liste :** liste = [élément1, élément2] \* nombre\_de\_fois

• **Obtenir la longueur d’une liste :** len(liste )

• **Donner le nombre fois où un élément se trouve dans la liste :** liste.count(élément)

• **Ajouter les éléments de la liste2 à la fin de la liste :** liste.extend(liste2)

• **Manipuler deux chaines comme un dictionnaire :** zip(liste1, liste2)

### Tuples

• **Déclarer un tuple :** tuple = (élément1, élément2)

Remarque : les fonctions s’appliquant aux listes s’appliquent aux tuples (sauf celles qui modifient les données)

### Sets

• **Déclarer un set :** set = {élément1, élément2}

• **Ajouter les éléments à un set :** set.add(élément)

### Dictionnaires

• **Déclarer un dictionnaire vide :** dico = { }

• **Déclarer un dictionnaire :** dico = { "clé": "valeur", "clé": "valeur" }

• **Créer un couple clé-valeur (ou modifier la valeur de l’élément) :** dico["clé"] = "valeur"

• **Supprimer un élément par une clé de dictionnaire :** del dico["clé"]

• **Supprimer un élément et renvoyer une valeur :** dico.pop("clé")

• **Supprimer un élément et renvoyer une clé et une valeur :** dico.popitem("clé")

• **Supprimer tous les éléments du dictionnaire :** dico.clear()

• **Afficher toutes les clés d’un dictionnaire :** dico.keys()

• **Afficher toutes les valeurs d’un dictionnaire :** dico.values()

• **Afficher tous les couples d’un dictionnaire :** dico.items()

• **Déterminer si la clé existe dans le dictionnaire :** dico.has\_key() (si la clé est dans le dictionnaire, la fonction renvoie true, sinon false)

### Listes en compréhension

• **Liste en compréhension :** expression qui permet de générer une liste de manière très compacte, équivalente à une boucle for qui construirait la même liste en utilisant la méthode append().

• Syntaxe :

result = [valeur for x in séquence condition2]

# est équivalent à :

result = []

for x in séquence:

condition2:

result.append(valeur)

Remarque : la condition2 peut aller de « if x > 23 » à « for y in séquence2 »

## Structures de contrôle

### Structures conditionnelles

• **Exécuter un bloc d’instructions si une condition est remplie avec IF :**

if condition1:

instructions

elif condition2:

instructions

else:

instructions

• **Syntaxe compacte du IF :**

variable = valeur1 if condition else valeur2

# est équivalent à :

if condition:

variable = valeur1

else:

variable = valeur2

• **Exécuter un bloc d’instructions si une condition est remplie avec MATCH** (à partir de Python 3.10)**:**

case 0:

instructions

…

case N:

instructions

case default:

instructions

• **Vérifier si deux conditions sont vraies en même temps :** if condition1 and condition2:

• **Vérifier si l’une des deux conditions est vraie :** if condition1 or condition2:

• **Vérifier si une variable est égale à 0 / faux / vide :** if not variable:

• **Vérifier si une valeur est présente dans une liste :** if valeur in liste:

• **Vérifier si deux variables pointent vers le même objet :** if variable1 is variable2:

• **Vérifier si deux variables ne pointent pas vers le même objet :** if variable1 is not variable2:

### Boucles

• **Exécuter un bloc d’instructions tant qu’une condition est remplie :** while condition:

### Structures itératives

• **Parcourir une liste/tuple/set :** for valeur in liste/tuple/set:

• **Exécuter un bloc d’instructions sur un nombre défini d’itérations :** for k in range(nombre+1):

• **Parcourir des chiffres de nombre1 à nombre2 :** for k in range(nombre1,nombre2+1):

• **Parcourir les clés d’un dictionnaire :** for k in dico.keys():

• **Parcourir les valeurs d’un dictionnaire :** for k in dico.values():

• **Parcourir les couples (clé/valeur) d’un dictionnaire :** for k in dico.items():

• **Parcourir deux listes à la fois, tel un dictionnaire :** for valeur1, valeur2 in zip(liste1, liste2):

• **Parcourir deux listes à la fois, tel un dictionnaire mais en itérant :** for k, (valeur1, valeur2) in enumerate(zip(liste1, liste2)):

## Fonctions

• **Écrire une fonction :**

def fonction(paramètre1, paramètre2 : type = valeur\_par\_défaut) -> type\_retourné :

""" Documentation de la fonction. """

instructions

Remarques :

— un paramètre peut être une fonction.

— il est préférable de définir les paramètres par défaut en dernier.

• **Appeler une fonction :**

fonction(paramètre1=valeur1, valeur2)

• **Passer un nombre variable de paramètres dans une fonction et les parcourir avec un(e) tuple/liste :**

def fonction(\*args):

for k in args:

• **Appeler une fonction avec un nombre variable de paramètres avec un(e) tuple/liste :** fonction(\*liste)

• **Passer un nombre variable de paramètres dans une fonction et les parcourir avec un dictionnaire :**

def fonction(\*\*kwargs):

for k in kwargs.values():

• **Appeler une fonction avec un nombre variable de paramètres avec un dictionnaire :** fonction(\*\*dico)

• **Sortir des valeurs d’une fonction :** return valeur1, valeur2 (un retour de fonction arrête l’exécution de la fonction)

• **Récupérer les valeurs d’une fonction dans des variables :** variable1, variable2 = fonction()

• **Typer les paramètres d’une fonction et le retour :**

def fonction(paramètre1 : type, paramètre2 : type) -> type:

Remarque : c’est une bonne pratique de typer la sortie mais Python n’en tient pas compte.

• **Créer une fonction anonyme, qui n’a pas de nom et qu’on ne va jamais réutiliser** (expression lambda) **:** (lambda paramètre1, paramètre2 = valeur\_par\_défaut : paramètre1 + paramètre2) (valeur\_paramètre1)

• **Appliquer une fonction sur chaque élément d’une liste avec une expression lambda :** list(map(expression\_lamda, liste)

• **Filtrer une liste en fonction d’une condition avec une expression lambda :** list(filter(expression\_lamda, liste)

• **Réduire une liste en une seule valeur avec une expression lambda :** reduce(expression\_lamda, liste)

## Modules

• **Importer un module complet** (mauvaise pratique)**:** import module as nom\_module\_résumé

• **Importer des fonctions particulières d’un module :** from module import fonction1, fonction2

• **Empêcher une partie du code d’être exécutée lorsque le module est importé** (vérification que le fichier est bien exécuté en tant que premier fichier)**:**

if \_\_name\_\_ == "\_\_main\_\_"

• **Voir la liste des fonctions d’un module** (après avoir importé le module)**:** print(dir(module))

• **Utiliser une fonction d’un module** (si importé « import module ») **:** module.fonction()

• **Utiliser une fonction d’un module** (si importé « from module import fonction ») : fonction()

## PIP : gestionnaire de paquets utilisé en ligne de commandes

• **Rechercher un paquet :** aller sur <https://pypi.org/>

• **Installer la dernière version de paquets** (qui n’est pas dans la bibliothèque standard)(avant de pouvoir l’importer et l’utiliser comme module)**:** pip install paquet1, paquet2

Remarque : pour installer des paquets uniquement sur un projet particulier, il faut être connecté sur l’environnement virtuel avant de faire « pip install ».

• **Installer une version précise de paquet :** pip install -Iv paquet==version

• **Mettre à jour un paquet vers la dernière version :** pip install -U paquet

• **Mettre à jour un paquet vers une version précise :** pip install -U paquet==version

• **Désinstaller un paquet :** pip uninstall paquet

• **Lister les paquets installés :** pip freeze

• **Sauvegarder la liste des paquets installés dans un fichier :** pip freeze > fichier.txt

• **Installer tous les paquets depuis un fichier :** pip install -r fichier/base.txt

• **Voir les informations d’un paquet installé :** pip show paquet

## Classes et objets

### Dans le fichier principal

• **Utiliser un fichier de classe dans le fichier principal :** from Classe import \*

• **Instancier une classe** (créer un objet à partir de la classe) **:** objet = Classe()

• **Appeler une méthode depuis l’objet :** objet.méthode()

• **Utiliser un attribut depuis l’objet :** objet.attribut

• **Convertir un objet en chaine de caractère :** print(objet)

• **Afficher les attributs d’un objet :** print(dir(objet)) **OU :** print(objet.\_\_dict\_\_)

• **Afficher un objet :** print(objet)

### Dans la classe

• **Déclarer une classe :**

class Classe:

""" Documentation de la classe. """

instructions

• **Définir une méthode dans une classe :** def méthode(self, paramètre1, paramètreN):

• **Appeler une méthode depuis une autre méthode dans la classe :** self.méthode()

• **Instancier une classe** (constructeur)**:** def \_\_init\_\_(self, paramètre1, paramètreN): (la bonne pratique est de déclarer les attributs ici uniquement)

• **Définir un attribut dans une méthode :** self.attribut = valeur

• **Appeler un attribut depuis une autre méthode :** self.attribut (ne fonctionne pas si la variable n’est pas déclaré dans \_\_init\_\_)

• **Convertir un objet en chaine de caractère** (méthode)**:**

def \_\_str\_\_(self):

return "chaine de caractère"

• **Afficher les attributs d’un objet** (méthode)**:**

def \_\_dir\_\_(self):

return ["attribut1", "attributN"]

**OU :**

def \_\_dict\_\_(self):

return {"attribut1": "valeur1", "attributN": "valeurN"}

• **Afficher un objet** (méthode) **:**

def \_\_repr\_\_(self):

return "Classe(attribut1='valeur1', attribut2='valeur2')"

• **Définir un attribut de classe** (attribut statique) **:**

class Classe:

attribut = valeur (avant le \_\_init\_\_)

• **Définir une méthode de classe** (méthode statique) **:**

@staticmethod

def méthode():

• **Appeler une méthode depuis la classe :** Classe.méthode()

• **Utiliser un attribut depuis la classe :** Classe.attribut

### Héritage

• **Créer une classe à partir d’une classe existante** : class ClasseFille(ClasseMère):

• **Appeler une méthode de classe parente dans la classe fille :** super().méthode()

• **Redéfinir une méthode :** définir une méthode avec le même nom dans la classe fille

• **Surcharger une méthode :** appeler la méthode de la classe parente dans la classe fille et ajouter du code

### Visibilité

• **Public :** accessible depuis n’importe où. Ex : self.attribut = valeur

• **Protected :** accessible depuis la classe et ses classes filles. Ex : self.\_attribut = valeur (c’est juste une convention)

• **Private :** accessible uniquement depuis la classe. Ex : self.\_\_attribut = valeur

### Encapsulation

• **Définir un getter :**

def get\_attribut(self)

return self.\_\_attribut

**OU :**

@property

def attribut(self)

return self.\_\_attribut

• **Définir un setter :**

def set\_attribut(self, valeur)

self.\_\_attribut = valeur

**OU :**

@attribut.setter

def attribut(self, valeur)

self.\_\_attribut = valeur

### Polymorphisme

• **Polymorphisme de substitution :** les objets de classes différentes peuvent être utilisés de manière interchangeable, à condition que la classe définie soit la classe parente. Ex :

*class Animal:*

*def make\_sound(self):*

*print("Animal making sound")*

*class Dog(Animal):*

*def make\_sound(self):*

*print("Woof")*

*class Cat(Animal):*

*def make\_sound(self):*

*print("Meow")*

***def make\_animal\_sound(animal : Animal):***

*animal.make\_sound()*

• **Polymorphisme de surcharge :** on peut redéfinir plusieurs méthodes provenant d’une classe parente pour changer leur comportement

*class Shape:*

***def area(self):***

*print("Shape’s area")*

*class Rectangle(Shape):*

***def area(self):***

*print(f"Rectangle’s area: {self.width \* self.height}")*

*class Circle(Shape):*

***def area(self):***

*print(f"Circle’s area: {3.14 \* self.radius \* self.radius}")*

• **Polymorphisme d’interface :** on peut définir des méthodes avec le même nom dans des classes différentes qui accompliront des tâches différentes via l’utilisation de classes abstraites

*class Shape(ABC):*

***@abstractmethod***

*def area(self):*

***pass***

*class Rectangle(Shape):*

*def area(self):*

*print(f"Rectangle’s area: {self.width \* self.height}")*

*class Circle(Shape):*

*def area(self):*

*print(f"Circle’s area: {3.14 \* self.radius \* self.radius}")*

### Abstraction (déprécié)

Remarque : utiliser « from abc import ABC, abstractmethod »

• **Déclarer une classe abstraite** (une classe abstraite ne peut pas être instanciée)**:**

class Classe(ABC):

• **Déclarer une méthode abstraite** (une méthode abstraite est une méthode qui n’a pas de corps et qui doit être redéfinie dans les classes filles)**:**

@abstractmethod

def méthode(self):

### Relation entre les objets

• **Composition :** relation entre deux objets où l’un est contenu dans l’autre (si l’objet contenant est détruit, l’objet contenu est détruit). Exemple :

*class House:*

*def \_\_init\_\_(self):*

*self.porte = Door()*

• **Agrégation :** relation entre deux objets où l’un est importé dans l’autre. Exemple :

*class Voiture:*

*…*

*class Garage:*

*def \_\_init\_\_(self, voiture):*

*self.voiture = voiture*

*voiture = Voiture():*

*garage = Garage(voiture)*

## Gestion d’erreurs

• **Tester un code qui peut poser problème** (try) **et définir les actions à prendre si une exception est rencontrée** (except)**, exécuter du code s’il n’y a aucune erreur** (else) **et exécuter du code dans tous les cas** (finally) **:**

try:

instructions

except Exception: # toutes les exceptions (pas recommandé)

instructions

except NomException1 as nom1:

instructions

except NomException2 as nom2:

instructions

else:

instructions

finally:

instructions

• **Lever volontairement une exception :** raise NomException, "message\_d’erreur"

• **Voir la liste des erreurs possibles à gérer :**

import builtins

print(dir(builtins))

## Ignorer une partie de code

• **Sortir d’une boucle :** break

• **Sauter la partie d’une boucle où une condition est déclenchée, passer à l’itération suivante** (sans sortir de la boucle)**:** continue

## Références

• Si un objet modifiable est affecté, tout changement sur un objet modifiera l’autre. Ex :

liste1 = ['a', 'b'] # liste1 = ['a', 'b']

liste2 = liste1 # liste2 = ['a', 'b']

liste1[1] = 'c' # liste1 = ['a', 'c'], liste2 = ['a', 'c']

• **Faire une vraie copie d’un objet :**

import copy

copie = copy.deepcopy(objet\_à\_copier)

## Variables locales et globales

• **Obtenir les variables locales :** locals()

• **Obtenir les variables globales :** globals()

• **Indiquer qu’on utilise une variable globale :** global variable

• Remarque : si, dans une fonction, on affecte une nouvelle valeur à une variable locale qui a le même nom qu’une variable globale, on masque la variable globale et on crée une variable locale.

## Dates

Remarque : utiliser « from datetime import \* »

• **Obtenir la date du jour :** date.today().day

• **Obtenir le mois en cours :** date.today().month

• **Obtenir l’année en cours :** date.today().year

## Chiffres aléatoires

Remarque : utiliser « import random »

• **Créer un chiffre aléatoire entier :**

chiffre = random.randint(minimum, maximum)

• **Créer un chiffre aléatoire flottant :**

chiffre = random.uniform(-minimum, maximum)

## Expressions régulières

Remarque : utiliser « import re »

• **Rechercher une sous-chaine de caractères et sa position dans une chaine** (ne trouve que la première occurrence)**:** re.search("sous-chaine", "chaine")

• **Rechercher différentes sous-chaines dans une chaine :**

re.search(r"chai[alternative1][alternative2]ne", "chaine")

• **Rechercher toutes les occurrences d’une sous-chaine de caractères dans une chaine :** re.findall("sous-chaine", "chaine")

• **Remplacer une sous-chaine de caractères dans une chaine :**

re.sub("sous‑chaine\_à\_remplacer", "sous-chaine\_qui\_remplace", "chaine")

## Fichiers

• **Ouvrir un fichier en mode lecture / écriture / ajout :** fichier = open("fichier.txt", "r/w/a")

• **Ouvrir un fichier sans avoir besoin de le fermer en mode lecture / écriture / ajout :** with open("fichier.txt", "r/w/a") as fichier:

• **Écrire une donnée simple dans un fichier :** fichier.write(valeur)

• **Écrire des données multiples dans un fichier :** fichier.writelines(liste)

• **Ajouter des données à la fin d’un fichier :** print >> fichier, données

• **Lire la totalité d’un fichier :** fichier.read(), fichier.readlines()

• **Lire au plus x octets :** fichier.read(x)

• **Lire la ligne suivante :** fichier.readline()

• **Parcourir les lignes d’un fichier :** for ligne in fichier:

• **Trouver si une ligne contient une valeur :** if valeur in ligne: **OU :** if ligne.find(valeur) != -1: **OU:** if ligne.\_\_contains\_\_(valeur)

• **Remplacer une chaine de caractères par une autre sur une ligne :** ligne.replace(chaine\_à\_remplacer, chaine\_qui\_remplace)

• **Supprimer les blancs qu’il y a en trop sur une ligne :** ligne.strip()

• **Fermer un fichier :** fichier.close()

## Venv : module utilisé pour créer et gérer des environnements virtuels

• **Installer ven :** py -m pip install --user virtualenv

• **Créer un environnement virtuel :** py -m venv .venv

• **Activer l’environnement virtuel** (sous Powershell)**:** .\venv\Script\Activate.ps1 (ou utiliser le plugin « Python Environment Manager)

• **Autoriser l’exécution du script « Activate.ps1 »** (sous Powershell en mode administrateur) **:** Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope CurrentUser

## Module « pandas » : utilisé pour gérer des données

Remarque : utiliser « pip install pandas » et « import pandas »

• **Récupérer un dataframe à partir d’un fichier CSV :** dataframe = pandas.read\_csv(fichier)

• **Sélectionner les lignes k à i du dataframe :** dataframe[k:i]

## Module « pymysql » :

Remarque : utiliser « pip install pymysql » et « import pymysql »

• **Fonction pour créer une connexion à la base de données :**

def se\_connecter\_db(host : str, user : str, password : str, database : str) -> pymysql.connections.Connection:

conn = pymysql.connect(host=host, user=user, password=password, database=database)

return conn

/\*

*def* inserer\_donnees(*conn*, *table*):

    """\_summary\_ : insère des données utilisateurs dans la base de données

    Args:

        conn (\_type\_): \_description\_

        utilisateurs (\_type\_): \_description\_

    """

    cursor = *conn*.cursor() # curseur qui se déplace d'élément en élément sur la table

    for row in *table*.iterrows():

        sql = "INSERT INTO clients (id, prenom, nom, email, profession, pays, ville) VALUES (%s, %s, %s, %s, %s, %s, %s)"

        cursor.execute(sql, (row['id'], row['firstname'], row['lastname'], row['email'], row['profession'], row['country'], row['city']))

*conn*.commit()

\*/

## Module « sys »

Remarque : utiliser « import sys »

• **Vérifier le système Python utilisé :** print(sys.argv)

• **Vérifier la version de Python utilisée :** print(sys.version)

• **Vérifier les chemins d’accès des modules Python :** print(sys.path)

• **Ajouter un chemin d’accès dans le sys :** sys.path.append("chemin")

• **Vérifier les entrées/sorties/erreurs :**

for i in (sys.stdin, sys.stdout, sys.stderr):

print(i)

• **Vérifier les modules intégrés au système :** print(sys.builtin\_module\_names)

• **Faire des appels aux scripts shell :** os.system("bash -c \"read -n 1\"")

## Module « os »

Remarque : utiliser « import os »

• **Vérifier le système d’exploitation utilisé :** print(os.name) # nt pour Windows

• **Afficher le répertoire de travail :** print(os.getcwd())

• **Changer de répertoire courant :** os.chdir("chemin")

## Fichiers à créer lorsque l’on débute un projet

• **« LICENCE » :** la licence MIT est bien pour la plupart des projets

• **« .gitignore » :** ce fichier permet d’éviter de sauvegarder des répertoires inutiles pour le repo git ou des fichiers qui peuvent être générés automatiquement

— mettre le dossier de l’environnement virtuel dedans

• **« README.md » :** explique à quoi sert le projet, sous le format Markdown

-> sous la forme : titre du projet, description, prérequis, installation

## Autre

• **Récupérer le sinus de Pi :** sin(pi)

• **Pour que les espaces soient au bon endroit dans VS Code :** alt + shift + f