

PYTHON

Présenté par :
Xavier TABUTEAU

Plan de cours

- Introduction
- Installations
- Premiers pas
- Importation de modules
- Les variables
- les blocs d'instructions
- Les structures répétitives
- Les structures conditionnelles
- Les fonctions
- TP de validation des acquis 1

- Les exceptions
- Les fichiers
- Les expressions régulières
- Les classes
- Les classes avancées
- TP de validations des acquis 2

- Les modules utiles
- Les bases de données
- Patrons de conception
- Gestionnaire de paquets
- Environnements virtuels
- Tests
- L'interface graphique wxPython
- Programmation parallèle (Threads)

Introduction

Présentation des participants

- Votre nom et prénom
- Votre entreprise et le poste occupé
- Votre expérience en programmation
- Vos attentes de cette formation
- Votre système d'exploitation
- Les accès administrateurs de votre machine

Prérequis

- Avoir les bases de la programmation (variables, fonctions, expressions, etc...).
- Savoir utilisé un IDE (Integrated Development Environment).
- Avoir des notions de classes, héritages, encapsulations et polymorphismes (concept de Programmation Orienté Objet).

Pourquoi Python ?

- Python est un langage compilé et interprété avec un typage dynamique fort, portable, extensible, gratuit, qui permet, sans l'imposer, une approche modulaire et orientée objet de la programmation. Il est compilé en byte-code (fichier « .pyc ») une seule fois à l'exécution du programme, puis ce byte-code est interprété à chaque fois. Il sera à nouveau compilé à l'exécution seulement si un changement est effectué dans le code source (fichier « .py »).
- Particularité importante : Python n'utilise pas d'accolades ou d'autres délimiteurs (begin/end...) pour repérer les blocs d'un programme, mais l'indentation.
- Un exemple : en fonction d'une liste de valeurs, nous souhaitons savoir celles qui sont des nombres pairs et celles qui ne le sont pas.

```
1  <?php
2  function valeurs_paires($liste_valeurs) {
3      $classement = array();
4
5      for($i=0; $i<count($liste_valeurs); $i++) {
6          if($liste_valeurs[$i] % 2 == 0) {
7              array_push($classement, True);
8          }
9          else {
10             array_push($classement, False);
11         }
12     }
13
14     return $classement;
15 }
16
17 echo var_dump(valeurs_paires([51, 8, 85, 9]));
18 ?>
```

```
1  def valeurs_paires(liste_valeurs):
2      return [(False if v % 2 else True) for v in liste_valeurs]
3
4  print(valeurs_paires([51, 8, 85, 9]))
```

Champs d'application de Python

Nous trouvons Python dans le Web, les multimédias, la bureautique, les utilitaires, l'intelligence artificielle, ...

Nous le retrouvons dans tous les domaines professionnels tel que :

Le domaine scientifique, les finances, la programmation système et réseau, les bases de données, ...

Python à cette force de pouvoir réunir des profils d'informaticiens assez différents (administrateur système, développeur généraliste, développeur web, etc...).

Positionnement de Python

- Indice TIOBE des langages de programmations

<https://www.tiobe.com/tiobe-index/>

Au jour où sont écrits ces lignes, Python est le langage le plus utilisé avec 15% d'avance sur C++, C et Java.

- Historique de Python

Décembre 1989 : Guido van Rossum commence à développer Python.
Février 1991 : Première version publique (v0.9.0).
26 Janvier 1994 : Python 1.0.
Octobre 2000 : Python 2.0.
3 décembre 2008 : Python 3.0 (refonte majeure du code et rupture de compatibilité).
2010 : Python devient un projet communautaire sous la Python Software Foundation.
2020 : Fin du support de Python 2 (après presque 20 ans).
Octobre 2021 : Python 3.10.

Depuis, tous les ans en octobre, Python évolue d'une version (3.11, etc...)

Avantages et inconvénients

- **Avantages**

- **Simplicité et lisibilité**

- Syntaxe claire, facile à apprendre même pour les débutants.
Idéal pour l'enseignement et le prototypage rapide.

- **Polyvalence**

- Utilisable pour le web, la data science, l'IA, les scripts, la robotique, etc.

- **Grande communauté**

- Nombreux forums, documentations et ressources d'apprentissage.

- **Enorme bibliothèque standard + packages externes**

- Modules intégrés et frameworks (Django, Flask, NumPy, Pandas, TensorFlow...).

- **Multi-plateforme**

- Fonctionne sur Windows, macOS, Linux, et même sur certaines cartes embarquées (Raspberry Pi).

Avantages et inconvénients

- **Inconvénients**

- **Vitesse d'exécution plus lente**

Interprété → moins rapide que C, C++ ou Java pour des calculs très lourds.

- **Consommation mémoire**

Peut être gourmand en ressources pour de gros projets.

- **Pas toujours adapté au mobile**

Moins utilisé pour les applications mobiles natives par rapport à Swift ou Kotlin.

- **Gestion du multithreading limitée**

A cause du Global Interpreter Lock (GIL), certaines tâches parallèles CPU sont moins efficaces.

- **Conclusion**

Python est idéal pour apprendre à programmer, démarrer rapidement un projet et travailler sur des domaines modernes comme l'IA ou l'analyse de données.

Cependant, pour des applications très performantes ou embarquées, d'autres langages peuvent être plus adaptés.

Les interpréteurs Python

Bien que Python possède son propre interpréteur (CPython) et soit le standard, il en existe d'autres tel que IPython et bpython. IPython est un interpréteur interactif avancé, très populaire dans la communauté scientifique. bpython est une alternative plus légère et plus colorée à lpython.

- **Python (standard)**

Avantages :

- Léger et toujours disponible.

- Suffisant pour tester des commandes simples.

Inconvénients :

- Pas très convivial.

- Pas de complétion automatique ni d'historique riche.

- Pas de coloration syntaxique. (sauf pour les sorties d'erreurs depuis les nouvelles version 3.10+).

- **IPython (Interactive Python)**

Avantages :

- Complétion automatique avec tabulation.

- Historique des commandes amélioré.

- Coloration syntaxique.

- Support du debugging, du profiling, et des magics (commandes spéciales comme %time, %run, etc.).

- Intégration avec Jupyter Notebook.

Usage typique :

- Très utilisé en science des données, recherche...

Les interpréteurs Python

Le "b" dans bpython ne correspond pas à un mot précis officiellement documenté par les créateurs, mais il est généralement compris comme signifiant "better" ou "beautiful", pour insister sur le fait que bpython est une version améliorée, plus "jolie" et ergonomique que l'interpréteur Python standard.

- **bpython**

Avantages :

- Interface colorée et agréable.

- Suggestions automatiques de code.

- Possibilité de voir le code source d'une fonction.

- Rewind : possibilité d'effacer la dernière commande.

- Fonctionne bien dans des environnements limités.

Usage typique :

- Pour les développeurs qui aiment travailler en terminal, avec un outil léger et interactif.

Installation de Python

Environnement de travail

- Installer Python

Python sera installé grâce à un fichier exécutable que vous pouvez télécharger à ce lien :
<https://www.python.org/downloads/>

- Lancer son premier programme

Grâce à la console Python, nous pourrons exécuter notre première application.

Installation de Python

Avant de se rendre pour le téléchargement, vous pouvez vérifier si Python est installé dans votre ordinateur en exécutant la commande : `python --version`

Python fait partie des principales distributions Linux et vient avec tout Mac équipé de Mac OS.

Grace au lien ci-dessous, vous êtes en mesure de sélectionner l'exécutable qui correspond à votre système.

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

Files							
Version	Operating System	Description	MD5 Sum	File Size	GPG	Sigstore	
Gzipped source tarball	Source release		1aea68575c0e97bc83ff8225977b0d46	26006589	SIG	CRT	SIG
XZ compressed source tarball	Source release		b8094f007b3a835ca3be6bdf8116cccc	19618696	SIG	CRT	SIG
macOS 64-bit universal2 installer	macOS	for macOS 10.9 and later	4c89649f6ca799ff29f1d1dffcb9393	40865361	SIG	CRT	SIG
Windows embeddable package (32-bit)	Windows		7e4de22bfe1e6d333b2c691ec2c1fcee	7615330	SIG	CRT	SIG
Windows embeddable package (64-bit)	Windows		7f90f8642c1b19cf02bce91a5f4f9263	8591256	SIG	CRT	SIG
Windows help file	Windows		643179390f5f5d9d6b1ad66355c795bb	9355326	SIG	CRT	SIG
Windows installer (32-bit)	Windows		58755d6906f825168999c83ce82315d7	27779240	SIG	CRT	SIG
Windows installer (64-bit)	Windows	Recommended	bfbe8467c7e3504f3800b0fe94d9a3e6	28953568	SIG	CRT	SIG

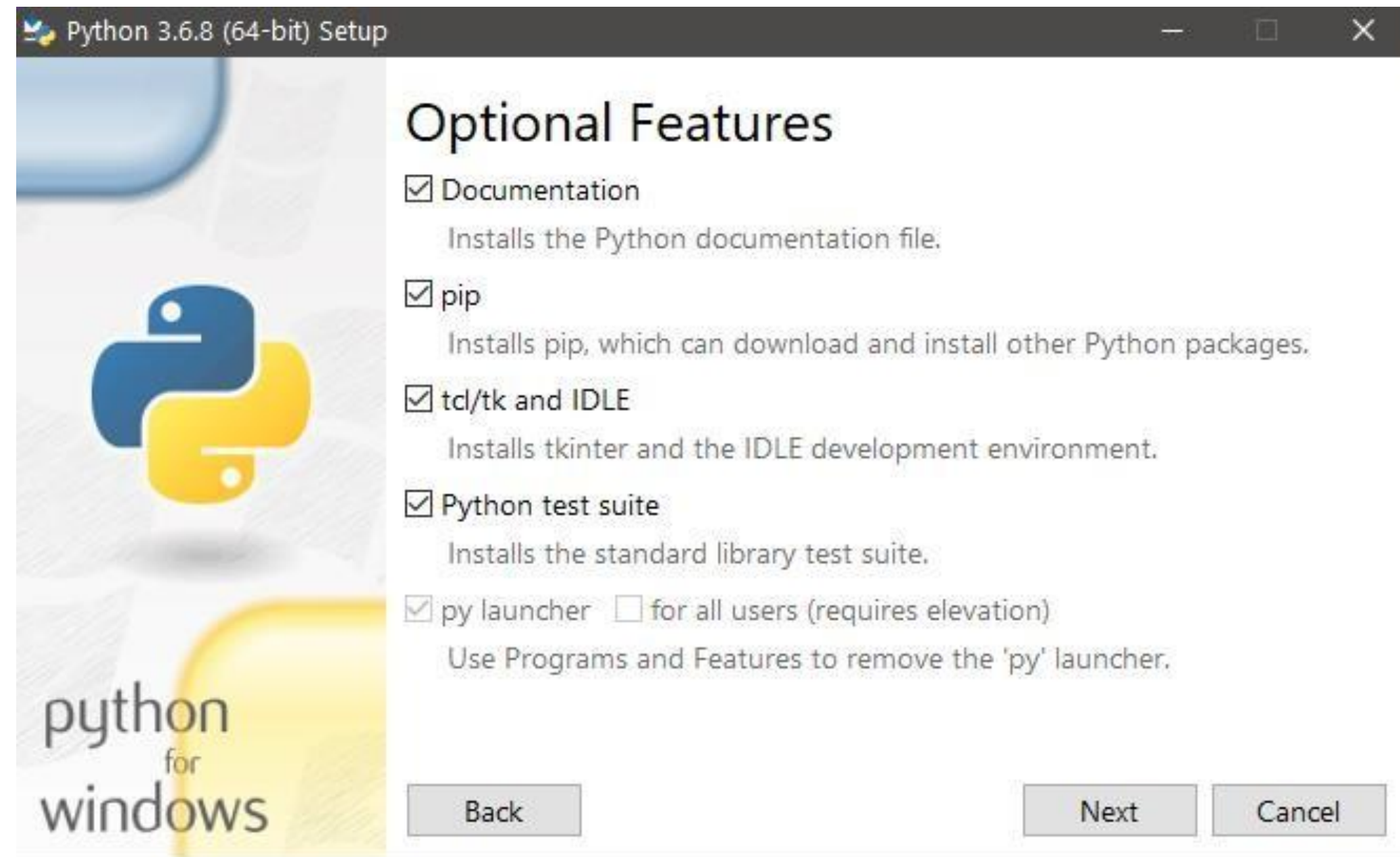
Installation de Python

A la première fenêtre choisir Customize Installation et cliquez sur suivante.



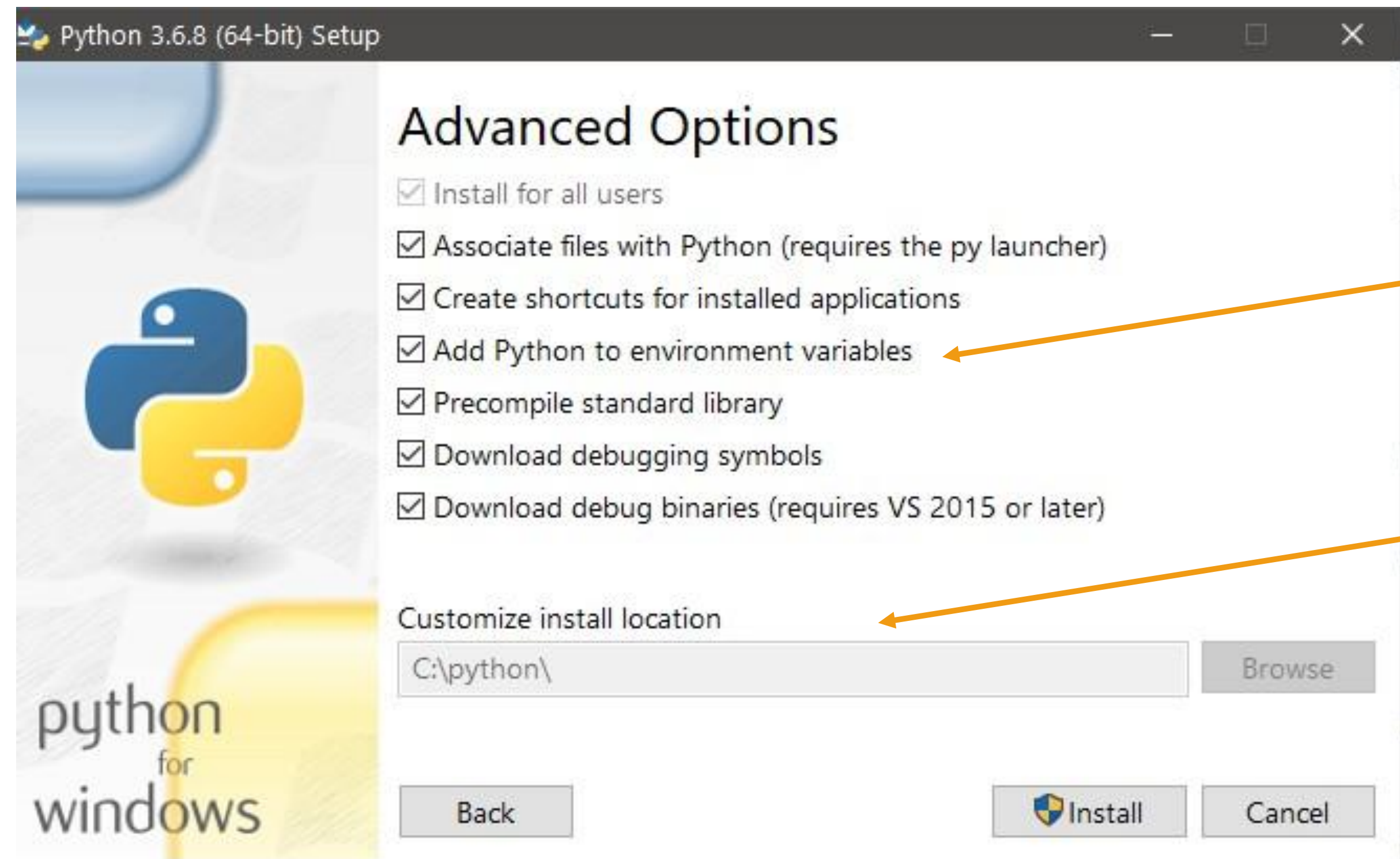
Installation de Python

A la fenêtre des options cocher toutes les cases puis suivant.



Installation de Python

La prochaine étape vous permettra de choisir dans quel dossier sera installé. Python et principalement de l'ajouter aux variables d'environnements. Cocher tout et cliquer sur le bouton « Install ». Attendre la fin de l'installation et cliquer sur le bouton « finish ».



Installation de Visual Studio Code

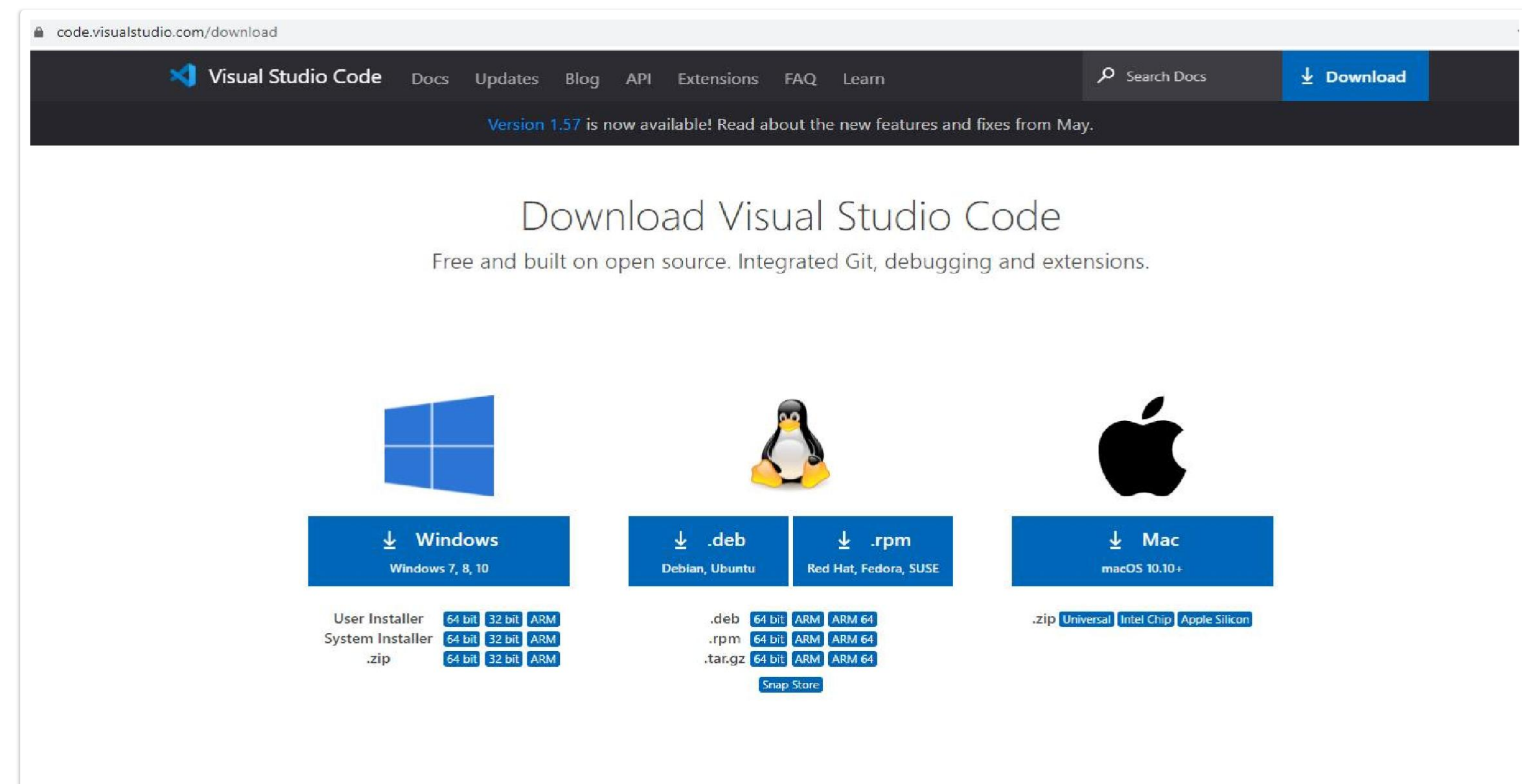
Installation de VS Code

Un IDE (Integrated Development Environment) est un regroupement d'outils utiles pour le développement d'applications (éditeur de code, débbugger, builder, indexation du code pour recherches « intelligentes » dans les projets...), rassemblés dans un logiciel unique. (Eclipse, Netbeans, Xcode, Pycharm, Wingware).

Python est avant tout un langage de script, et un simple éditeur de code avec quelques fonctions utiles peut suffire.

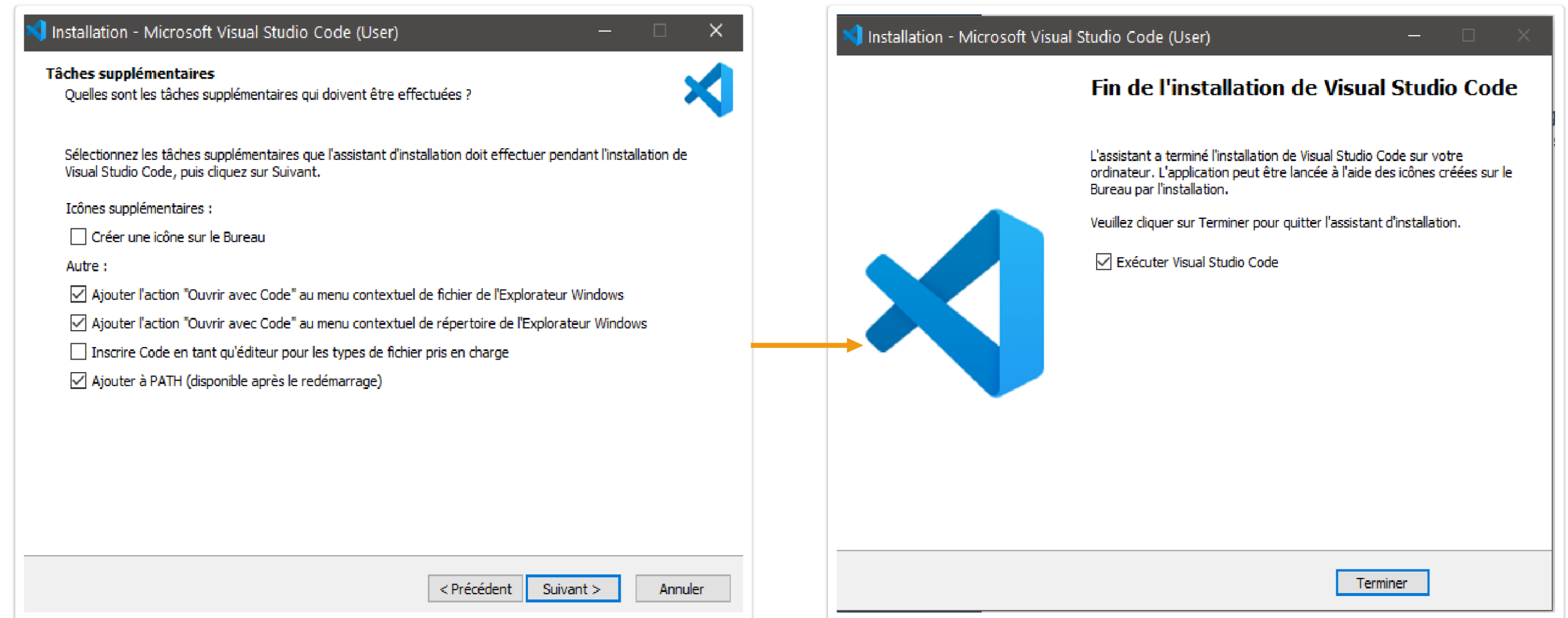
Cet éditeur, qui est gratuit, nous accompagnera tout au long de notre programmation en python.

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



Installation de VS Code

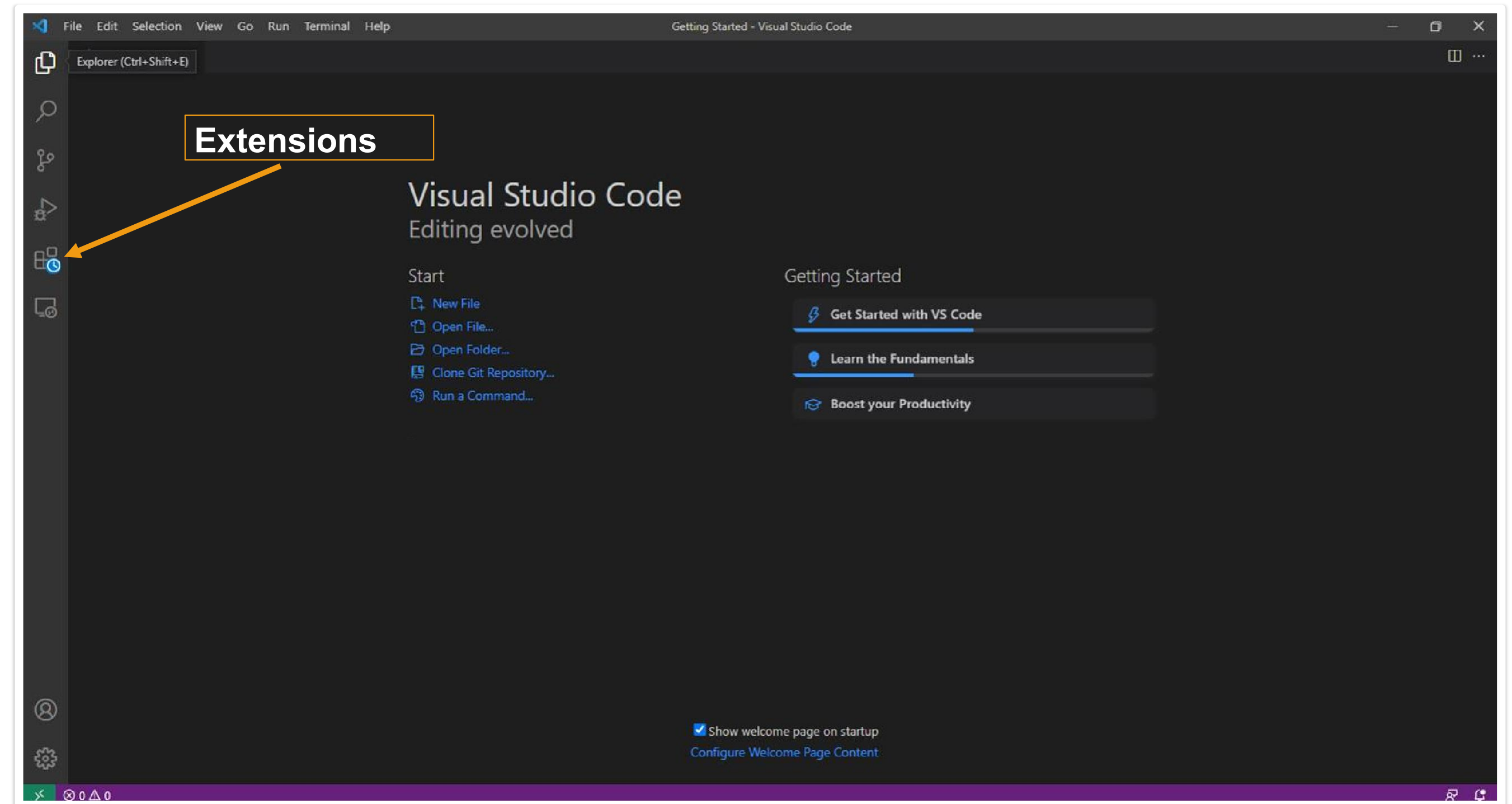
Le processus d'installation de VS Code est très simple et n'exigera pas trop de configurations. Une fois la tâche finie, vous pouvez déjà lancer l'éditeur.



Installation de VS Code

Afin de faciliter la saisie de nos codes Python, nous allons installer l'extension « Python ».

1. Placer votre curseur à gauche et cliquer sur l'option « Extensions ».



Installation de VS Code

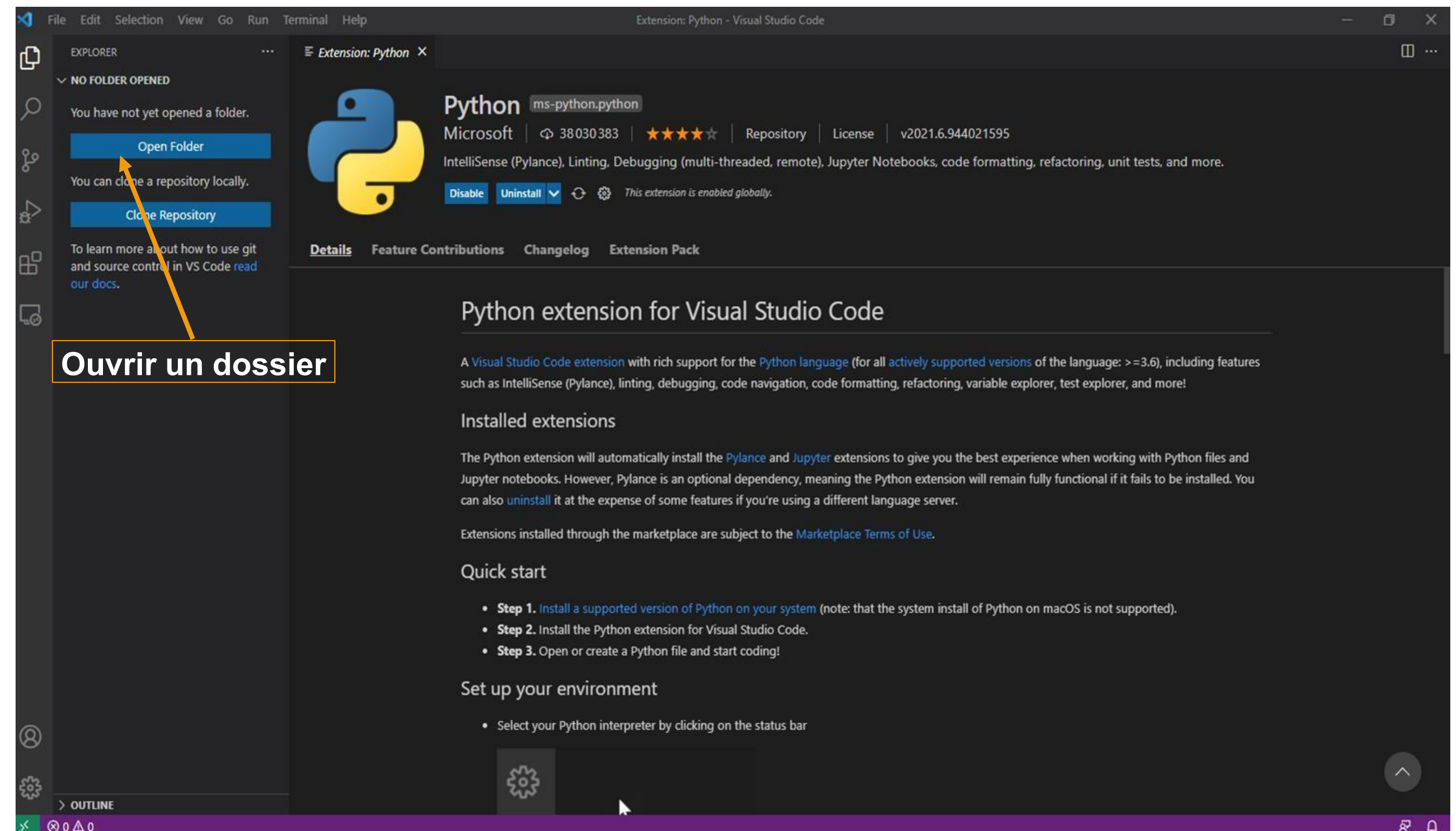
2. Dans la zone de recherche (coin supérieur gauche), vous cherchez et trouvez l'extension « Python » et ensuite vous l'installez.

Rechercher et sélectionner



Installation de VS Code

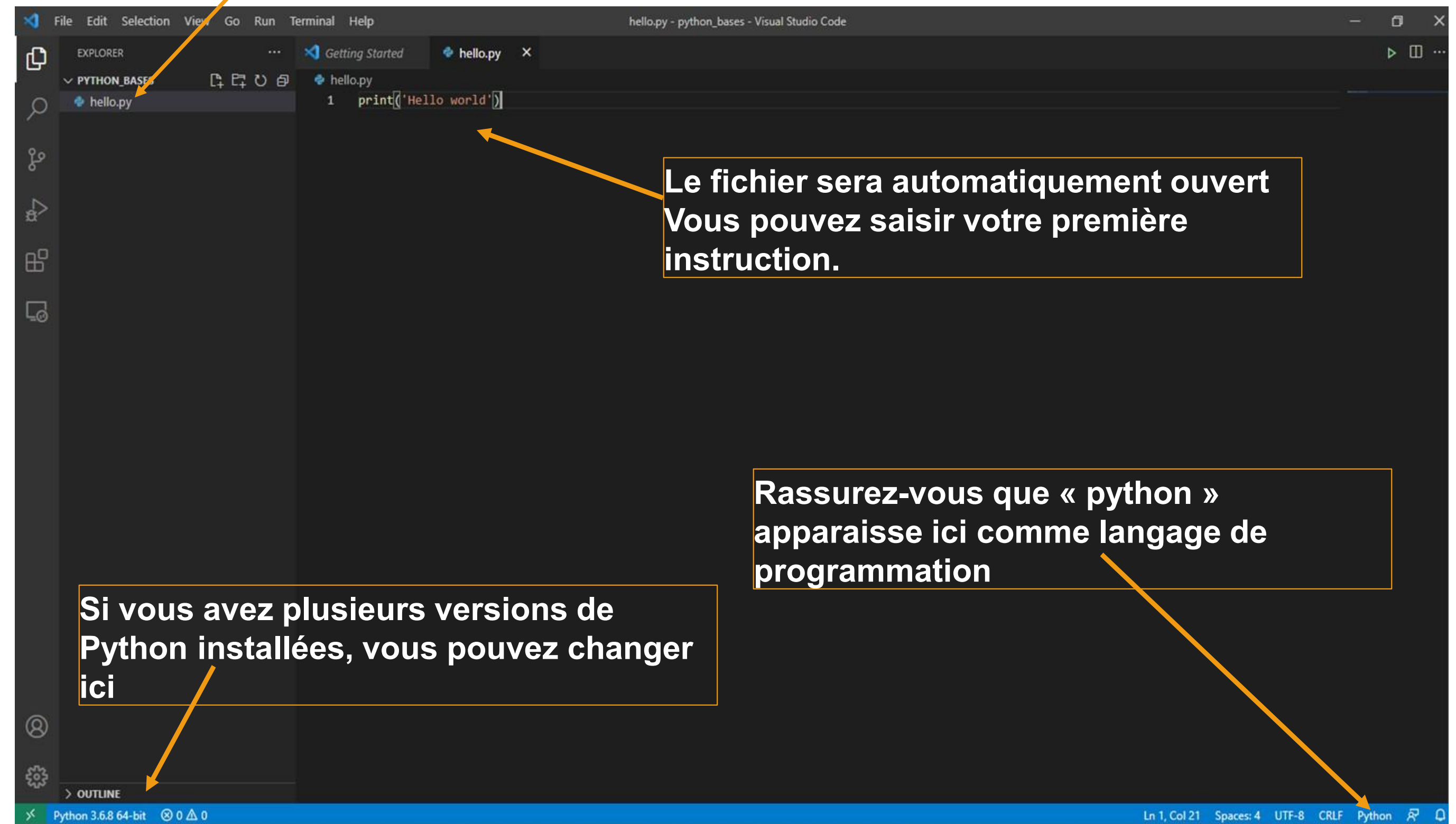
Nous allons créer notre premier fichier capable d'exécuter du code python. D'abord, il faudra sélectionner ou créer un dossier.



Installation de VS Code

Une fois le dossier ouvert, nous créons un fichier avec l'extension python « .py ». Ensuite, nous nous rassurons si le fichier est bien pris en charge par l'extension « python » que nous avons installé préalablement.

Ajouter un fichier avec le nom hello.py



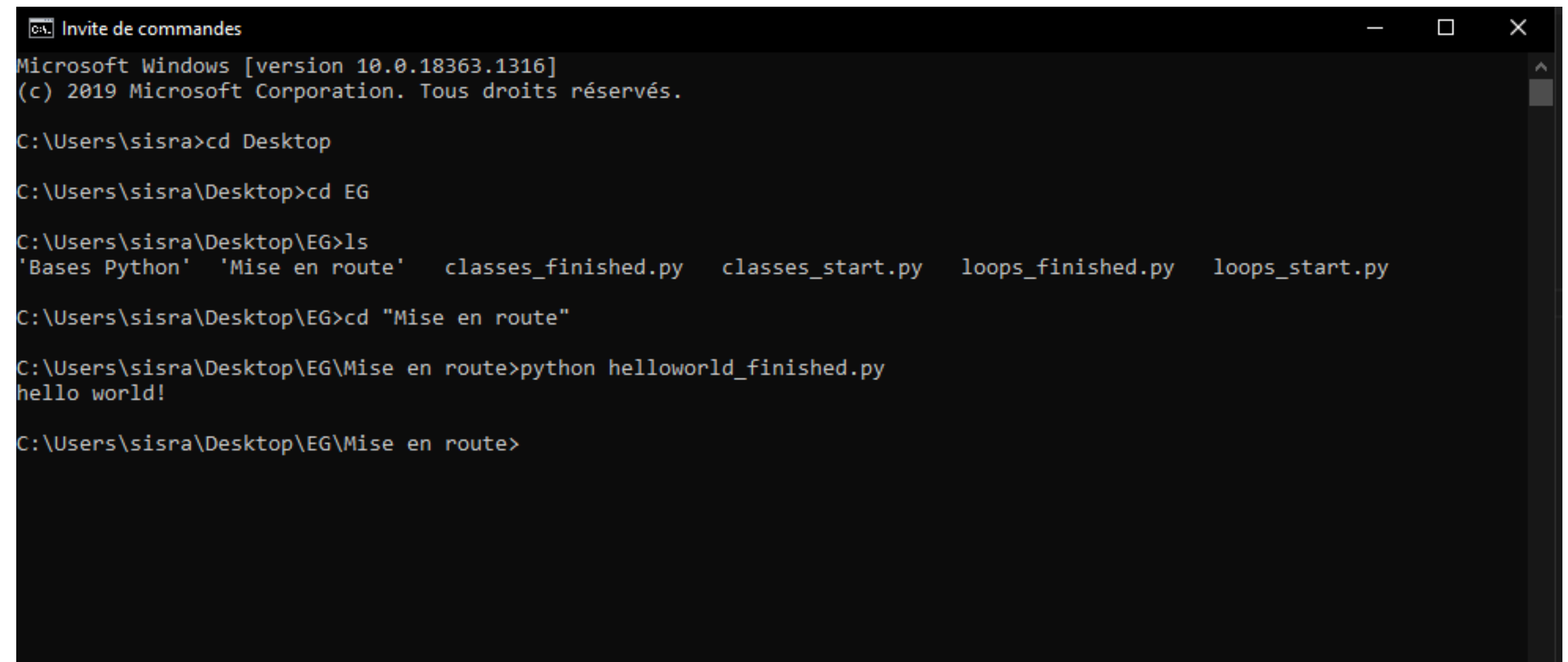
Comment exécuter Python ?

- Invite de commande
- Console Python
- VS Code

Invite de commande

Nous pouvons tester différentes commandes du langage python grâce à l'invite de commande.

En ouvrant le terminal sur notre machine, il suffit de se rendre dans le répertoire où se trouve notre fichier .py puis saisir la commande python suivi du nom du fichier.



```
C:\> Invite de commandes
Microsoft Windows [version 10.0.18363.1316]
(c) 2019 Microsoft Corporation. Tous droits réservés.

C:\Users\sisra>cd Desktop

C:\Users\sisra\Desktop>cd EG

C:\Users\sisra\Desktop\EG>ls
'Bases Python'  'Mise en route'  classes_finished.py  classes_start.py  loops_finished.py  loops_start.py

C:\Users\sisra\Desktop\EG>cd "Mise en route"

C:\Users\sisra\Desktop\EG\Mise en route>python helloworld_finished.py
hello world!

C:\Users\sisra\Desktop\EG\Mise en route>
```

Console Python

Nous pouvons tester différentes commandes du langage python grâce à la console. En ouvrant le terminal sur notre machine, il suffit de saisir la commande « python » pour accéder à la console Python.

Lancer la console Python

```
C:\>python
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world')
hello world
>>> |
```

Instruction python pour afficher du texte

Résultat de l'instruction

Installation d'Eclipse

Installation d'Eclipse

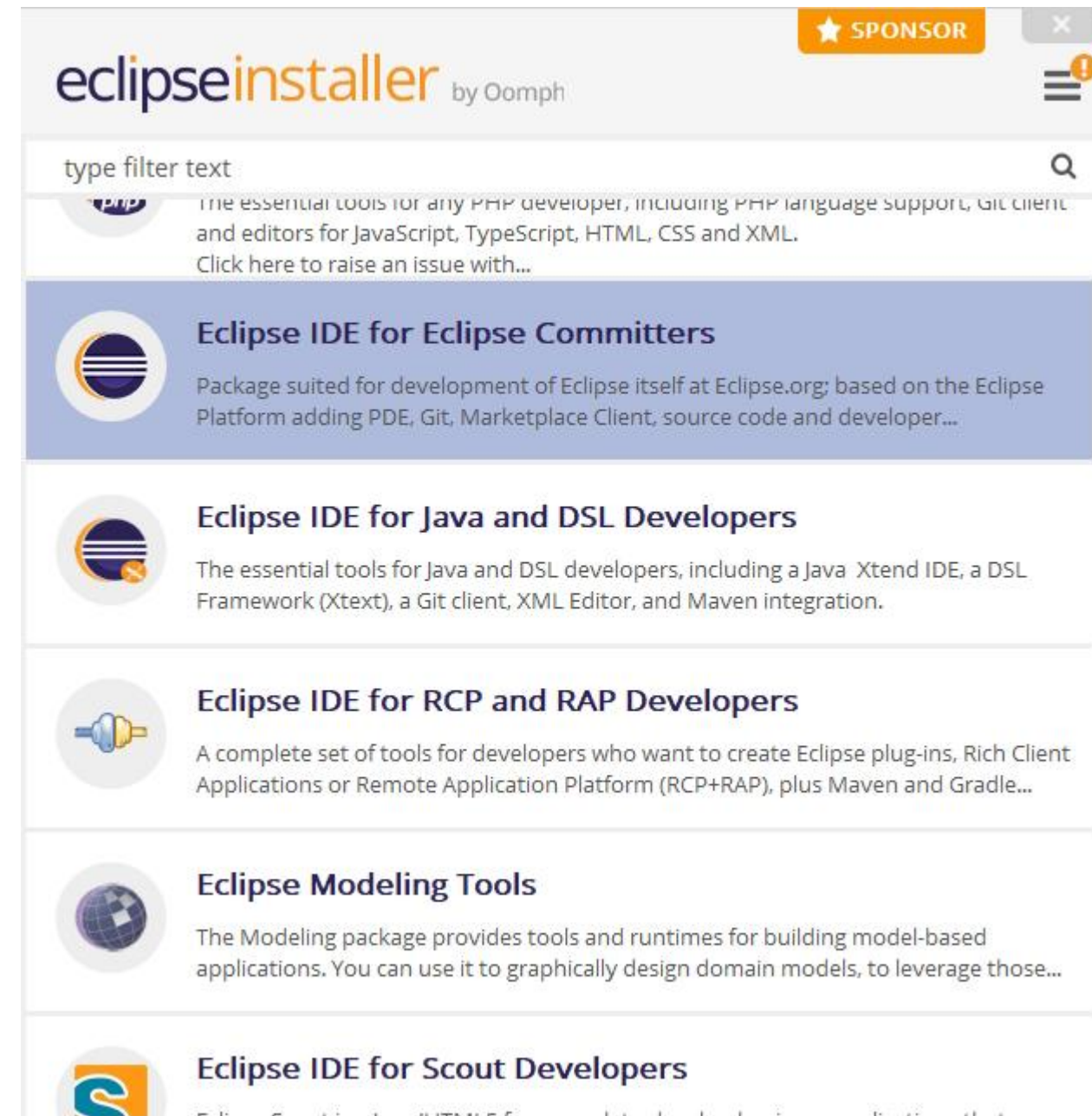
Télécharger « Eclipse installer » à cette adresse :

<https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2025-06/R/eclipse-inst-jre-win64.exe>

Lancer le fichier exécutable en mode administrateur.

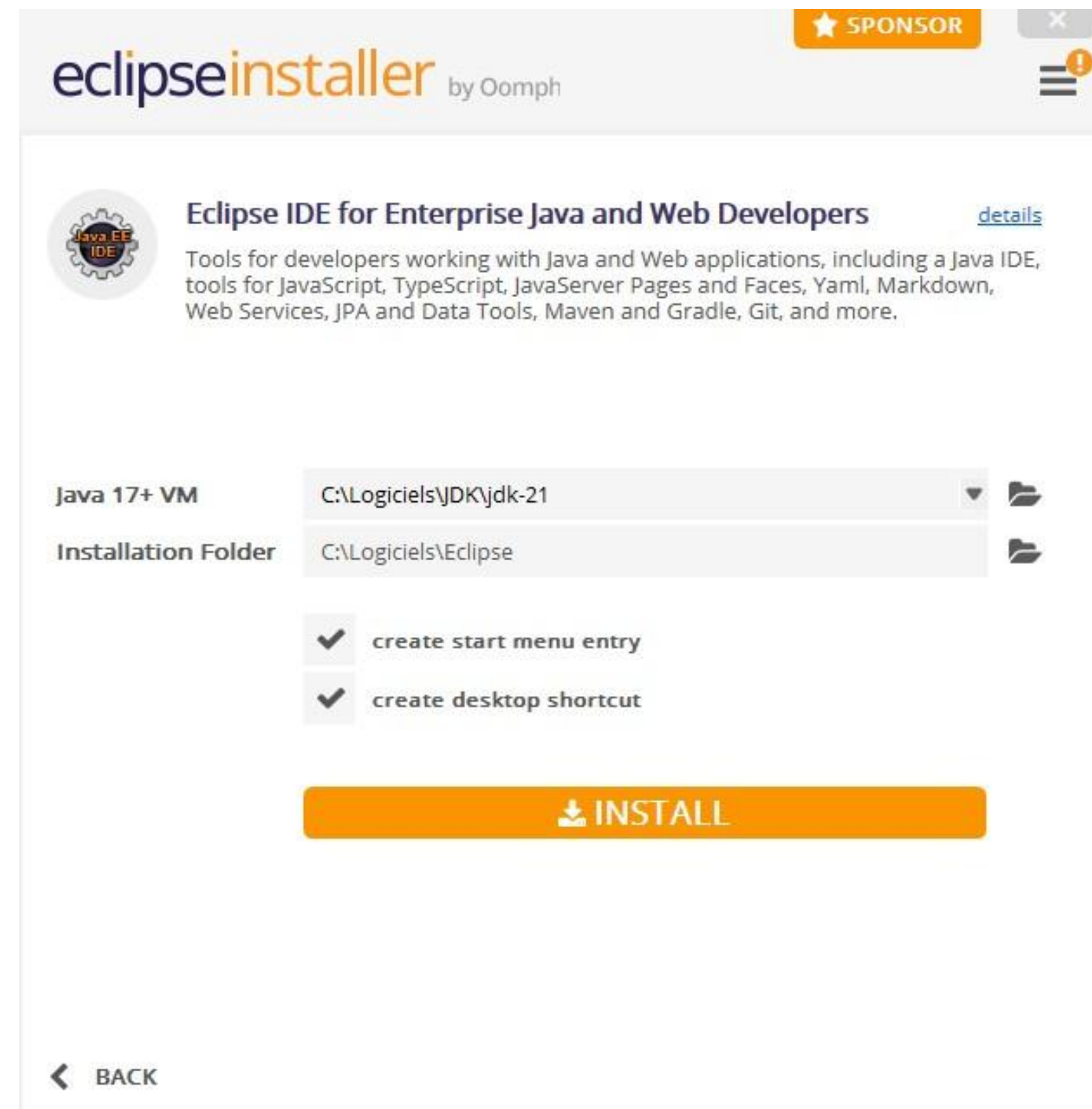
Installation d'Eclipse

Sélectionner Eclipse IDE for Enterprise Java and Web Developers.



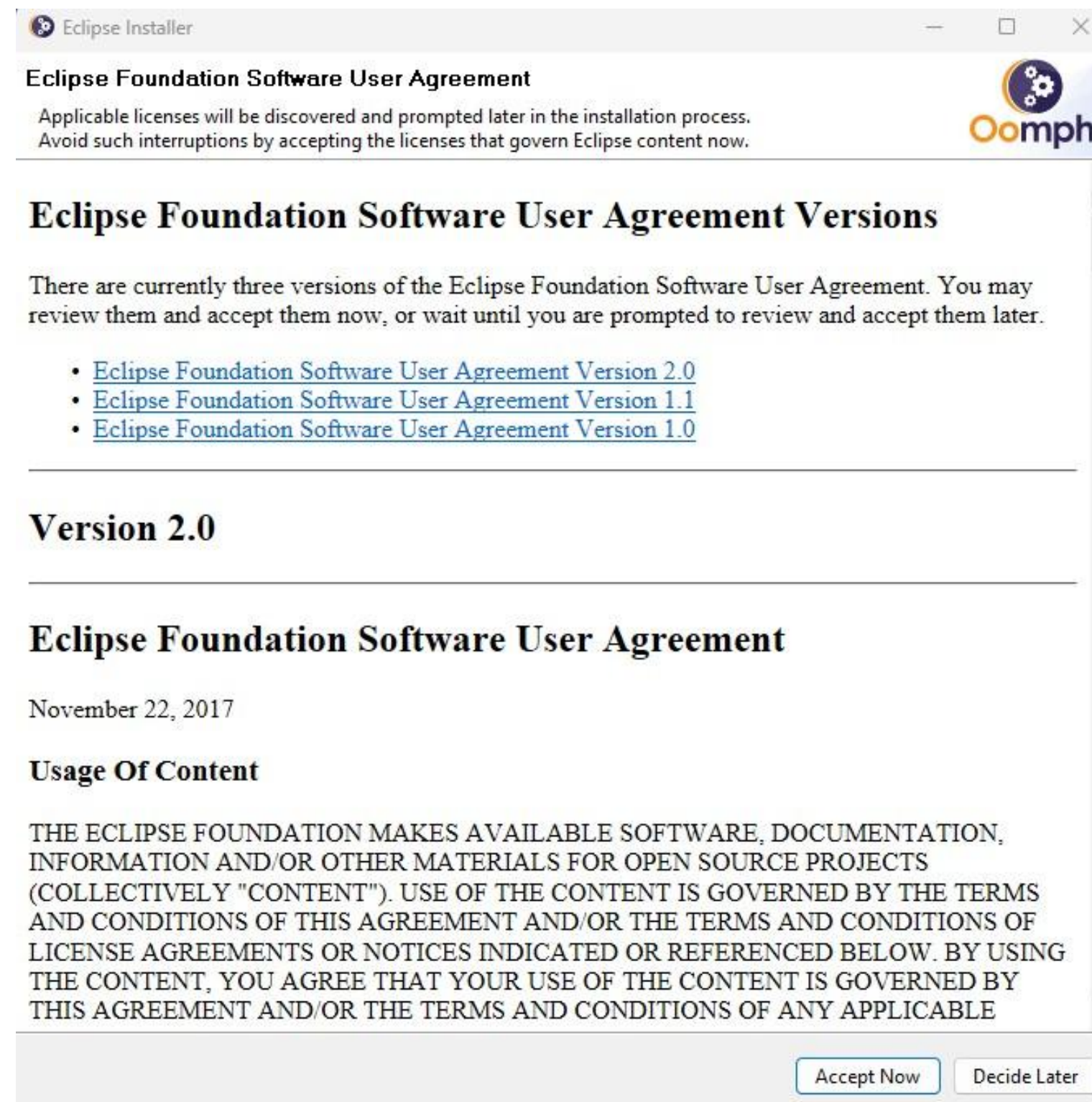
Installation d'Eclipse

Choisir l'emplacement de l'installation et cliquer sur « Install ». Si l'emplacement de la JVM n'est pas indiqué, on le fait.



Installation d'Eclipse

Cliquer sur le bouton « Accept Now » et attendre la fin de l'installation. Une fois l'installation finie, cliquer sur « Launch ».



Installation d'Eclipse

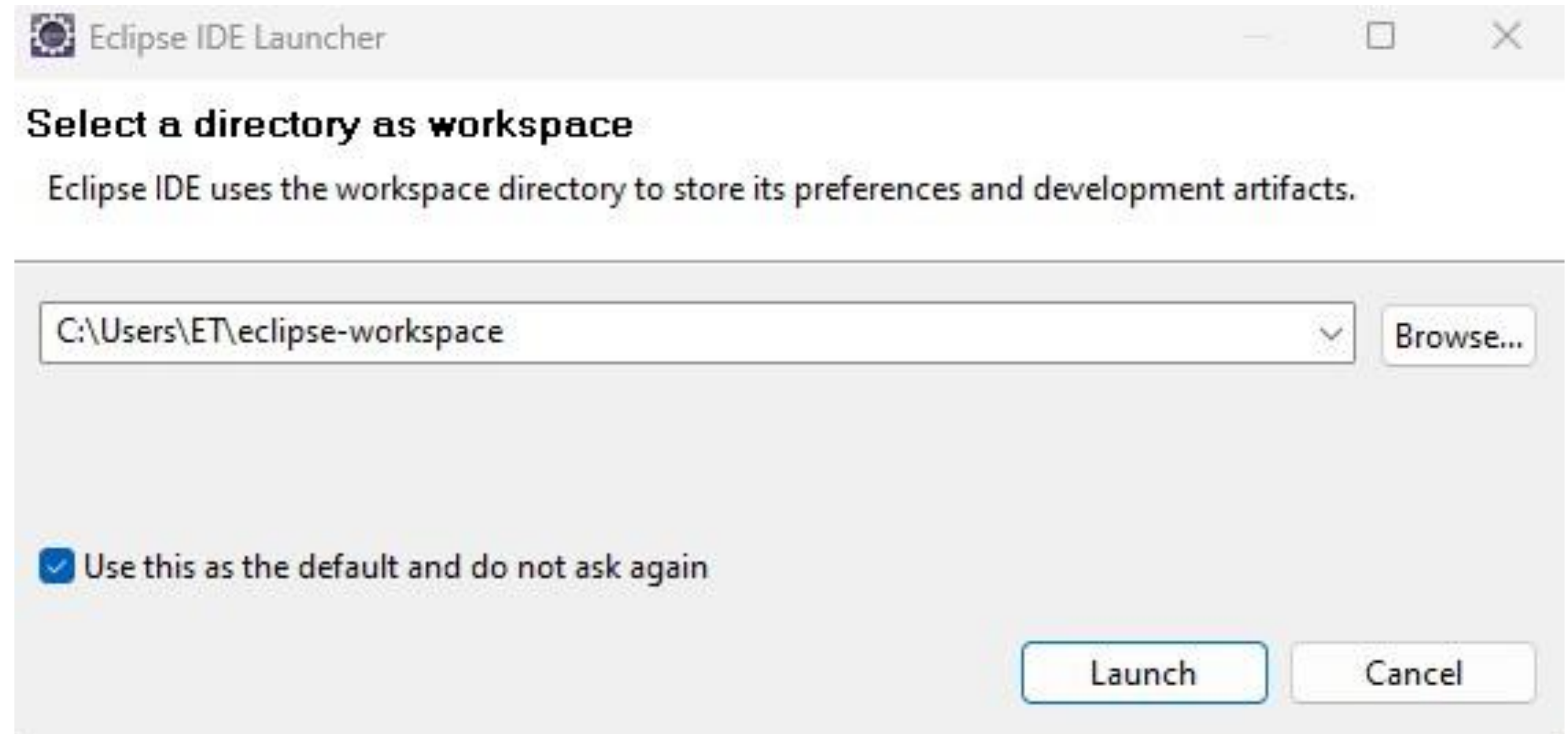
Aller dans le dossier d'Eclipse, et ouvrir le fichier « eclipse.ini » avec un éditeur de texte. Rajouter au début du fichier les deux lignes suivantes :

```
-vm  
« chemin complet du JDK »\bin\javaw.exe
```

Cliquer avec le bouton droit sur l'icône d'Eclipse. Aller dans l'onglet « Propriété » et cocher la case « Exécuter en tant qu'administrateur ».

Installation d'Eclipse

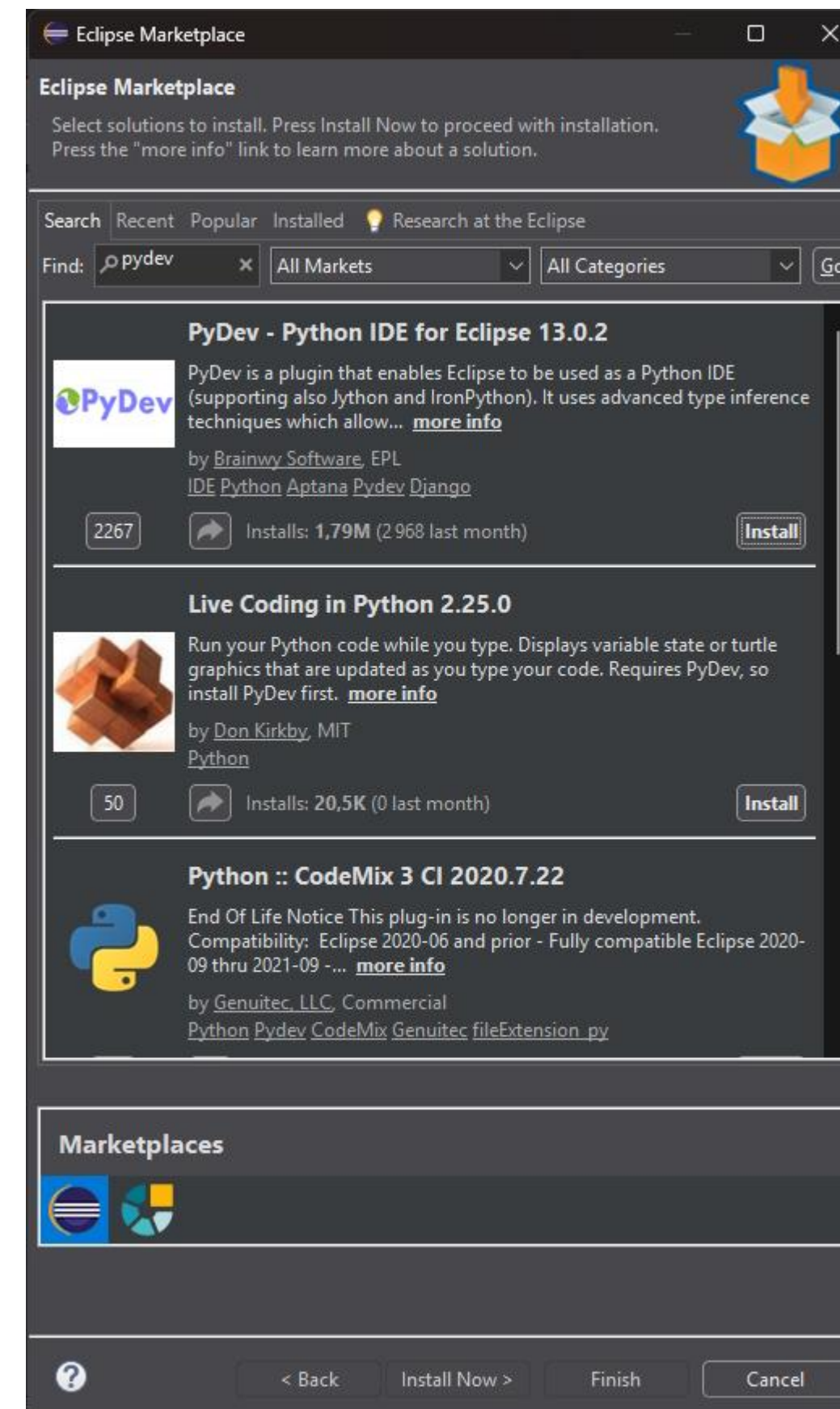
Au premier démarrage d'Eclipse, une fenêtre apparaît et demande de saisir un chemin pour définir le dossier de travail. Laisser la case cochée pour ne pas avoir à recommencer à chaque démarrage d'Eclipse.



Installation d'Eclipse

- Installation du plugin PyDev

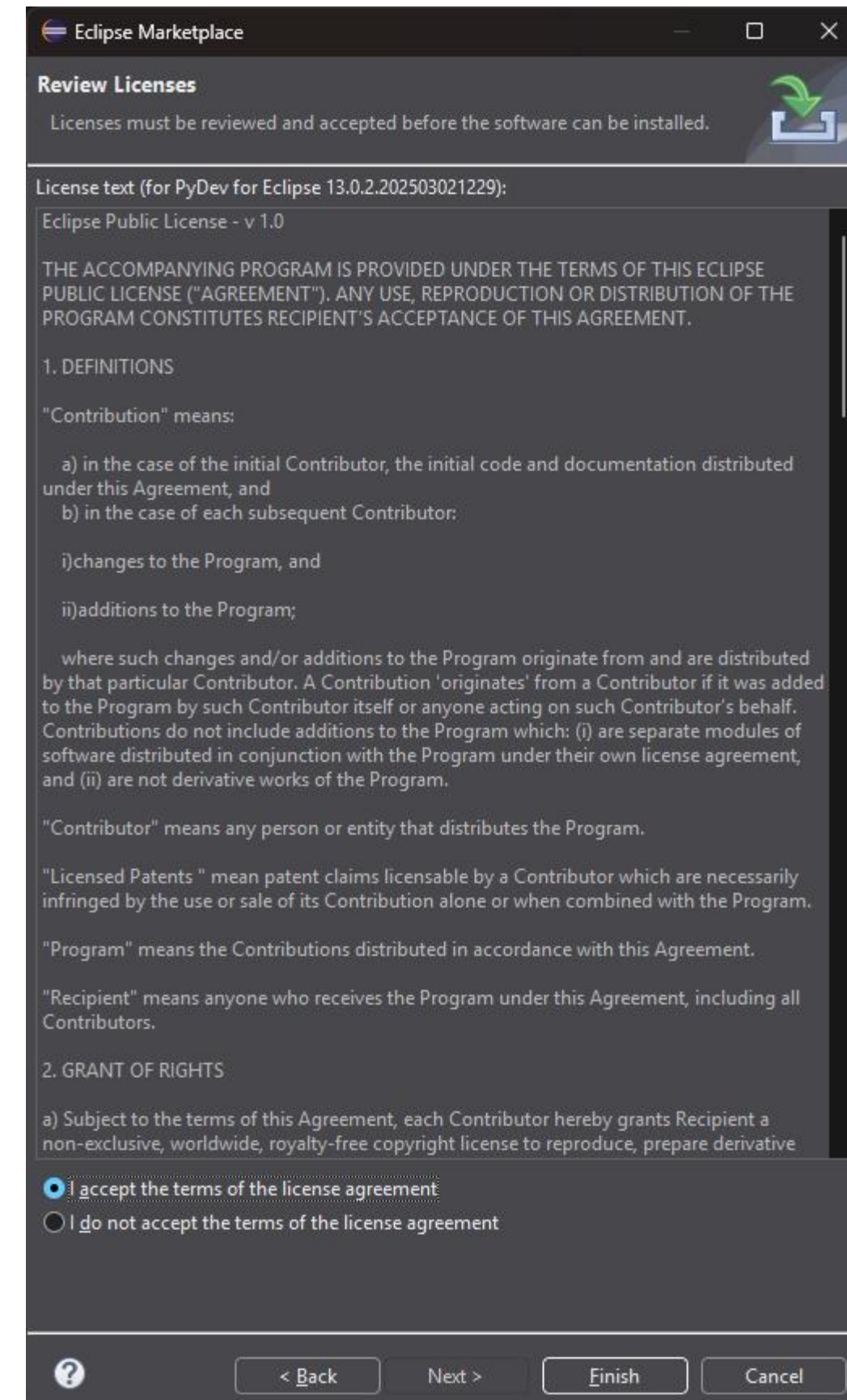
Ouvrir Eclipse, allez dans le menu Help > Eclipse Marketplace. Dans la barre de recherche, taper PyDev. Cliquer sur « Install » à côté de PyDev.



Installation d'Eclipse

- Installation du plugin PyDev

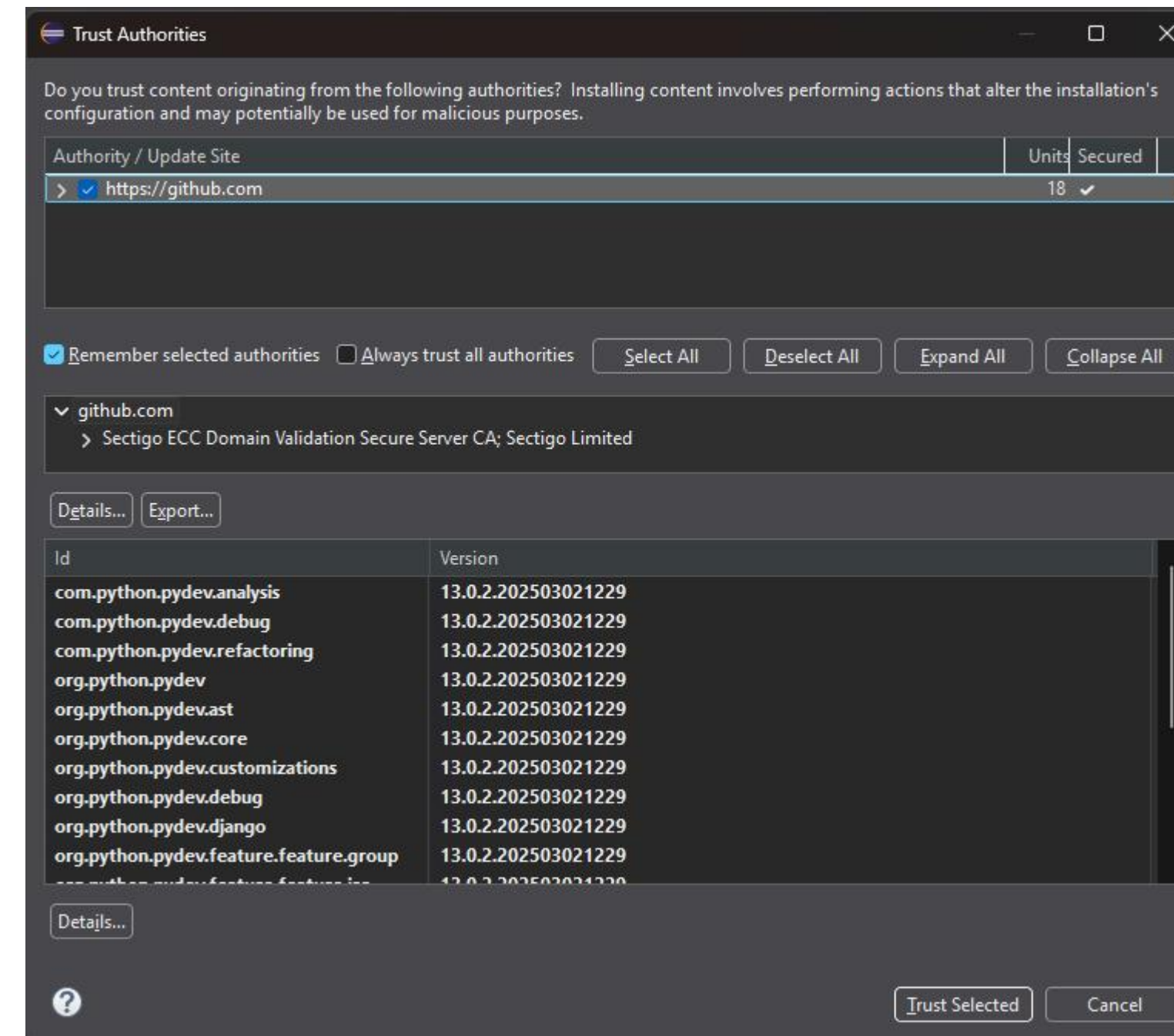
Suivre les instructions pour terminer l'installation.



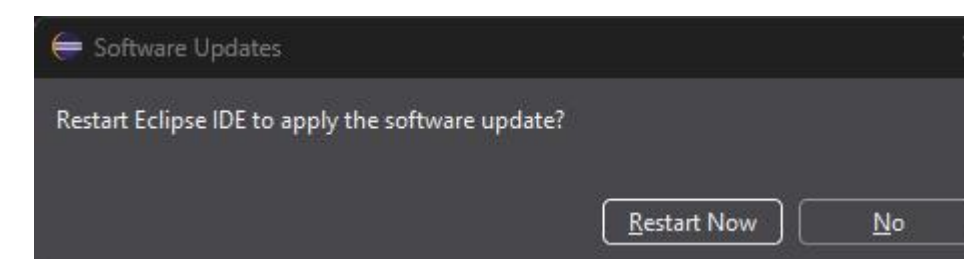
Installation d'Eclipse

- Installation du plugin PyDev

Lorsque la fenêtre Trust Authorities s'ouvre, sélectionnez tout et cliquez sur « Trust Selected »



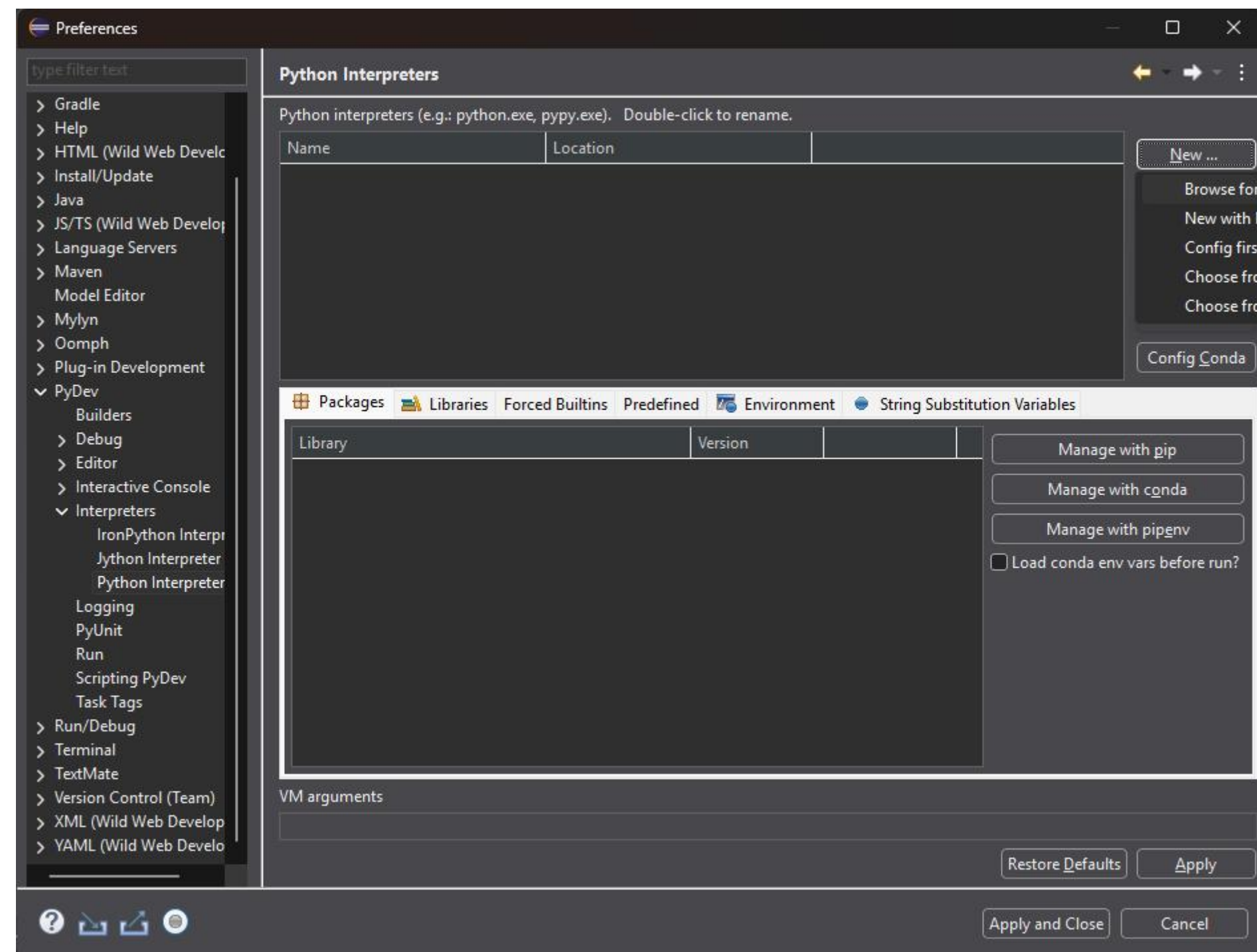
puis redémarrer Eclipse quand demandé.



Installation d'Eclipse

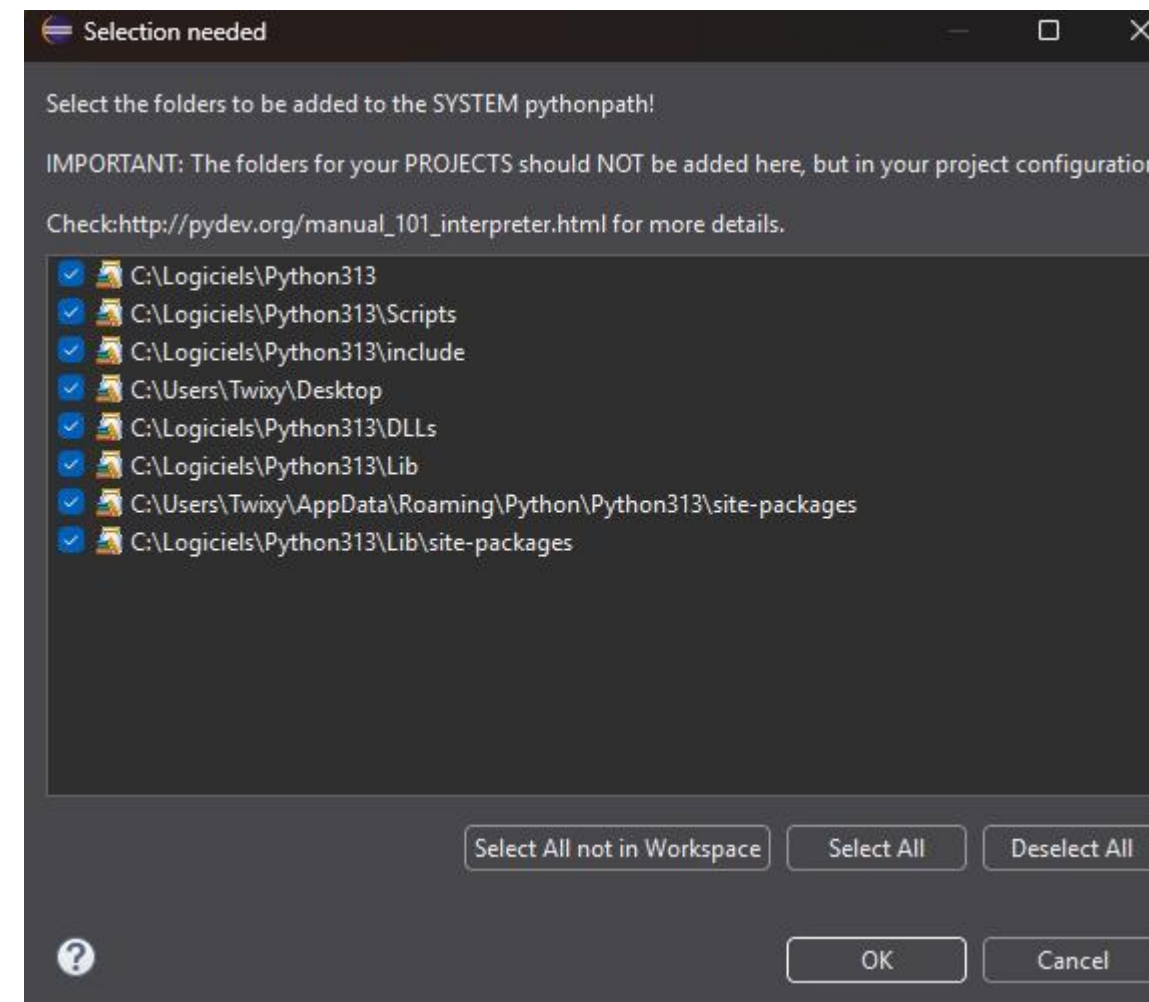
• Configuration de l'interpréteur Python

Aller dans Window > Preferences. Naviguer vers PyDev > Interpreters > Python Interpreter. Cliquer sur « New.... » et sélectionnez le chemin vers l'interpréteur Python installé. Par exemple :
Sous Windows : C:\Python313\python.exe - Sous macOS/Linux : /usr/bin/python3.



Installation d'Eclipse

- Configuration de l'interpréteur Python

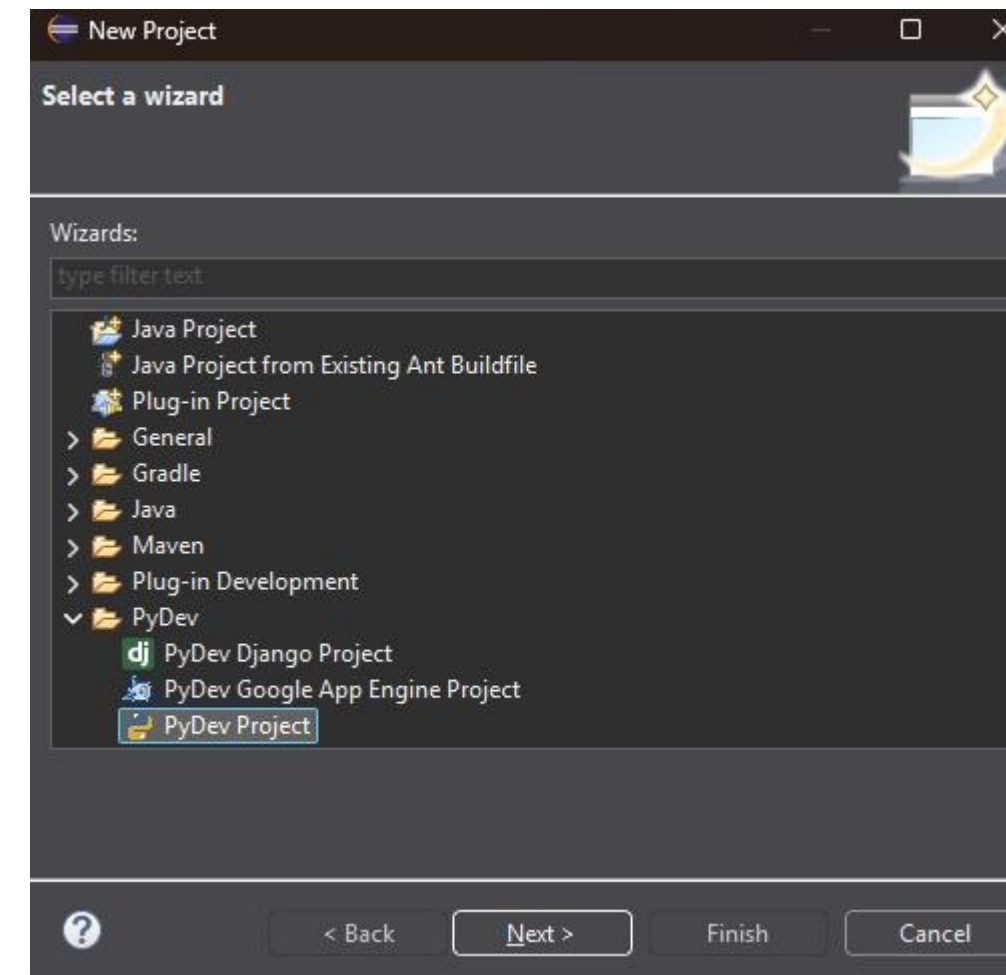


Valider et laisser Eclipse détecter les packages installés puis cliquer sur « apply and close ».

Installation d'Eclipse

- Création d'un projet Python

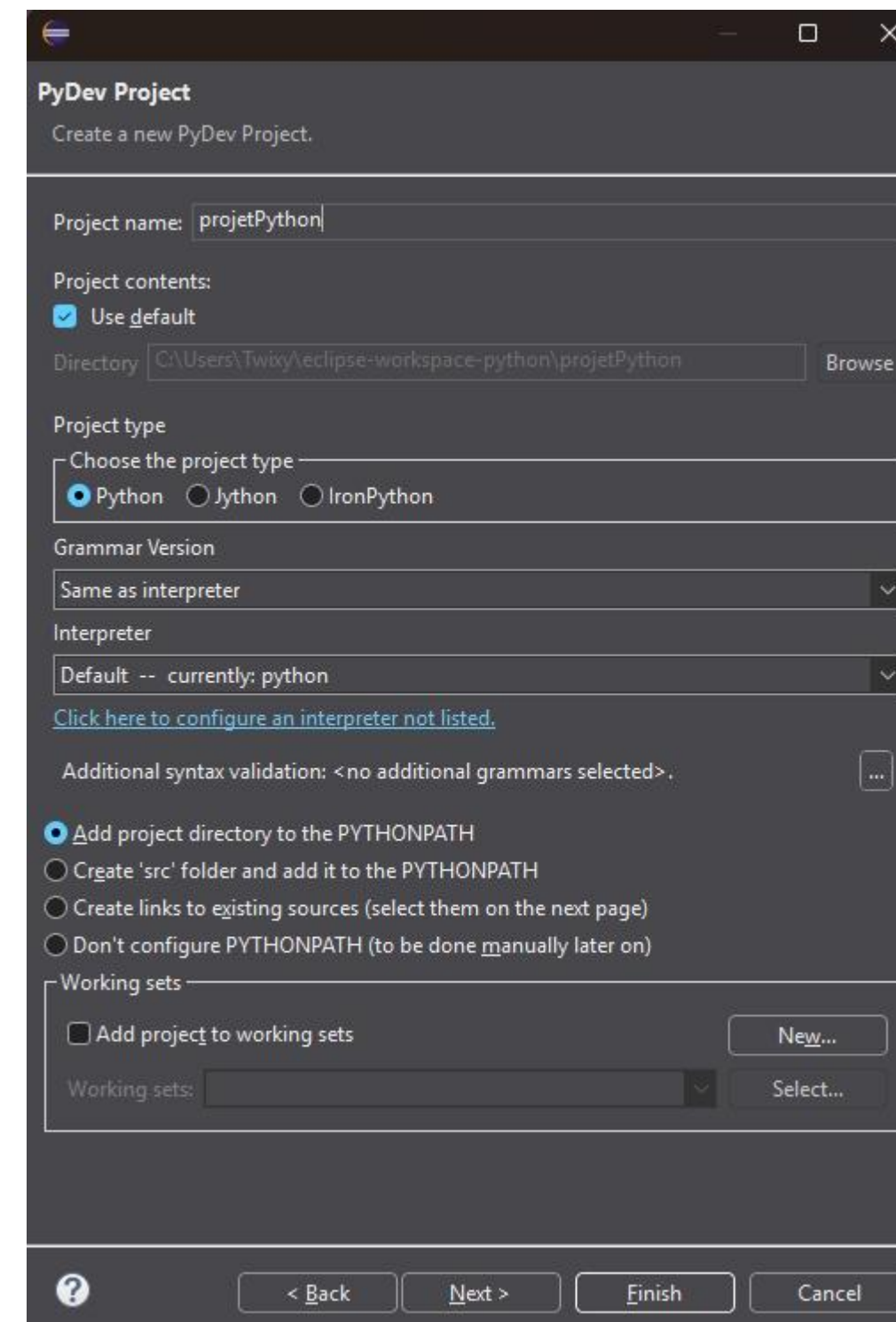
Aller dans File > New > Project..., sélectionner PyDev Project.



Installation d'Eclipse

- Création d'un projet Python

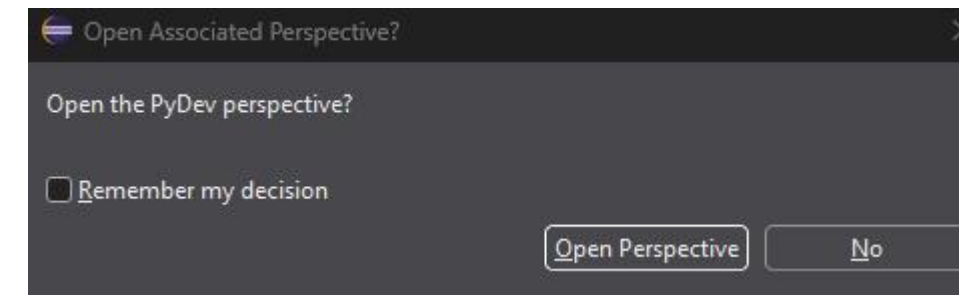
Donner un nom au projet. Sélectionner le type : Python et la version de l'interpréteur. Le reste peut rester par défaut. Cliquer sur Finish.



Installation d'Eclipse

- Création d'un projet Python

Cliquer sur « open perspectives » pour obtenir une vue avec PyDev pour Python.

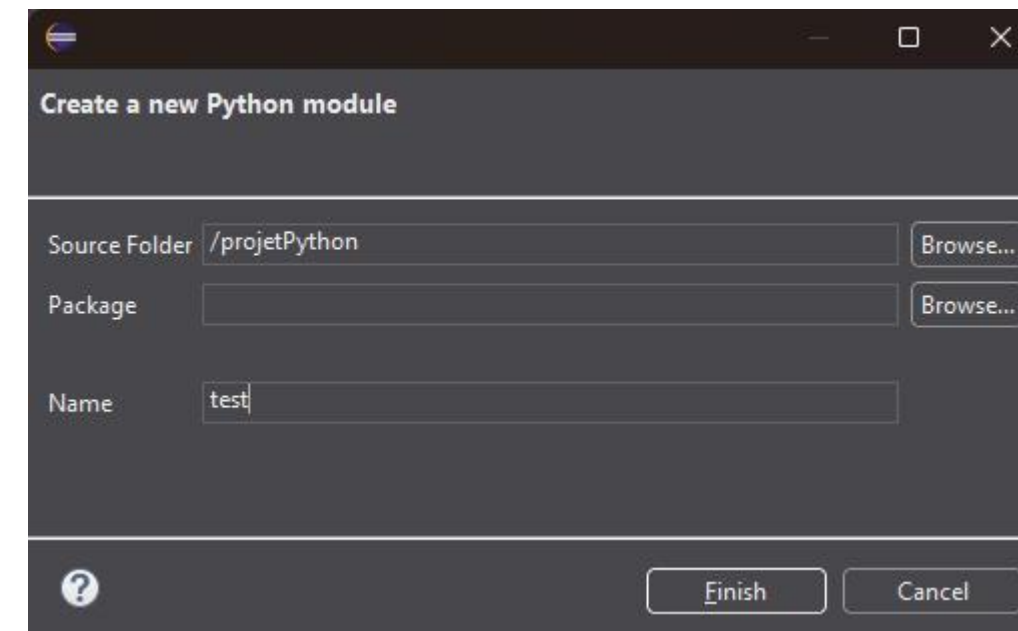


Si on a cliqué sur « No » par erreur, on peut changer de perspective à tout moment :
Aller dans le menu Window > Perspective > Open Perspective > Other... et sélectionner PyDev
puis cliquer sur Open.

Installation d'Eclipse

- Création d'un fichier Python

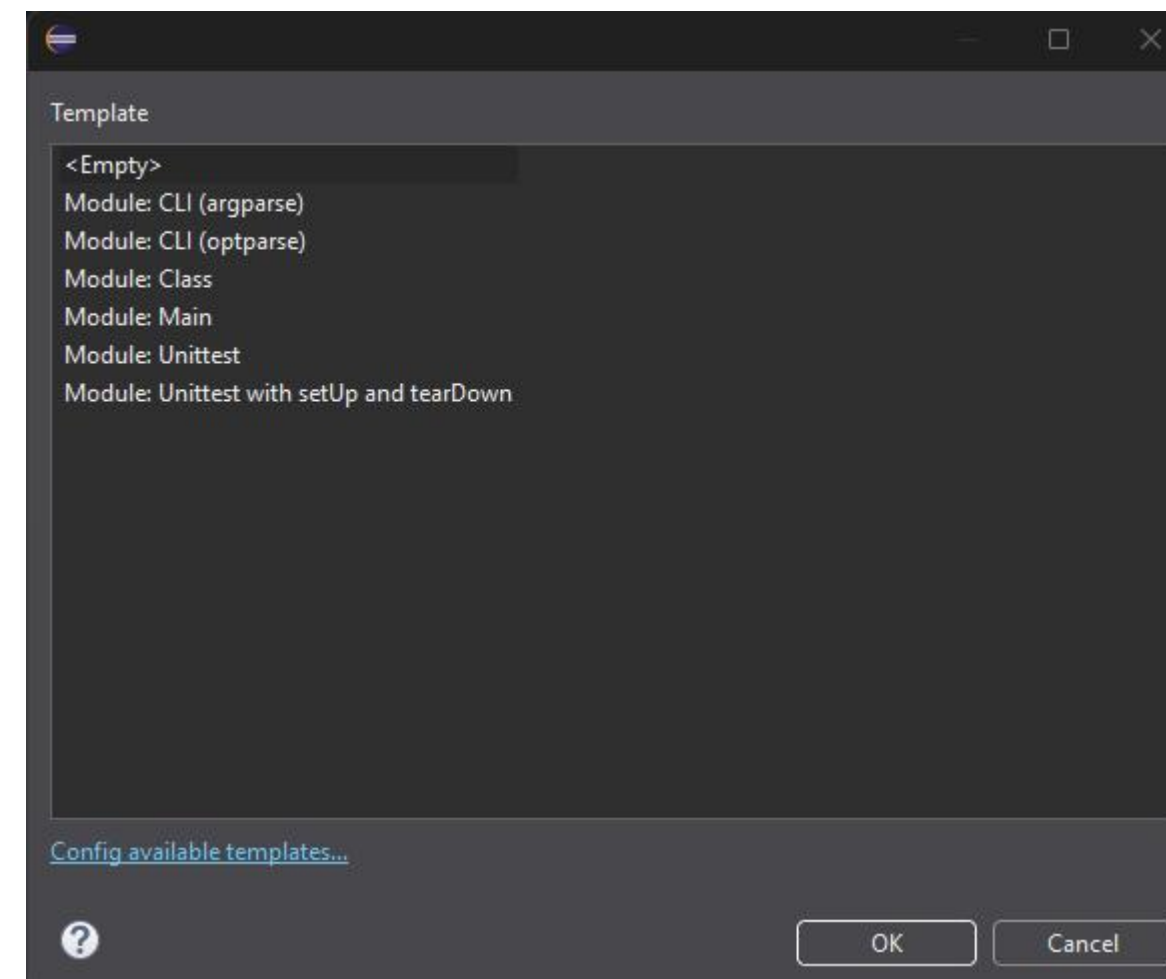
Clic droit sur le projet > New > PyDev Module. Donner un nom au fichier (ex. : test.py).



Installation d'Eclipse

- **Création d'un fichier Python**

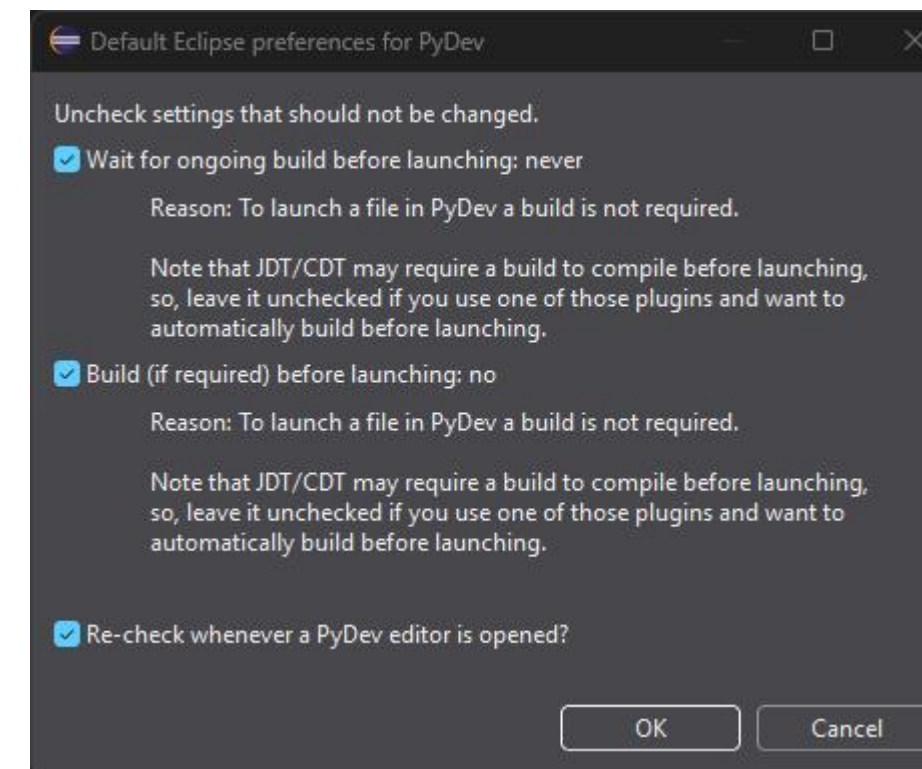
Une fenêtre s'ouvre et demande le modèle que l'on veut. Un squelette peut être généré automatiquement, selon ce choix.



Installation d'Eclipse

- Création d'un fichier Python

Une autre fenêtre demande les préférences de PyDev par défaut, laissez tout cocher et cliquer sur « Ok ».



Étape 6 : Exécuter ton code Python

Clic droit sur le fichier > Run As > Python Run.

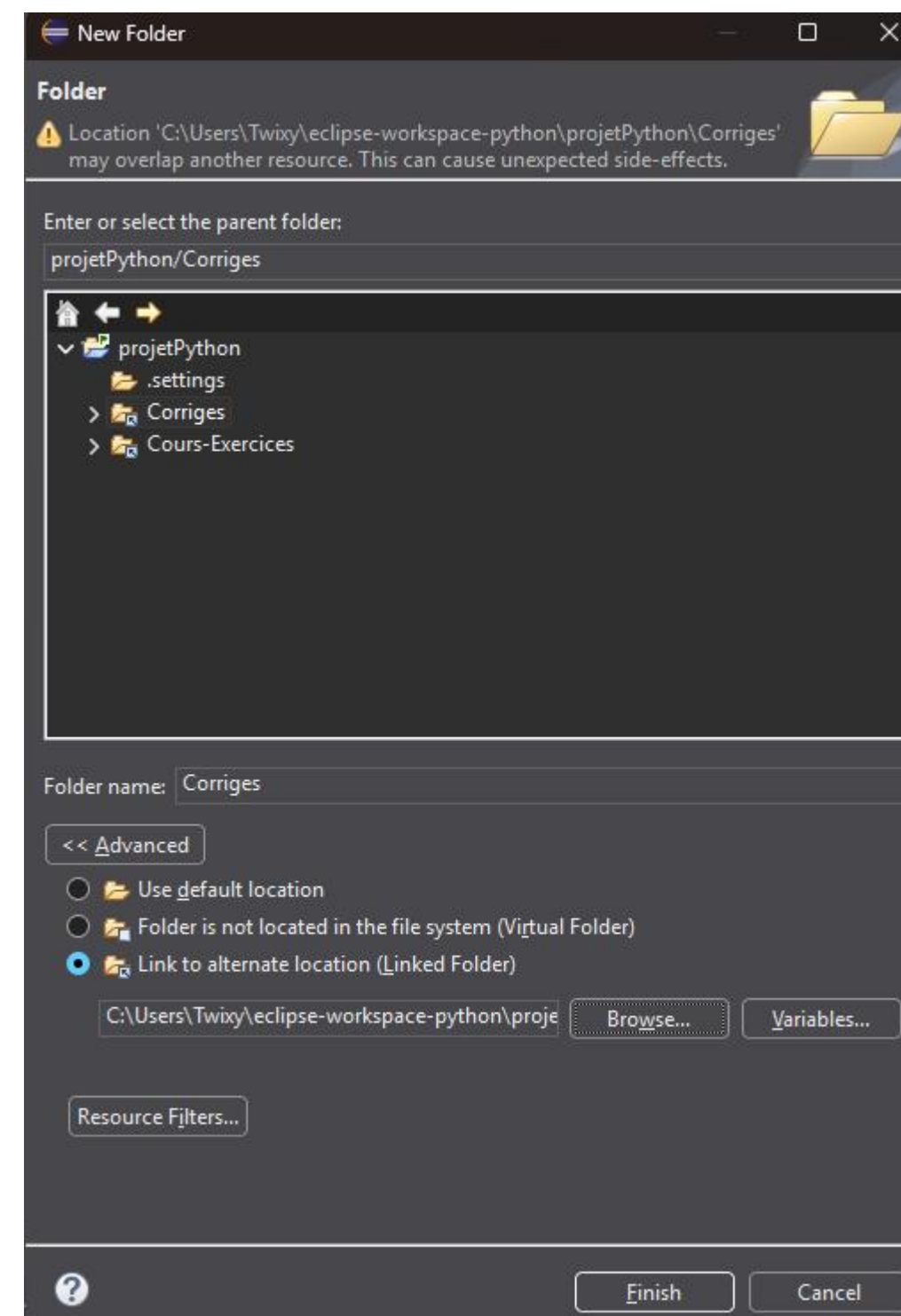
Tu verras la sortie dans la console Eclipse.

Installation d'Eclipse

- **Exécuter le code Python**

Clic droit sur le fichier > Run As > Python Run. La sortie se fera dans la console d'Eclipse.

- **Ajouter un dossier existant dans un projet**



Copier le dossier dans le projet situé dans le workspace défini au début de l'installation.

Dans Eclipse, sélectionner File > New > Folder et cliquer ensuite sur le bouton « Advanced >> ».

Choisir « Link to alternate location », cliquer sur « Browse » et sélectionner le dossier à ajouter.

Valider et cliquant sur « Finish ». Le dossier doit apparaître dans l'arborescence de la fenêtre du projet.

Installation d'Eclipse

- Dictionnaire français

<https://www.pallier.org/extra/liste.de.mots.francais.frgut.txt>

Cliquer sur le lien ci-dessus. Faire « Ctrl+S » pour enregistrer le fichier, le nommer « fr.dic ».
Créer un sous dossier dans eclipse et copier le fichier dedans (exemple : .\dictionaries\fr)

Ensuite, dans Eclipse, aller dans Window > Preferences > General > Editors > Text Editors > Spelling.

Cliquer sur le bouton « Browse... » à côté de User defined dictionary, sélectionner le fichier fr.dic qui as été téléchargé et enregistré.

Cliquer sur « Apply and Close » et redémarrer eclipse.

Les textes en français dans le code devraient être reconnu.

Installation de PyCharm

Installation de PyCharm

PyCharm est un IDE très utilisé par les développeurs en Python car il est spécialisé pour dans ce langage de programmation. Il existe 2 versions de PyCharm. La version « Community » (pure Python) gratuite et suffisante pour la plupart des développeurs. Et la version « Professional », payante, contenant plus d'option (support Python Web, HTML, JS et SQL).

Lien de téléchargement incluant les 2 versions :

<https://www.jetbrains.com/pycharm/download/>

Lancer le programme d'installation en mode administrateur. Choisir le dossier par défaut. Cocher la case « Add bin folder to the PATH » et la case « Create Desktop Shortcut » si on veut un raccourci sur le bureau. Cliquer sur « Install ». Attendre la fin de l'installation et cliquer sur « Finish ».

La version « Pro » est installé par défaut. Elle passera en version « Community » au bout d'un mois si aucune licence n'est indiquée. Il s'agit d'un plugin qui peut être désactivé avant la fin du mois.

Pour installer ou désinstaller un plugin dans PyCharm, il faut aller dans File -> Settings -> Plugins

Si on veut donner les couleurs syntaxiques du code Python correspondant à VSCode, on peut installer le plugin « VS Code Theme » ou un autre. Il existe beaucoup de plugin pour PyCharm. Redémarrer PyCharm pour que le plugin soit pris en compte. Puis aller dans File -> Settings -> Appearance & Behavior -> Appearance et choisir dans la liste le theme installer.

Premier pas

Utilisons la console Python

- Calculer avec Python

```
>>> 15 + 4      >>> 5 ** 2      >>> 5 % 3
>>> 2 - 9       >>> 20 / 3     >>> a @ b (si numpy installé ou class définie)
>>> 4 * 3       >>> 5 // 3
```

- Données, Variables, Affectation

```
>>> a = 15
>>> nom = "Kaiser Söze"
```

- Affichage de la valeur d'une variable

```
>>> print(a)
```

- Affectation multiple

```
>>> x = y = 7      # Affectation parallèle
>>> a, b = 5, 18   # Affectation multiple
```

- Composition

```
>>> print(17 + 3)  # Ceci est un affichage
>>> print("somme est de" , 15 * 3 + 4)
```

- Opération bit à bit

```
>>> 0b0001 & 0b1010  >>> 0b0001 | 0b1010  (& = et,      | = ou)
>>> 0b0001 ^ 0b1010  >>> ~0b1010          (^ = ou exclusif, ~ = non)
>>> 0b0001 >> n      (>> décalage à droite de n bits)
>>> 0b0001 << n      (<< décalage à gauche de n bits)
```

Petites choses diverses à savoir pour débiter en Python

- **Commentaire en python**

Les commentaires en Python commencent avec un caractère dièse, #, et s'étendent jusqu'à la fin de la ligne. Un commentaire peut apparaître au début d'une ligne ou à la suite d'un espace ou de code, mais pas à l'intérieur d'une chaîne de caractères littérale.

Il y a aussi les commentaires multi lignes commençant et finissant par triple quotes simples ou doubles. On les appelle les docstrings.

- **Quelques mots sur l'encodage des sources et UTF-8**

Pour pouvoir utiliser des caractères « spéciaux » (accents notamment) dans vos programmes (même dans les commentaires), vous devez dire à Python, de manière explicite, que vous souhaitez utiliser le codage de caractères UTF-8.

```
# -*- coding: UTF-8 -*-
```

- **if `__name__` == '`__main__`': # kesako ?**

`main()` n'existe pas en Python, comme on peut le trouver en C ou java par exemple.

Il y a néanmoins un cas de figure où le fait de ne pas avoir ce genre de fonction peut être problématique : quand on inclut un module dans un autre, Python réalise un import de tout le contenu du module. Le problème, c'est que si on y place des instructions à l'extérieur de toute fonction ou méthode, elles seront exécutées systématiquement, même lors de l'inclusion du module. On souhaite généralement importer les fonctions et classes, mais pas lancer les instructions.

C'est ici qu'intervient le test `if __name__ == '__main__':` # (Programme Principal)

- **Python Extension Proposal : PEP-8**

PEP 8 (pour Python Extension Proposal) est un ensemble de règles qui permet d'homogénéiser le code et d'appliquer de 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 PEP8 tranche : ce sont les espaces qui gagnent, au nombre de 4.

Encodage

A préciser en première ligne de code si besoin. Par défaut l'UTF-8 est utilisé.

Import

A mettre en début de programme après l'encodage.

Indentation Lignes

Les lignes ne doivent pas dépasser 79 caractères. Séparer les fonctions et les classes à la racine d'un module par 2 lignes vides. Les méthodes par 1 ligne vide.

Les espaces

Les opérateurs doivent être entourés d'espaces. On ne met pas d'espace à l'intérieur des parenthèses, crochets ou accolades.

Exercice dans la console Python

1. Assigner les valeurs respectives 3, 5, 7 à trois variables a, b, c.
2. Effectuer l'opération `a - b // c`. (division entière)
3. Interpréter le résultat obtenu ligne par ligne.

Importation de module

Importation de modules

Un module est un fichier python que l'on veut importer.

Un package est un dossier dans lequel ont stock des fichiers pythons (modules) et le fichier « `__init__.py` » (même vide).

On peut par exemple créer un fichier « `calculatrice.py` » ou bien un dossier « `calculatrice` » contenant le fichier « `__init__.py` » qui contiendra le code de « `calculatrice.py` ». Dans ce dernier cas le nom du module sera le nom du dossier. Cela est utilisé pour exécuter du code d'initialisation, configurer des variables globales, charger des données ou initialiser des sous-modules. Pas pour un programme entier.

`PYTHONPATH` est une variable d'environnement qui définit les chemins supplémentaires où Python cherche des modules lorsqu'on les importe. Le dossier à partir duquel on a exécuté le programme est toujours inclus dans les chemins. Il est possible de modifier cette variable directement dans le système d'exploitation de façon constante ou de la modifier dans un script Python de façon temporaire.

Ordonner les lignes d'import :

import de module standard

import d'une partie du contenu d'un module standard

import de module tierce

import d'une partie du contenu d'un module tierce

import de module personnel

import d'une partie du contenu d'un module personnel

Importation de modules

```
import os  
import sys
```

```
# import module de la librairie standard  
# on groupe car même type
```

```
from itertools import islice  
from collections import namedtuple
```

```
# import le contenu d'une partie d'un module  
# on groupe car même type
```

```
import requests  
import arrow
```

```
# import librairie tierce partie  
# on groupe car même type
```

```
from django.conf import settings  
from django.shortcuts import redirect
```

```
# contenu d'une partie d'un module tierce  
# on groupe car même type
```

```
# import une fonction du projet  
from myPackage.mySubPackage.myModule import myFunc
```

```
# import une classe du projet  
from myPackage.mySubPackage.myModule import MyClass
```

Les Variables

Règles de nommage

- Python ne possède pas de syntaxe particulière pour créer ou déclarer une variable.
- Il existe quelques règles usuelles pour la dénomination des variables.
- Les variables vont pouvoir stocker différents types de valeurs comme des nombres, des chaînes de caractères, des booléens, et plus encore.
- La casse est significative dans les noms de variables.

Il y a 3 règles à respecter pour déclarer une variable :

- Le nom doit débuter par une lettre ou un underscore « _ ».
- Le nom d'une variable doit contenir que des caractères alphanumériques courant, sans espace ni caractères spéciaux ou accents.
- On ne peut pas utiliser certains mots qui possède déjà une signification en Python. Ce sont les mots réservés.

Les types

- Integer

```
>>> a = 15
>>> type(a)
>>> class <int>
```

- String

```
>>> texte1 = 'Les crêpes.'
>>> texte2 = ""Oui", répondit-il,"
>>> texte3 = "j'aime bien."
>>> print(texte1, texte2, texte3)
>>> 'Les crêpes. Oui, répondit-il, j'aime bien. '
```

- Float

```
>>> b = 16.0
>>> type(b)
>>> class <Float>
```

- Boolean

```
>>> c = True
>>> type(c)
>>> class <bool>
```

Les chaînes de caractères

Une donnée de type string peut se définir en première approximation comme une suite quelconque de caractères. Dans un script python, on peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double quotes).

- **Le caractère spécial « \ » (antislash)**

En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).

À l'intérieur d'une chaîne de caractères, l'antislash permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.).

Exemples :

```
>>> print("ma\tpetite\nchaine")
```

# affiche ma	petite
# chaine	

Notion de formatage

- Format historique (Python 1):

%s	: string	>>> n = "Celine"
%d	: decimal integer	>>> a = 42
%f	: float	>>> print("nom : %s - age : %d" % (n, a))
%g	: generic number	>>> print("nom : %(nom)s - age : (age)s" % {"nom": n, "age": a})

Ce format vient directement de la fonction printf du langage C.

- Depuis Python 2.6 :

.format()	>>> print("nom : {1} - age : {0}".format(a, n))
	>>> print("nom : {age} - age : {nom} ".format(nom=n, age=a))

- Depuis Python 3.6 :

fstring	>>> print(f"nom : {n} - age : {a}")
---------	-------------------------------------

Les 3 formatages existent dans les dernières versions de Python.

Typage dynamique fort

Python est fortement typé dynamiquement.

Un typage fort signifie que le type d'une valeur ne change pas de manière inattendue. Chaque changement de type nécessite une conversion explicite.

En Python, le typage dynamique signifie que le type des variables est déterminé automatiquement au moment de l'exécution (et non lors de l'écriture du code). En d'autres termes, il n'y a pas besoin de déclarer le type d'une variable à l'avance, Python le déduit automatiquement en fonction de la valeur qu'on lui assigne. Les valeurs attribuées n'ont pas de limite contrairement aux autres langages. La seule limite est la mémoire du PC.

```
>>> points = 3.2                                # points est du type float (nombre décimal)
>>> print("Tu as " + points + " points !")        # Génère une erreur de typage
>>> points = int(points)                          # points est maintenant du type int (entier),
                                                    # sa valeur est arrondie à l'unité inférieure (ici 3)
>>> print("Tu as " + points + " points !")        # Génère une erreur de typage
>>> points = str(points)                          # points est maintenant du type str (string)
>>> print("Tu as " + points + " points !")        # Plus d'erreur de typage, affiche "Tu as 3 points !"
```

variables.py

Les collections

Les chaînes de caractères que nous avons abordées constituaient un premier exemple de données composites. On appelle ainsi les structures de données qui sont utilisées pour regrouper de manière structurée des ensembles de valeurs.

Listes

```
#Déclarer une list avec la possibilité de modifier son contenu  
list_data = [1, 2, 3, 4]
```

Tuples

```
#Déclarer un tuple sans la possibilité de modifier son contenu  
tuple_data = (1, 2, 3, 4)
```

Sets

```
# Un set ne contient pas de doublon.  
# ici, le deuxième élément "pierre" sera ignoré.  
mon_set = {"pascal", "pierre", "paul", "pierre"}
```

Dictionnaires

```
#Construire un dictionnaire  
dico = {}  
dico['computer'] = 'ordinateur'  
dico['mouse'] = 'souris'  
dico['keyboard'] = 'clavier'  
print("dico")  
dico = {'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
```

collection.py

Exercice « ex_collection.py »

- Enoncé :
 - Soit une chaîne de caractères comprenant, trois champs séparés par des caractères '--' , comprenant à son tour trois champs séparés par des caractères ';' (un numéro d'étudiant, un nom et un prénom).
 - Faites un algorithme qui retourne un dictionnaire dont les clés sont les numéros d'étudiants et les valeurs sont, pour chaque numéro d'étudiant, une chaîne correspondant à la concaténation des prénom et nom de la personne.
- 213615200;BESNIER;JEAN--213565488;DUPOND;MARC--214665555;DURAND;JULIE

Les blocs d'instructions

Les blocs d'instructions

Avec Python, elles sont définies par la mise en page.

Vous devez utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs.

En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur.

```
def power(num, x=1):  
    result = 1  
    for i in range(x):  
        result = result * num  
    return result
```

Les structures répétitives

Répétitions en boucle

- L'instruction while

```
# boucle while
x = 0
while (x < 5):
    print(x)
    x += 1
```

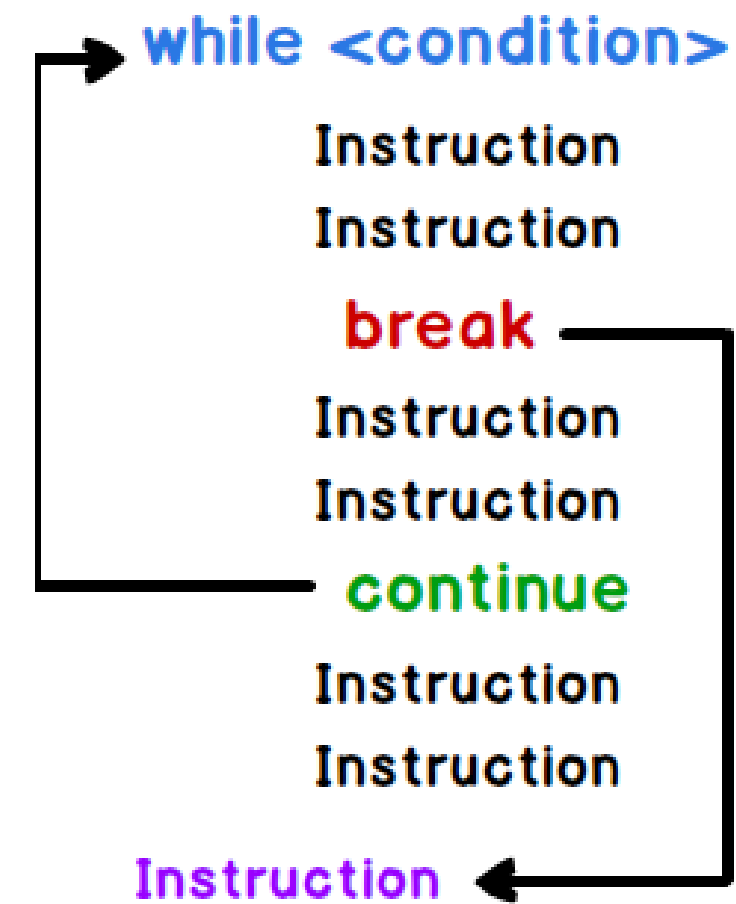
L'instruction For

```
# boucle for
for x in range(5, 10):
    print(x)
```


Instruction continue et break

Break est utilisé pour quitter une boucle for/while en cours d'exécution.

Continue est utilisé pour ignorer la suite du bloc actuel et revenir à l'instruction for/while.



Les structures conditionnelles

Instruction conditionnelle If

```
# if, elif, else
if x < y:
    st = "x est plus petit que y"
elif x == y:
    st = "x est égale à y"
else:
    st = "x est plus grand que y"
print(st)
```

Remarques :

Les boucles et conditions sont imbriquables.

Les parenthèses ne sont pas obligatoires mais donne de la visibilité.

Lorsqu'une seule instruction est exécutée, on peut la mettre sur la même ligne que le if, elif ou else.

L'instruction conditionnelle « if » permet de faire des comparaisons entre deux objets selon l'opérateur utilisé.

Opérateurs de comparaison

X == Y	# égale
X != Y	# différent
X < Y	# inférieur
X > Y	# supérieur
X >= Y	# supérieur ou égale
X <= Y	# inférieur ou égale

Opérateurs logiques

X or Y	# ou
X and Y	# et
not X	# négation de X

Les booléens ont la valeur True ou False avec une majuscule

Instruction conditionnelle If

- **Opérateurs d'identité**

X is Y # Retourne True si les deux variables sont les mêmes objets.

X is not Y # Retourne True si les deux variables ne sont pas les mêmes objets.

- **Opérateurs d'inclusion**

X in Y # Retourne True si X fait partie de Y.

X not in Y # Retourne True si X ne fait pas partie de Y.

Condition ternaire

Permet d'affecter une valeur selon une condition de type if else en une seule ligne de code.

Compréhension de liste

La compréhension de liste permet de créer une liste à partir d'une boucle for combiné ou non avec une instruction de condition if (ou if else) sur une seule ligne.

Nouvelle structure conditionnelle depuis python v3.10

```
# match case
match (x):
    case 1:
        y = 0
    case 2:
        y -= 1
    case _:
        y += x + 3
```

Exercice « ex_liste_1 »

- Enoncé :
 - Ecrivez un programme qui recherche le plus grand élément présent dans une liste donnée.
 - Par exemple, si on l'appliquait à la liste [32, 5, 12, 8, 3, 75, 2, 15], ce programme devrait afficher :
le plus grand élément de cette liste à la valeur 75.

ex_liste_1

Exercice « ex_liste_2 »

- Enoncé :
 - Ecrivez un programme qui donne la somme des tous les nombres supérieurs à 10 se trouvant dans une liste.
 - Si on l'appliquait à la liste [32, 5, 12, 8, 3, 75, 2, 15], ce programme devrait afficher :
la somme demandée est 134

ex_liste_2

Exercice « ex_listes.py »

- Enoncé :
 - Ecrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : ['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']) pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.

Les fonctions

Les fonctions prédéfinies

• Quelques fonctions prédéfinies / natives (builtin)

<code>abs(x)</code>	# retourne la valeur absolue de x.
<code>all(iterable)</code>	# retourne True si toutes les valeurs de l'itérable sont True.
<code>any(iterable)</code>	# retourne True si au moins une valeur de l'itérable est True.
<code>bin(int)</code>	# convertit nombre entier en chaîne de caractère binaire.
<code>hex(int)</code>	# convertit nombre en valeur hexadécimale.
<code>len(obj)</code>	# retourne la longueur de l'objet.
<code>list(obj)</code>	# cast l'objet au format liste.
<code>map(fct, obj)</code>	# applique une transformation à tous les éléments d'un itérable.
<code>filter(fct, list)</code>	# filtre avec un prédicat les éléments d'un itérable et retourne un booléen.

• Fonctions natives de l'objet str

<code>str.capitalize()</code>	# retourne la string avec une majuscule en début de phrase
<code>str.title()</code>	# retourne la string avec une majuscule en début de chaque mot
<code>str.upper()</code>	# retourne la string en majuscule
<code>str.lower()</code>	# retourne la string en minuscule
<code>str.strip()</code>	# retourne la string str en supprimant les espaces avant et après la phrase
<code>str.count(x)</code>	# retourne le nombre d'occurrence de x dans str
<code>str.endswith("x")</code>	# retourne True si la str fini par 'x'
<code>str.startswith("x")</code>	# retourne True si la str commence par 'x'
<code>str.find(x)</code>	# retourne l'index de la première occurrence de x (-1 si n'existe pas)

ex_chaines

Print et Input

- **Fonction print()**

Elle permet d'afficher sur la sortie standard (paramètre file="sys.stdout"), n'importe quel nombre de valeurs fournies en arguments (c'est-à-dire entre les parenthèses). Par défaut, ces valeurs seront séparées les unes des autres par un espace (paramètre sep=" "), et le tout se terminera par un saut à la ligne (paramètre end="\n"). Le paramètre flush=False permet de ne pas afficher immédiatement la sortie, cela passe par un tampon géré par Python.

```
>>> print("Bonjour", "à", "tous", sep="*")    # Bonjour*à*tous sur la sortie standard
>>> print("Bonjour", "à", "tous", sep="" , file=sys.stderr)    # Bonjouràtous sur la sortie d'erreur
```

- **Fonction input() : Interaction avec l'utilisateur**

Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <Enter>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a saisi.

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
OU
print("Veuillez entrer un nombre positif quelconque : ")
ch = input()
nn = int(ch) # conversion de la chaîne en un nombre entier
print("Le carré de", nn, "vaut", nn ** 2)
```

Exercice « ex_fonctions_natives.py »

- Enoncé :
 - Ecrire une boucle de programme qui demande à l'utilisateur d'entrer des notes d'élèves. La boucle se terminera seulement si l'utilisateur entre une valeur négative.
 - Avec les notes ainsi entrées, construire progressivement une liste.
 - Après chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), afficher le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes (informations stockées dans un dictionnaire).

Exercice « ex_input.py »

- Enoncé :
 - Ecrivez un programme qui convertisse en mètres par seconde une vitesse fournie par l'utilisateur en Km/h.
 - Demander à l'utilisateur de rejouer. « Voulez-vous rejouer ? » Réponse possible oui ou non.

ex_input

Exercice « ex_boucles_imbriquees.py »

- Enoncé :
 - Ecrivez un programme qui affiche les 20 premiers résultats de la table de multiplication par un nombre entré par l'utilisateur.
 - Demander à l'utilisateur de rejouer. « Voulez-vous rejouer ? » Réponse possible oui ou non.

ex_boucles_imbriquees

Les fonctions

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples. D'autre part, il arrivera souvent qu'une même séquence d'instructions doit être utilisée à plusieurs reprises dans un programme. La déclaration d'une fonction commence par le mot clé « def » suivi de son nom, puis, des parenthèses pouvant contenir des arguments, et finie par « : ». Cela constitue sa signature. Son nom est aussi appelé identificateur. Son contenu c'est son implémentation. Si on ne veut pas l'implémenter tout de suite on doit écrire le mot clé « pass ».

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

• Les arguments des fonctions

Une fonction peut avoir un nombre fixe ou variable d'arguments. Ces arguments peuvent être des variables (int, string, list...) mais aussi d'autres fonctions. Les arguments peuvent être positionnels ou nommés. Dans certains cas, vous pouvez obliger les appelants de votre fonction particulière à spécifier les arguments en utilisant uniquement leurs noms afin d'améliorer la lisibilité du code ou d'attirer l'attention sur l'importance du paramètre.

Pour obtenir un nombre variable de paramètre à transmettre à une fonction, on préfixe le ou les arguments de la fonction par un ou deux astérisques.

Un * correspond aux arguments positionnels (sous forme de tuple).

Deux ** correspond aux arguments nommés (sous forme de dictionnaire).

Ils sont souvent nommés *args et **kwargs.

Les fonctions

- **Variable locale vs variables globales**

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

Les fonctions

- **Fonction récursive**

Une fonction récursive est une fonction qui s'appelle elle-même. Attention à bien mettre une condition pour sortir de l'appel récursif, sinon on provoque une boucle infinie. Dans certains cas cela peut être intéressant, mais la plupart du temps ce n'est pas une bonne idée car elles sont peu performantes et il y a une limite de nombre d'appels (999 empilables en comptant la fonction main() et le premier appel de la fonction récursive).

Exemple la fonction factorielle(x)

```
def factorielle(n):  
    if n < 2:  
        return 1  
    else:  
        return n * factorielle(n-1)
```

```
def factorielle(n):  
    return 1 if n < 2 else n * factorielle(n-1)
```

Exercice « ex_fonction.py »

- Enoncé :
 - Définissez une fonction `maximum(n1, n2, n3)` qui renvoie le plus grand de 3 nombres `n1`, `n2`, `n3` fournis en arguments. Par exemple, l'exécution de l'instruction :

`print(maximum(2, 5, 4))` doit donner le résultat : 5.

Exercice « ex_fonctions.py »

- Enoncé :
 - Ecrire une fonction cube qui retourne le cube de son argument.
 - Ecrire une fonction volumeSphere qui calcule le volume d'une sphère de rayon r fourni en argument et qui utilise la fonction cube.
 - Tester la fonction volumeSphere par un appel dans le programme principal.

ex_fonctions

Les fonctions

- **Fonction Lambda**

Pour Python, c'est la seule façon d'écrire une fonction anonyme. Ceci est particulièrement utile pour la programmation fonctionnelle. En effet, une fonction peut être directement écrite dans un appel de fonction sans avoir à la définir au préalable.

```
>>> list(map(lambda x: x ** 2, range(10)))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Bien que les fonctions lambda soient utilisées dans le but de créer des fonctions anonymes, on peut décider tout de même de leur donner un nom :

```
>>> f = lambda x: x ** 2  
>>> f(5)  
25  
>>> list(map(f, range(10)))  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Syntaxe pour passer une variable à l'exécution : `lambda x: monCalculAvecX`

Syntaxe pour passer une variable à la création : `lambda x = x: monCalculAvecX`

TP validation des acquis 1

TP validation des acquis 1

- Enoncé :
- Créer un petit programme interactif de gestion de notes pour un élève. Le programme devra permettre de :
 - Demander à l'utilisateur combien de notes il veut entrer.
 - Saisir ces notes une par une (entre 0 et 20) et les ajouter dans une liste.
 - Calculer la moyenne de ces notes via une fonction nommée "calcul_moyenne".
 - Afficher un message différent selon la moyenne :
 - Moins de 10 : "En difficulté"
 - Entre 10 et 15 : "Peut mieux faire"
 - 15 ou plus : "Très bien"

- Résultat attendu :

Combien de notes voulez-vous entrer ? 3

Entrez la note 1 : 12

Entrez la note 2 : 8

Entrez la note 3 : 15

Moyenne : 11.67

Appréciation : Peut mieux faire

Les docstrings

Docstring

- L'utilisation du docstring

C'est toujours une bonne idée d'écrire de la documentation pour vos fonctions, classes et modules. Et l'une des grandes caractéristiques de Python est qu'il vous permet de le faire dans du code. La documentation du code de votre programme, ainsi que du code de Python, est directement disponible lorsque votre programme est exécuté ou utilisé par un autre développeur.

```
>>> print(any.__doc__)  
>>> help(objet) # objet peut être un module, une classe une méthode, une fonction native, etc ...
```

Cela fonctionne pour les fonctions mais aussi pour les modules et les classes.

```
>>> import collections  
>>> print(collections.__doc__)
```

```
def maFonction(arg1, arg2=None):  
    """maFonction(arg1, arg2=None) --> Ne fait rien de spéciale.  
  
    Parameters:  
    arg1: premier argument standard.  
    arg2: deuxieme argument avec valeur par défaut à None.  
    """  
  
    print(arg1, arg2)  
  
def main():  
    print(maFonction.__doc__)
```

Docstring

- Les chaînes de documentation doivent toujours être placées entre des triples guillemets ou triples apostrophes, même si elles ne font qu'une ligne.
- La première ligne de votre docstring doit résumer la fonction, la classe ou le module et son objectif principal.
- Pour les paquets et les modules, listez les classes et les sous-modules importants qu'ils contiennent, ainsi que les exceptions personnalisées que le développeur doit connaître.
- Pour les classes, listez les méthodes et les informations importantes.
- Pour les fonctions, il y a une variété d'éléments importants à lister :
 - Assurez-vous que votre docstring liste et explique chacun des arguments (un par ligne), y compris les facultatifs.
 - Si la fonction renvoie une valeur, expliquez-la dans la docstring. Sinon, la convention habituelle consiste à économiser de l'espace en ne l'indiquant pas. Si la fonction soulève des exceptions, veillez à les mentionner également.

Docstring

- Il existe plusieurs standards de docstrings :

- **PEP257 (Style officiel de Python)**

Première phrase : résumé en une ligne (phrase complète).

Une ligne vide.

Description détaillée des paramètres et du retour.

Pas de sections particulières.

- **Style Google**

Première phrase : résumer en une ligne.

Utilise des sections nommées Args, Attributs, Returns et Raises suivi de « : ».

Des lignes vides entre chaque section.

Les variables sont décrites avec leurs types entre parenthèses sur une ligne. La description est séparée du reste par un « : ».

- **Style Numpy**

Première phrase : résumé en une ligne.

Utilise les sections Parameters et Returns.

Une ligne vide entre chaque section.

Chaque section est soulignée par des tirets.

Les variables sont indiquées avec leur type sur une ligne (séparé par un « : »), et une ligne décalée d'une indentation en dessous pour leur description.

docstrings.py

Les exceptions

Les exceptions et la gestion des erreurs

Toutes les erreurs qui se produisent lors de l'exécution d'un programme Python sont représentées par une exception. Une exception est un objet qui contient des informations sur le contexte de l'erreur. Lorsqu'une exception survient et qu'elle n'est pas traitée alors elle produit une interruption du programme et elle affiche sur la sortie standard un message ainsi que la pile des appels (stacktrace). La pile des appels, présente dans l'ordre, la liste des fonctions et des méthodes qui étaient en cours d'appel au moment où l'exception est survenue.

Parfois un programme est capable de traiter le problème à l'origine de l'exception. Par exemple si le programme demande à l'utilisateur de saisir un nombre et que celui-ci saisit une valeur erronée, le programme peut simplement demander à l'utilisateur de saisir une autre valeur plutôt que de faire échouer le programme.

```
try:  
    pass  
except:  
    pass  
else:  
    pass  
finally:  
    pass
```


Exercice « ex_exception.py »

- Enoncé :
 - Ecrire un programme qui demande à l'utilisateur de saisir des entiers un par un (on saisira le mot fin pour finir la saisie des nombres) puis à l'aide de parcours successifs de la liste effectuer les actions suivantes :
 - 1) Afficher la liste.
 - 2) Afficher la liste en colonne de manière à afficher l'index et le contenu.
 - 3) Créer une nouvelle liste qui sera chaque élément de la liste multiplié par 3 en utilisant une fonction lambda.
 - 4) Obtenir le plus grand nombre de la liste.
 - 5) Obtenir le plus petit nombre de la liste.
 - 6) Obtenir la quantité de nombre pair présents dans la liste.
 - 7) Calculer la somme de tous les nombres impairs de la liste.
 - Le programme doit gérer les exceptions au niveau de la saisie des données de l'utilisateur.

Les fichiers textes

Les fichiers textes

Un fichier texte est un fichier contenant des caractères lisibles (comme .txt, .csv, .html, etc.). Python permet de lire, écrire, ajouter ou modifier ces fichiers facilement.

Ecriture

```
>>> f = open('Monfichier', 'w')
>>> f.write('Bonjour, fichier !')
>>> f.write("Quel beau temps!")
>>> f.close()
```

Lecture

```
>>> f = open('Monfichier', 'r')
>>> t = f.read()
>>> print(t)
Bonjour, fichier !Quel beau temps !
>>> f.close()
```

- Liste des modes d'ouverture des fichiers textes.

'r'	Read (lecture)	Ouvre un fichier en lecture seule (le fichier doit exister).
'w'	Write (écriture)	Ecriture seule ; crée ou écrase le fichier s'il existe.
'x'	Exclusive	Créer le fichier. provoque une erreur s'il existe déjà.
'a'	Append (ajout)	Ajoute à la fin ; crée le fichier s'il n'existe.
'r+'	Read / Write	Lecture / écriture ; ne crée pas ; n'efface rien ; curseur au début.
'w+'	Read / Write (écrase)	Lecture / écriture ; crée ou écrase le fichier.
'a+'	Read / Append	Lecture / ajout à la fin ; curseur au début pour lecture, fin pour écriture.
't'	Text	Mode texte. Valeur par défaut de open().
'b'	Binary	Mode binaire. Pour lire/écrire d'autres choses que du texte.

Les modes «t » et « b » se combine aux autres modes. Exemple : « rt », « rt+, etc...

Les fichiers textes

- Les fonctions pour manipuler les fichiers

<code>open()</code>	: Ouvre le fichier. Attend deux arguments minimums, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier, et le second est le mode d'ouverture.
<code>write()</code>	: Réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Chaque nouvel appel de <code>write()</code> (en mode <code>a</code>) continue l'écriture à la suite de ce qui est déjà enregistré.
<code>writelines()</code>	: Accepte un itérable (liste, tuple, etc...) contenant des strings. La fonction écrira tous les éléments de l'itérable les uns à la suite de autres. Il ne faut pas oublier d'insérer les « <code>\n</code> » à la fin des strings si on veut des sauts de lignes à ce moment là.
<code>read()</code>	: Permet de lire le contenu d'un fichier dans son ensemble.
<code>readline()</code>	: Lit qu'une ligne par appel de cette instruction.
<code>readlines()</code>	: Lit les lignes individuellement dans une liste.
<code>seek(offset, whence)</code>	: Déplace le curseur dans le fichier (<code>whence=0</code> : début, <code>1</code> : position actuelle, <code>2</code> : fin). Si <code>whence</code> n'est pas spécifié le curseur est déplacé de la valeur de l'offset à partir de sa position actuelle.
<code>close()</code>	: Referme le fichier dans n'importe qu'elle mode.

fichier_texte.py

Exercice « ex_fichier.py »

- Enoncé :
- Ecrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 30 (chacune d'entre elles incluant les termes de 1 à 20 seulement).

ex_fichier

Exercice « ex_multi_fichiers.py »

- Enoncé :
- A partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A... et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux.
- Complétez ensuite C avec les éléments restant sur l'autre.

Exercice « ex_fichier_mini_bdd.py »

- Enoncé :
 - Mini Système BDD à partir d'un fichier qu'on nommera utilisateurs.txt et qui se situera dans le même dossier que ce script.
 - Dans un premier temps :
 - Créer un dictionnaire qui permettra d'enregistrer en clé le nom et en valeur l'age et la taille de l'utilisateur.
 - Créer une fonction inscription pour saisir les données utilisateurs, les inscrire dans le dictionnaire et poser la question si on veut continuer à saisir un utilisateur.
 - Créer une fonction consultationTotale qui permet de voir les données des utilisateurs enregistrés dans le dictionnaire.
 - Créer une fonction consultation qui permettra de consulter les données d'un utilisateur.
 - Créer un menu pour choisir entre quitter, inscription ou consultation.
- Dans un deuxième temps :
 - Créer une fonction enregistrer qui enregistrera les infos utilisateurs dans le fichier nommé utilisateurs.txt
 - On utilisera le caractère séparateur @ pour séparer la clé des valeurs du dictionnaire, et le caractère # pour séparer les données constituant ces valeurs. Exemple Juliette@18#1.68
 - Créer une fonction lecture qui permettra de lire le fichier utilisateurs.txt et d'inscrire les données lues dans le dictionnaire.
 - Modifier le menu pour ajouter les fonctions enregistrement et lecture.

ex_fichier_mini_bdd

Exercice « ex_fichier_mini_bdd.py »

- Enoncé :
- Exemple de menu :
 - (L) Lecture
 - (I) Inscription
 - (C) Consultation par nom
 - (T) Consultation totale
 - (E) Enregistrement
 - (Q) Quitter

ex_fichier_mini_bdd

Les expressions régulières

Expressions régulières

Pour faire des tests, utilisez le site <https://regex101.com/>

Les expressions régulières représentent les motifs qu'on recherche dans une chaîne de caractères. Par exemple, on peut chercher dans un fichier :

- Les lignes qui contiennent un numéro de téléphone
- Les lignes qui débutent par une adresse IP

On peut chercher dans une chaîne de caractère :

- Une séquence qui commence par « ex » et se termine par « le »
- Les mots de quatre caractères de long
- Les mots sans majuscules

On utilise les symboles suivants qui ont une signification spécifique chacune !

. ^\$*+?{}[]\()

Les quantificateurs

- +** : Le résultat peut contenir une ou plusieurs occurrences du caractère qui le précède.
- *** : Le résultat peut contenir zéro ou plusieurs occurrences du caractère qui le précède.
- ?** : Le résultat peut inclure zéro ou une occurrence du caractère qui le précède.
- {x}** : Précise le nombre x de répétitions des chaînes de caractères qui le précède.
- []** : Obtient le résultat sur plusieurs occurrences.
- ()** : Créer des groupes.

- |** : Correspond à l'expression OU. On peut s'en servir pour trouver une correspondance entre plusieurs expressions.
- : Sert à regrouper des caractères dans un intervalle donné.

Les métacaractères et tokens

- **Les métacaractères**

- `^` : Précise que le résultat doit commencer par la chaîne de caractères qui le suit.
- `$` : Précise que le résultat doit se terminer par la chaîne de caractère qui le précède.
- `.` : Correspond à n'importe quel caractère sauf le saut de ligne.

- **Les tokens**

- `\d` : [0-9]
- `\w` : [a-zA-Z0-9_]
- `\s` : correspond à tous les espaces dans la chaîne de caractères
- `\.` : juste les points
- `\b` : frontière des mots

Fonctions usuelles

- `re.compile(motif)` : Compile un motif (expression régulière) utilisable par les autres fonctions telle que `search`, `findall`, etc ...
- `re.search(motif, string)` : Renvoie la première correspondance du motif trouvée dans la string.
- `re.findall(motif, string)` : Renvoie une liste de toutes les correspondances du motif trouvées dans la string.
- `re.match(motif, string)` : Renvoie la correspondance trouvée si zéro ou plus caractères en début de string correspondent au motif.
- `re.sub(motif, chaîne, string)` : Remplace par chaîne les occurrences trouvées dans la string (sans chevauchement) et renvoie le résultat. Si le motif n'est pas trouvé, string est renvoyée inchangée.

Exercice « ex_regex.py »

- Dans une chaîne de caractères donnée :
 - 1 - Retrouvez tous les mots dans une chaîne de caractères.
 - 2 - Retrouver tous les mots qui se terminent par un caractère donné.
 - 3 - Retrouver tous les mots qui commencent par un caractère donné.
 - 4 - Retrouver tous les mots qui contiennent au moins trois caractères.
 - 5 - Retrouver tous les mots qui possèdent exactement n caractères.

Les classes

Les classes

- **Orienté objet : ça veut dire quoi ?**

Globalement, les langages de programmation objet implémentent le paradigme de programmation orientée objet (POO). Ce paradigme consiste en la réunion des données et des traitements associées à ces données au sein d'entités cohérentes appelées objets. Python est un langage objet composé de classes. Une classe représente un « moule » permettant de créer des objets (instances), et regroupe les attributs et méthodes communes à ces objets.

- **Les classes**

L'orienté objet facilite beaucoup dans la conception, la maintenance, la réutilisabilité des éléments (objets). Le paradigme de POO permet de tirer profit de classes parents et de classes enfants (phénomène d'héritage), etc.

Tout objet donné possède deux caractéristiques :

Son état courant (attributs)

Son comportement (méthodes)

En approche orienté objet on utilise le concept de classe, celle-ci permet de regrouper des objets de même nature.

Une classe est un moule (prototype) qui permet de définir les attributs (variables) et les méthodes (fonctions) de tous les objets de cette classe.

Les classes sont les principaux outils de la POO. Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent entre eux et avec le monde extérieur.

Les classes

- **Attributs de classe**

Une classe peut également avoir des attributs. Pour cela, il suffit de les déclarer dans le corps de la classe. Les attributs de classe sont accessibles depuis la classe elle-même et sont partagés par tous les objets. Si un objet modifie un attribut de classe, cette modification est visible de tous les autres objets. Les attributs de classe sont le plus souvent utilisés pour représenter des constantes.

- **Méthode de classe**

Tout comme il est possible de déclarer des attributs de classe, il est également possible de déclarer des méthodes de classe. Pour cela, on utilise le décorateur `@classmethod`. Comme une méthode de classe appartient à une classe, le premier paramètre correspond à la classe. Par convention, on appelle ce paramètre `cls` pour préciser qu'il s'agit de la classe et pour le distinguer de `self`.

- **Méthode statique**

Une méthode statique est une méthode qui appartient à la classe mais qui n'a pas besoin de s'exécuter dans le contexte d'une classe. Autrement dit, c'est une méthode qui ne doit pas prendre le paramètre `cls` comme premier paramètre. Pour déclarer une méthode statique, on utilise le décorateur `@staticmethod`. Les méthodes statiques sont des méthodes utilitaires très proches des fonctions mais que l'on souhaite déclarer dans le corps d'une classe.

Les classes

- Exemple de classe

```
#Class avec COnstructor
class Velo:
    roues = 2

    def __init__(self, marque, prix, poids):
        self.marque = marque
        self.prix = prix
        self.poids = poids

    def rouler(self):
        print("Wouh, ça roule mieux avec un vélo {} !".format(self.marque))
```

Quelques remarques importantes

- Tous les attributs et méthodes des classes Python sont « publics » au sens de C++, parce que « nous sommes tous des adultes ! » (citation de Guido von Rossum, créateur de Python).
- Le constructeur d'une classe est une méthode spéciale qui s'appelle `__init__()`.
- En Python, on n'est pas tenu de déclarer tous les attributs de la classe comme d'autres langages : on peut se contenter de les initialiser dans le constructeur !
- Toutes les méthodes prennent une variable `self` comme premier argument. Cette variable est une référence à l'objet manipulé.
- Python supporte l'héritage simple et l'héritage multiple. La création d'une classe fille est relativement simple, il suffit de préciser entre parenthèses le nom de la classe mère lors de la déclaration de la classe fille.

Exercice « ex_cercle_cylindre.py »

- Enoncé :
 - Définissez une classe Cercle(). Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre rayon), vous définirez une méthode surface(), qui devra renvoyer la surface du cercle.
 - Définissez ensuite une classe Cylindre() dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres rayon et hauteur.
 - Vous y ajouterez une méthode volume() qui devra renvoyer le volume du cylindre (rappel : volume d'un cylindre = surface de section × hauteur).

Exercice « ex_compte_bancaire.py »

- Enoncé :
 - Définissez une classe `CompteBancaire()`, qui permette d'instancier des objets tels que `compte1`, `compte2`, etc.
 - Le constructeur de cette classe initialisera deux attributs d'instance `nom` et `solde`, avec les valeurs par défaut 'Dupont' et 1000.
 - Trois autres méthodes seront définies :
 - `depot(somme)` permettra d'ajouter une certaine somme au solde.
 - `retrait(somme)` permettra de retirer une certaine somme du solde.
 - `affiche()` permettra d'afficher le nom du titulaire et le solde de son compte.

Exercice « `ex_compte_epargne.py` »

- Enoncé :
 - Ecrivez un nouveau script qui récupère le code de (compte bancaire) en l'important comme un module.
 - Définissez-y une nouvelle classe `CompteEpargne()`, dérivant de la classe `CompteBancaire()` importée, qui permette de créer des comptes d'épargne rapportant un certain intérêt au cours du temps.
 - Pour simplifier, nous admettrons que ces intérêts sont calculés tous les mois.
 - Le constructeur de votre nouvelle classe devra initialiser un taux d'intérêt mensuel par défaut égal à 0,3 %. Une méthode `changeTaux(valeur)` devra permettre de modifier ce taux à volonté.
 - Une méthode `capitalisation(nombreMois)` devra :
 - Afficher le nombre de mois et le taux d'intérêt pris en compte.
 - Calculer le solde atteint en capitalisant les intérêts composés, pour le taux et le nombre de mois qui auront été choisis.
 - Redéfinir la fonction d'affichage héritée pour ajouter le taux mensuel du compte épargne.

`ex_compte_epargne`

Exercice « ex_jeu_de_cartes.py »

- Enoncé :
 - Définissez une classe `JeuDeCartes()` permettant d'instancier des objets dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les quatre méthodes suivantes :
 - Méthode constructeur : Création et remplissage d'une liste de 52 éléments. Ces éléments sont des tuples contenant la couleur (Coeur, Trèfle, Pique, Carreau) et la valeur (2, 3, 4, 5, 6, 7, 8, 9, 10, Valet, Dame, Roi, As) de chacune des cartes. Dans une telle liste, l'élément (Valet , Pique) désigne donc le Valet de Pique, et la liste terminée doit être sous la forme : [(2, Coeur), (3, Coeur),, (As, Carreau)].
 - Méthode `nom_carte()` : cette méthode doit renvoyer, sous la forme d'une chaîne, l'identité d'une carte quelconque dont on lui a fourni le tuple descripteur en argument. Par exemple, l'instruction : `print(jeu.nom_carte((valeur, couleur)))` doit provoquer l'affichage de : 2 de Carreau
 - Méthode `melanger()` : Cette méthode sert à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre.
 - Méthode `tirer()` : lorsque cette méthode est invoquée, la première carte de la liste est retirée du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer `None` au programme appelant.

ex_jeu_de_cartes

Exercice « ex_jeu_a_et_b.py »

- Enoncé :
 - Complément de l'exercice précédent : définir deux joueurs A et B.
 - Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger.
 - Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des deux qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant.
 - Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.

ex_jeu_a_et_b

Les classes avancées

Les classes avancées

- **Polymorphisme**

Le polymorphisme est un mécanisme important dans la programmation objet. Il permet de modifier le comportement d'une classe fille par rapport à sa classe mère. Le polymorphisme permet d'utiliser l'héritage comme un mécanisme d'extension en adaptant le comportement des objets.

- **Propriétés**

Les propriétés permettent de définir des comportements de 'getter' et 'setter' sur les méthodes d'une classe. Cela nous permet également d'appeler une méthode sans avoir besoin d'utiliser les parenthèses.

- Pour créer une propriété de type getter, on utilise le décorateur `@property` sur une méthode.
- Pour ajouter un 'setter', il faut décorer la méthode du même nom avec un décorateur ayant la syntaxe `@nom_method.setter`. Si aucun setter n'est défini alors elle n'est pas modifiable, par conséquent c'est une constante.
- Il existe aussi un 'deleter' qui permet de supprimer la propriété. La syntaxe est la même que le setter en remplaçant « setter » par « deleter ».

polymorphisme.py

Les classes avancées

- **Encapsulation**

En Python, il n'existe pas de mécanisme dans le langage qui nous permettrait de gérer la visibilité. Par contre, il existe une convention dans le nommage. Une méthode ou un attribut dont le nom commence par « _ » (underscore) est considéré comme privé. Il est donc déconseillé d'accéder à un tel attribut ou d'appeler une telle méthode depuis l'extérieur de l'objet.

- **Masquer des attributs / méthodes à une classe fille**

Parfois, on ne souhaite pas qu'une méthode puisse être redéfinie ou qu'un attribut puisse être modifié dans une classe fille. Pour cela, il suffit que le nom de la méthode ou de l'attribut commence par deux « _ ». Nous avons vu précédemment que Python n'a pas de mécanisme pour contrôler la visibilité des éléments d'une classe. Par convention, les développeurs signalent par un « _ » le statut privé d'un attribut ou d'une méthode. Par contre le recours à deux underscores a un impact sur l'interpréteur qui renomme la méthode ou l'attribut sous la forme :

- `__<nom de la classe>__<nom>`

propriete_encapsulation.py

Les classes avancées

- **Destructeur**

Les destructeurs sont appelés lorsqu'un objet est détruit. En Python, les destructeurs ne sont pas aussi nécessaires que en C++, car Python dispose d'un ramasse-miettes qui gère automatiquement la gestion de la mémoire.

La méthode `__del__()` est une méthode appelée destructeur en Python. Elle est appelée lorsque toutes les références à l'objet ont été supprimées, c'est-à-dire lorsqu'un objet est nettoyé via le mot clé « `del` » ou à la fin du programme par exemple.

- **Fermer la liste des attributs**

Il est possible déclarer la liste finie des attributs. Pour cela, on déclare un attribut de classe appelé `__slots__`.

Lorsque `__slots__` n'est pas indiqué, on peut ajouter des attributs inexistants à la création d'une classe. Si on l'indique, cela n'est plus possible et génère une erreur de type `AttributeError`.

L'attribut `__slots__` a également une utilité pour l'optimisation du code. Comme on indique à l'interpréteur le nombre exact d'attributs, il peut optimiser l'allocation mémoire pour un objet de ce type.

destructeur.py

Exercice « ex_propriete_encapsulation.py »

- Enoncé :
 - Créez une classe Animal :
 - Attributs privés : `_nom`, `_age`
 - Propriétés pour accéder et modifier nom et age avec vérification (âge ≥ 0).
 - Une méthode `parler()` qui affiche "L'animal fait un bruit."
 - Un destructeur qui affiche : "{nom} a été retiré du zoo.«
 - Créez deux classes filles :
 - Chien, redéfinit `parler()` → affiche "{nom} aboie : Woof!"
 - Chat, redéfinit `parler()` → affiche "{nom} miaule : Miaou!«
 - Créez une fonction `faire_parler(animal)` qui prend un objet Animal et appelle sa méthode `parler()`.

Surcharges

- **Définition de la surcharge**

La surcharge (overloading) en programmation désigne la capacité d'une fonction ou d'un opérateur à fonctionner différemment selon le type ou le nombre d'arguments.

En Python, contrairement à d'autres langages comme Java ou C++, la surcharge des fonctions n'est pas directement prise en charge, car Python ne permet pas d'avoir plusieurs fonctions portant le même nom avec des signatures différentes. Cependant, il existe des moyens d'implémenter un comportement similaire.

- **La surcharge des fonctions**

Python ne permet pas plusieurs définitions d'une fonction avec le même nom. Si on définit plusieurs fois une fonction avec le même nom, seule la dernière définition est conservée. Pour contourner cette limitation, on peut utiliser des valeurs par défaut ou `*args` pour gérer un nombre variable d'arguments.

- **La surcharge d'opérateur**

Python permet la surcharge des opérateurs en redéfinissant des méthodes spéciales (dunder methods ou méthodes magiques).

Surcharges

• Méthodes mathématiques :

<code>__add__(self, other)</code>	<code>:</code>	<code>+</code>	(addition)
<code>__sub__(self, other)</code>	<code>:</code>	<code>-</code>	(soustraction)
<code>__mul__(self, other)</code>	<code>:</code>	<code>*</code>	(multiplication)
<code>__truediv__(self, other)</code>	<code>:</code>	<code>/</code>	(division)
<code>__floordiv__(self, other)</code>	<code>:</code>	<code>//</code>	(division entière)
<code>__mod__(self, other)</code>	<code>:</code>	<code>%</code>	(modulo)
<code>__pow__(self, other)</code>	<code>:</code>	<code>**</code>	(puissance)
<code>__matmul__(self, other)</code>	<code>:</code>	<code>@</code>	(multiplication matricielle)
<code>__and__(self, other)</code>	<code>:</code>	<code>&</code>	(Et logique)
<code>__or__(self, other)</code>	<code>:</code>	<code> </code>	(Ou logique)

• Méthodes de comparaison :

<code>__eq__(self, other)</code>	<code>:</code>	<code>==</code>	(égalité)
<code>__ne__(self, other)</code>	<code>:</code>	<code>!=</code>	(différent)
<code>__lt__(self, other)</code>	<code>:</code>	<code><</code>	(inférieur)
<code>__le__(self, other)</code>	<code>:</code>	<code><=</code>	(inférieur ou égal)
<code>__gt__(self, other)</code>	<code>:</code>	<code>></code>	(supérieur)
<code>__ge__(self, other)</code>	<code>:</code>	<code>>=</code>	(supérieur ou égal)

Surcharges

• Méthodes in-place :

<code>__iadd__(self, other)</code>	<code>: +=</code>	(addition)
<code>__isub__(self, other)</code>	<code>: -=</code>	(soustraction)
<code>__imul__(self, other)</code>	<code>: *=</code>	(multiplication)
<code>__itruediv__(self, other)</code>	<code>: /=</code>	(division)
<code>__ifloordiv__(self, other)</code>	<code>: //=</code>	(division entière)
<code>__imod__(self, other)</code>	<code>: %=</code>	(modulo)
<code>__ipow__(self, other)</code>	<code>: **=</code>	(puissance)
<code>__imatmul__(self, other)</code>	<code>: @=</code>	(multiplication matricielle)
<code>__iand__(self, other)</code>	<code>: &=</code>	(Et logique)
<code>__ior__(self, other)</code>	<code>: =</code>	(Ou logique)

• Méthodes d'affichage :

<code>__str__(self)</code>	<code>: str(objet), print(objet)</code>	(affichage)
<code>__repr__(self)</code>	<code>: repr(objet)</code>	(autre affichage)
<code>__format__(self, format_spec)</code>	<code>: format(objet, format_spec)</code>	(format)
<code>__bytes__(self)</code>	<code>: bytes(objet)</code>	(bytes code)

Surcharges

- Fonctions sur les attributs d'une classe

<code>__getattr__(self, attr)</code>	: <code>objet.attr</code>
<code>__getattribute__(self, attr)</code>	: <code>objet.attr</code>
<code>__setattr__(self, attr, val)</code>	: <code>objet.attr = val</code>
<code>__delattr__(self, attr)</code>	: <code>del objet.attr</code>
<code>__dir__(self)</code>	: <code>dir(objet)</code>

`surcharge_op.py`

Exercice « ex_surcharge.py »

- Enoncé :
 - On souhaite modéliser une classe Rectangle représentant un rectangle dans un plan cartésien. Un rectangle est défini par sa largeur (largeur) et sa hauteur (hauteur).
 - Vous devez :
 - Implémenter la classe Rectangle avec un constructeur prenant en paramètre largeur et hauteur.
 - Surcharger les operateurs suivants :
 - + : pour additionner deux rectangles (somme des largeurs et hauteurs respectives).
 - * : pour multiplier un rectangle par un entier (multiplier la largeur et la hauteur).
 - == : pour comparer si deux rectangles ont la même surface.
 - str : pour afficher un rectangle sous la forme "Rectangle(largeur, hauteur)".
 - Tester la classe avec différents cas.

Les patrons de conception

Les Patrons de conception

- **Qu'est-ce qu'un patron de conception**

En Python, comme dans d'autres langages orientés objet, les design patterns (ou patrons de conception) sont des solutions réutilisables à des problèmes courants de conception logicielle. Python est très flexible et certains patrons sont déjà implémentés naturellement grâce à ses fonctionnalités (décorateurs, introspection, etc.). On peut souvent remplacer des patterns classiques Java/C++ par des structures Python plus élégantes.

- **Singleton**

Assure qu'une classe n'ait qu'une seule instance dans tout le programme.

Exemple d'intérêt :

Gérer une seule connexion à une base de données.

Les Patrons de conception

- **Factory**

Délègue la création d'objet à une fonction ou une classe.

Exemple d'intérêt :

Simplifie la création d'objets complexes.

Permet d'instancier dynamiquement des classes selon les besoins.

Favorise l'extensibilité (on peut ajouter de nouveaux types sans changer le code client).

- **Observer**

Permet à des objets d'être notifiés automatiquement quand un autre change d'état.

Exemple d'intérêt :

Découpler l'émetteur (sujet) des récepteurs (observateurs) dans un système de messagerie où chaque utilisateur est notifié quand il reçoit un message.

Les Patrons de conception

- **Strategy**

Permet à un objet de changer dynamiquement son comportement sans changer son code. Cela fonctionne en définissant une interface de stratégie que des classes concrètes peuvent implémenter, puis en permettant à un "contexte" (l'objet qui utilise ces stratégies) de changer la stratégie pendant l'exécution.

Exemple d'intérêt :

Permet de changer d'algorithme à la volée sans toucher au reste du code.

Exercice « ex_patron.py »

- Enoncé :
 - Un centre météorologique souhaite développer un système de notifications.
 - Lorsqu'une mise à jour de la température est enregistrée, plusieurs dispositifs doivent être informés automatiquement :
 - Une application mobile
 - Un panneau d'affichage
 - Un site web
 - Implémente le design pattern Observer pour :
 - Créer une classe StationMeteo (le sujet/observable) qui contient la température actuelle.
 - Permettre aux dispositifs (observateurs) d'être ajouté ou retiré.
 - Informer automatiquement tous les observateurs ajoutés à chaque mise à jour de température.
 - Contraintes :
 - Chaque observateur doit implémenter une méthode mise_a_jour(temp) qui affiche un message personnalisé.
 - Tu dois utiliser des classes Python pour structurer le tout.

TP validation des acquis 2

TP validation des acquis 2

- Enoncé :
 - Créer un programme en Python pour enregistrer des commandes clients, en utilisant :
 - Une classe simple,
 - La gestion des erreurs,
 - Un fichier texte pour sauvegarder.
 - Consignes :
 - Créer une classe Commande avec 3 attributs :
 - client (nom du client),
 - produit (nom du produit),
 - quantite (nombre entier strictement positif).
 - Si la quantité est invalide (pas un entier positif), le programme doit lever une exception.
 - Ajouter une méthode afficher() qui retourne la commande sous forme de texte :
 - Exemple : "Client: Alice - Produit: Livre - Quantite: 2"
 - Ajouter une méthode sauvegarder() qui ajoute la commande dans un fichier texte commandes.txt.
 - Créer une fonction charger_commandes() qui affiche toutes les commandes enregistrées dans le fichier.
 - Dans le programme principal :
 - Créer deux commandes valides,
 - Tenter de créer une commande avec une quantité invalide,
 - Sauvegarder les commandes valides,
 - Afficher le contenu du fichier.

TP validation des acquis 2

- Enoncé :

- Résultat attendu dans le terminal :

Création de la commande 1 (valide).

Commande 1 créée avec succès.

Création de la commande 2 (valide).

Commande 2 créée avec succès.

Création de la commande 3 (invalid).

Erreur lors de la création de la commande: Quantité invalide. Elle doit être un entier positif.

Commandes enregistrées :

Client: Alice - Produit: Livre - Quantité: 2

Client: Bob - Produit: Stylo - Quantité: 5

Les modules utiles

Modules

Il existe un grand nombre de modules préprogrammés qui sont fournis d'office avec Python. C'est ce qu'on appelle la bibliothèque standard. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle des bibliothèques.

Math

```
from math import *  
sqrt()  
sin()  
cos()  
pi  
...
```

DateTime

```
from datetime import date  
from datetime import time  
from datetime import datetime
```

module_math.py
module_dates.py

Modules

PyDoc

PyDoc est un module intégré à Python qui permet de consulter et de générer automatiquement de la documentation pour le code Python. Il fonctionne à partir des docstrings que l'on écrit dans les modules, classes, fonctions et méthodes.

Utilité principale

- Consulter la documentation d'un module ou d'une fonction sans quitter le terminal.
- Générer de la documentation HTML lisible dans un navigateur.
- Accéder à l'aide pour des fonctions intégrées ou du code personnalisé.

Utilisation :

`pydoc nom_module` ou `pydoc fonction_builtin`
Affiche la docstring du module ou de la fonction builtin.

`pydoc nom_module.fonction`
Affiche la docstring de la fonction du module `nom_module`.

`pydoc nom_dossier` ou `pydoc dossier.sous_dossier....`
Affiche la docstring du fichier `__init__.py` situé dans le dossier (s'il existe) et la liste de tous les modules dans le dossier.

`pydoc -w ...`
Génère un fichier HTML dans le dossier où l'on se trouve au lieu d'afficher dans la console.

Modules

subprocess

Le module subprocess en Python permet de lancer des processus externes, de communiquer avec eux et de récupérer leur sortie ou leur code de retour. Il est très utile pour exécuter des commandes système depuis un script Python.

Les fonctions principales :

- call() : Exécute une commande, attend qu'elle se termine. Renvoie un code de retour de type int et ne lève pas d'exception. Existe depuis la version 2.5.
- run() : Comme call mais retourne un objet CompletedProcess. Supporte plus de paramètres. Renvoie une exception si le code de retour est différent de 0 et l'option check = True. Existe depuis la version 3.5 de Python. Cette méthode est prévue pour remplacer call().
- getstatusoutput() : S'utilise comme call mais retourne un tuple contenant le code de retour et le résultat.
- Popen() : Exécute une commande, et continue le code python même si la commande n'est pas terminée.

check_output() : Exécute une commande et retourne la sortie (stdout).

• Options courantes :

- stdin, stdout, stderr : Pour gérer les flux d'entrée/sortie.
- shell=True : Exécute la commande dans un shell (attention aux failles de sécurité).
- text=True (ou universal_newlines=True) : Pour avoir des chaînes de caractères au lieu de bytes.
- timeout : pour définir un délai d'expiration.

module_subprocess.py

Modules

sys

Ce module est utile pour interagir avec l'environnement d'exécution et manipuler les entrées/sorties ou les modules dynamiquement. Le module sys en Python fournit des fonctions et variables permettant d'interagir avec l'interpréteur Python. Voici ses principales fonctions et variables :

- Gestion des Arguments de la Ligne de Commande

`sys.argv` : Liste des arguments passés au script (le premier élément est le nom du script).
`sys.exit([code])` : Termine le programme avec un code de sortie optionnel.

- Flux d'Entrée et de Sortie

`sys.stdin` : Gère l'entrée standard.
`sys.stdout` : Gère la sortie standard.
`sys.stderr` : Gère les erreurs et les messages d'alerte.

- Informations sur l'Interpréteur Python

`sys.version` : Retourne la version de Python sous forme de chaîne.
`sys.platform` : Indique le système d'exploitation.
`sys.executable` : Chemin de l'exécutable Python utilisé.

Modules

sys

- Gestion des Modules

`sys.modules` : Dictionnaire des modules chargés.
`sys.path` : Liste des chemins où Python recherche les modules.
`sys.getrecursionlimit()` : Obtient la profondeur maximale de récursion.
`sys.setrecursionlimit(n)` : Définit la profondeur maximale de récursion.

- Gestion de la Mémoire

`sys.getsizeof(objet)` : Retourne la taille d'un objet en mémoire.
`sys.maxsize` : Taille maximale d'un entier en Python.
`sys.getrefcount(objet)` : Retourne le nombre de références à un objet.

Modules

os

Le module os en Python permet d'interagir avec le système d'exploitation. Ce module est souvent utilisé avec shutil pour des manipulations avancées de fichiers et répertoires. Voici ses principales fonctions :

- Gestion des processus

system(command) : Exécute une commande système.

popen(command) : Exécute une commande et retourne son résultat.

getpid() : Retourne l'ID du processus actuel.

getppid() : Retourne l'ID du processus parent.

- Informations système

name : Donne le nom du système d'exploitation ('posix', 'nt', etc.).

uname() : Retourne des informations sur le système (uniquement sous Unix).

environ : Dictionnaire contenant les variables d'environnement.

getlogin() : Retourne le nom de l'utilisateur connecté.

Modules

os

- Gestion des fichiers et répertoires

<code>getcwd()</code>	: Retourne le répertoire de travail actuel.
<code>chdir(path)</code>	: Change le répertoire de travail.
<code>listdir(path)</code>	: Liste les fichiers et dossiers d'un répertoire.
<code>mkdir(path)</code>	: Crée un dossier.
<code>makedirs(path)</code>	: Crée un dossier et ses parents si nécessaires.
<code>remove(path)</code>	: Supprime un fichier.
<code>rmdir(path)</code>	: Supprime un dossier vide.
<code>removedirs(path)</code>	: Supprime un dossier et ses parents s'ils sont vides.

- Manipulation des chemins

<code>path.join(path1, path2)</code>	: Construit un chemin valide.
<code>path.exists(path)</code>	: Vérifie si un chemin existe.
<code>path.isfile(path)</code>	: Vérifie si un chemin est un fichier.
<code>path.isdir(path)</code>	: Vérifie si un chemin est un répertoire.
<code>path.abspath(path)</code>	: Donne le chemin absolu d'un fichier.
<code>path.basename(path)</code>	: Retourne le nom du fichier d'un chemin.
<code>path.dirname(path)</code>	: Retourne le dossier d'un chemin.

`module_os.py`

Modules

pathlib

Le module pathlib de Python v3.4 on supérieur fournit une interface orientée objet pour manipuler des chemins de fichiers et de répertoires. Ce module est particulièrement utile pour remplacer os.path avec une approche plus intuitive et plus puissante. Voici ses principales fonctions et classes :

- Création et manipulation de chemins

Path("chemin/vers/fichier") : Crée un objet Path représentant un chemin (relatif ou absolu).
Path.cwd() : Retourne le chemin du répertoire de travail actuel.
Path.home() : Retourne le chemin du répertoire utilisateur.

- Navigation et vérifications

exists() : Vérifie si le chemin existe.
is_file() : Vérifie si le chemin est un fichier.
is_dir() : Vérifie si le chemin est un répertoire.

- Opérations sur les chemins

name : Nom du fichier avec extension.
stem : Nom du fichier sans extension.
suffix : Extension du fichier.
parent : Répertoire parent du fichier/dossier.
parts : Renvoie les différentes parties du chemin sous forme de tuple.

Modules

pathlib

- Lecture et écriture de fichiers

`read_text()` : Lit le contenu du fichier sous forme de texte.
`read_bytes()` : Lit le contenu du fichier en bytes.
`write_text("contenu")` : Écrit du texte dans le fichier.
`write_bytes(b"contenu")` : Écrit des bytes dans le fichier.

- Création et suppression

`mkdir(parents=True, exist_ok=True)` : Crée un répertoire (y compris les parents s'ils n'existent pas).
`touch()` : Crée un fichier vide.
`unlink()` : Supprime un fichier.
`rmdir()` : Supprime un répertoire vide.

- Gestion des fichiers et répertoires

`iterdir()` : Itère sur les éléments d'un répertoire.
`glob("*.txt")` : Recherche des fichiers correspondant à un motif.
`rename("nouveau_nom")` : Renomme le fichier ou le répertoire.
`replace("nouveau_chemin")` : Déplace le fichier en écrasant s'il existe déjà.

`module_pathlib.py`

Les bases de données

Base de données

Une base de données est une collection organisée de données. Les informations doivent être structurées de manière à pouvoir être facilement accessibles, gérées et mises à jour. Il existe des bases de données relationnelles et non relationnelles.

Une base de données relationnelle est un ensemble structuré de données dans lequel les éléments sont liés entre eux par des relations prédéfinies. Les données sont généralement organisées sous forme de tableaux composés de lignes et de colonnes. Chaque colonne représente un type spécifique d'information, tandis que chaque ligne correspond à un enregistrement unique décrivant une entité. Les relations entre les données permettent de croiser les informations de plusieurs tableaux de manière cohérente.

Pour créer et gérer une base de données relationnelle, on utilise un Système de Gestion de Base de Données Relationnelle (SGBDR). Il s'agit d'un logiciel qui permet d'interagir avec des bases de données relationnelles, comme SQLite, MySQL, SQL Server ou PostgreSQL. Ces systèmes utilisent le langage SQL (Structured Query Language), normalisé par l'ISO, pour accéder aux données. Cependant, certaines fonctionnalités spécifiques varient d'un SGBDR à l'autre.

Une base de données non relationnelle (NoSQL) n'utilise pas le modèle classique de tableaux à lignes et colonnes. Le modèle de stockage est adapté à la nature des données stockées, souvent non structurées ou semi-structurées. Ces bases sont conçues pour traiter des volumes importants de données variées et évolutives. Il en existe plusieurs types, tels que les bases de type document (ex. : MongoDB), colonne (ex. : Cassandra), clé-valeur (ex. : Redis) ou graphe (ex. : Neo4j).

Base de données

Michael Widenius : finlandais créateur de 3 BDD.

MaxDB : c'est la première BDD qu'il a créée. Max est pour le prénom de son fils.

MySQL : c'est sa deuxième BDD pour corrigé les erreurs faites dans MaxDB. My est le prénom de sa première fille. C'est un prénom finlandais courant.

En 2009, Oracle rachète Sun Microsystems pour 7,4 milliards de dollars. Mais Michael Widenius et les dirigeant d'Oracle ne s'entendent pas du tout. Michael Widenius a quitter Oracle et puisque MySQL était open source, il a créé un fork de MySQL.

Ce fork, c'est sa troisième BDD. Elle se nomme MariaDB du prénom de sa deuxième fille qui était juste née.

Base de données

Python DB API 2.0 (PEP 249)

Encourage la conformité entre les modules Python utilisés pour accéder aux bases de données. Cela permet au code d'être plus facilement transférable d'une base de données à l'autre et d'élargir les systèmes de bases de données qui peuvent être utilisés avec Python. Presque tous les modules de base de données Python se conforment à cette interface. Ainsi, une fois que vous avez appris à utiliser les bases de données avec un module, il est facile de reprendre et de comprendre le code des autres modules.

- L'accès à la base de données doit être fourni par le biais d'un objet de connexion qui assure l'interface entre votre programme et la base de données.

```
Connection = connect(param...)
```

- Avec cet objet de connexion, plusieurs opérations peuvent être effectuées :

```
Connection.commit()
```

```
Connection.rollback()
```

```
Connection.close()
```

- Avec cet objet connexion, nous pouvons créer un curseur. Avec l'objet curseur, nous pouvons interagir avec la base de données :

```
Connection = connect(param...)
```

```
Cursor = connection.cursor()
```

```
Cursor.execute(param)
```

```
Cursor.executeMany(param)
```

```
Cursor.fetchone(param)
```

```
Cursor.fetchmany([numOfRows])
```

```
Cursor.fetchall()
```


Base de données

- SQLite

La bibliothèque standard de Python inclut un moteur de base de données relationnelles SQLite, qui a été développé en C, et implémente le standard SQL.

Cela signifie donc que vous pouvez écrire en Python une application contenant son propre SGBDR intégré, sans qu'il soit nécessaire d'installer quoi que ce soit d'autre.

Les instructions d'interaction à la BDDR sont standardisées (cf : PEP 249 sur python.org).

```
>>> import sqlite3
>>> fichierDonnees = "E:/python3/essais/bd_test.sq3"
>>> conn = sqlite3.connect(fichierDonnees)
>>> cur = conn.cursor()
>>> cur.execute("CREATE TABLE membres (age INTEGER, nom TEXT, taille REAL)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(21,'Dupont',1.83)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(15,'Blumâr',1.57)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(18,'Özémir',1.69)")
>>> conn.commit()
>>> cur.close()
>>> conn.close()
```

Base de données

- **Postgres**

Pour Postgres il faut installer le module via `pip install psycopg2`
Puis importer le module `psycopg2`.

Ensuite, le code ressemble beaucoup à celui de SQLite.

```
>>> import psycopg2
>>> psycopg2.connect(dbname = 'template1', user = 'dbuser', password = 'dbpass', host = 'localhost',
port = '5432')
>>> cur = conn.cursor()
>>> cur.execute("CREATE TABLE membres (age INTEGER, nom TEXT, taille REAL)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(21,'Dupont',1.83)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(15,'Blumâr',1.57)")
>>> cur.execute("INSERT INTO membres(age,nom,taille) VALUES(18,'Özémir',1.69)")
>>> conn.commit()
>>> cur.close()
>>> conn.close()
```

Base de données

• Modules pour les BDD Relationnelles

BDD	Module	Remarque
SQLite	-	Pas besoin d'installation
MySQL / MariaDB	my-sql-connector PyMySQL MySQLdb	Officiel, stable, compatible PEP249 Pur Python, facile à utiliser Ancien, à remplacer
PostgresSQL	psycopg2 asyncpg	Le plus populaire et performant Pour les applications asynchrones
Oracle	cx_Oracle	Officiel, maintenu chez Oracle
SQL Server	pyodbc pymssql	Supporte aussi d'autres bases via ODBC Simple à utiliser

• Modules pour les BDD NoSQL

BDD	Module	Remarque
MongoDB	pymongo	Officiel et largement utilisé
Redis	redis	Pour cache, sessions et file d'attente
Cassandra	cassandra-driver	Officiel par DataStax
CouchDB	CouchDB	Interface REST, peut aussi utiliser requests
Elasticsearch	elasticsearch	Pour la recherche full-text
Neo4 (graph)	neo4j	Accès via Cypher

Le gestionnaire de paquets

Gestionnaire de paquets PIP

Dans l'invite de commande de Windows :

```
pip install mon_module      # Installation d'un module (appelé aussi paquet ou dépendance) dans
                             # l'environnement actuel (virtuel ou non).

pip freeze                  # Liste les modules installés.

pip freeze > requirements.txt # Liste les packages de l'environnement actuel dans le fichier
                             # « requirements.txt ».

pip install -r requirements.txt # Installation des paquets listés dans le fichier « requirements.txt » dans
                                # l'environnement actuel.
```

Remarque : Les dépendances ne sont pas automatiquement mises à jour. Si on fait une mise à jour des modules il faut refaire la commande « pip » pour sauvegarder dans un fichier la liste des modules à jour.

« pip » est le successeur du gestionnaire de paquets « easy_install ».

Ces gestionnaires de paquets se base sur le dépôt de paquets PyPi (Python Package Index).

Si aucun environnement virtuel n'est actif, alors les commandes « pip » fonctionnent dans l'environnement réel général de Python installé sur le système d'exploitation. Il est possible de mettre un chemin relatif ou complet avant le « requirements.txt » ainsi que de changer son nom.

Si une librairie/module n'est pas disponible sur PyPi alors généralement c'est des fichiers sources fourni avec une documentation pour savoir comment les installer dans l'environnement Python. Souvent c'est « python setup.py install ».

Créer un exécutable avec PyInstaller

Lignes de commande :

```
pip install PyInstaller # installation de PyInstaller
```

```
PyInstaller -F chemin_complet_ou_relatif\nom_fichier.py # -F pour faire un exécutable indépendant
```

Le fichier exécutable sera dans un dossier « dist » créer à l'emplacement du script.

--distpath chemin : permet de changer le dossier où se situe le fichier exécutable.

--noconsole : permet de ne pas lancer la console quand on exécute le programme (utile pour les interfaces graphiques)

--onefile ou -F : permet de créer un exécutable intégrant tout, même python.

Il n'y a pas de différence entre -F et --onefile. -F est l'option courte. --onefile est l'option longue mais plus explicite.

Les environnements virtuels

Environnements virtuels

Installer Python c'est simple. Mais installer un environnement homogène et performant devient laborieux (un environnement scientifique par exemple) :

- Il faut commencer par isoler son environnement de travail du système.
- Il existe une multitude de librairies et gérer leurs dépendances peut s'avérer difficile.
- Il faut les compiler spécifiquement pour votre système si vous souhaitez exploiter toute la puissance de calcul des processeurs modernes.

Les environnements virtuels Python permettent d'avoir des installations de Python isolées du système et séparées les unes des autres. Cela permet de gérer plusieurs projets sur sa machine de développements, certains utilisant des modules de versions différentes, voir même des versions différentes de Python.

Nous utiliserons le module virtualenv pour la suite du cours.

Installation du package virtualenv :

```
pip install virtualenv
```

Création d'un environnement virtuel :

```
python -m venv monvenv      # avec venv intégrer a Python
```

```
virtualenv monvenv          # avec virtualenv
```

```
# avec virtualenv en spécifiant une version de Python déjà installée sur le PC.
```

```
virtualenv -p "c:\python311\python.exe" monvenv
```


Environnements virtuels

Activation d'un environnement virtuel :

Sous windows :

```
.\monvenv\Scripts\activate
```

Sous linux :

```
. monvenv/bin/activate
```

le « . » sous linux demande à exécuter la commande dans le shell courant au lieu d'un autre shell pour ne pas perdre les variables d'environnements.

Remarque :

lorsque l'environnement virtuel est actif, nous avons le nom de l'environnement virtuel entre parenthèses en début de ligne de console.

Désactivation d'un environnement virtuel :

Sous windows :

```
deactivate
```

Sous linux :

```
. venv/bin/deactivate
```

Remarque :

PyCharm créer directement un environnement virtuel à la création d'un nouveau projet.

Menu, file, settings, python interpreter : la fenêtre indique les librairies installées et leurs versions.

Cela indique aussi si une nouvelle version est disponible.

Interface graphique wxPython

Interface graphique wxPython

• Boîte à Outils GUI

Il existe plusieurs boîtes à outils GUI (Graphical User Interface) pour Python, chacune offrant différentes fonctionnalités et avantages. En voici quelques-unes :

- Tkinter : Tkinter est la boîte à outils GUI standard pour Python, qui fournit une interface Python vers la bibliothèque Tcl/Tk. Tkinter est facile à utiliser et à apprendre, et est inclus dans la distribution standard de Python.
- PyQt : PyQt est une boîte à outils GUI pour Python qui utilise le Framework Qt de Digia. Il offre des fonctionnalités avancées telles que des graphiques vectoriels, une prise en charge multiplateforme et une grande bibliothèque de widgets personnalisables. PyQt est sous licence GPL ou commerciale.
- PySide : PySide est une alternative open source à PyQt, développée par Digia. Il offre des fonctionnalités similaires à PyQt, mais est sous licence LGPL.
- wxPython : wxPython est une boîte à outils GUI pour Python qui utilise le Framework wxWidgets. Il est connu pour sa stabilité, sa compatibilité multiplateforme et sa prise en charge d'un grand nombre de widgets et de contrôles.
- Kivy : Kivy est une boîte à outils GUI open source pour Python qui permet de créer des interfaces utilisateur tactiles et interactives pour les applications mobiles et de bureau. Il est basé sur OpenGL et est disponible sur plusieurs plateformes.

Ces boîtes à outils GUI sont toutes très populaires et ont une grande communauté de développeurs qui les supportent. Le choix de l'une ou l'autre dépendra des besoins et des préférences spécifiques de chaque développeur.

Interface graphique wxPython

- **Pourquoi utiliser wxPython**

- Interfaces graphiques natives (look and feel du système)
- Multiplateforme
- Large choix de widgets
- Bonne documentation et communauté active
- Intégration facile avec Python

- **Installation de wxPython**

wxPython n'est pas fourni par défaut avec python. Pour l'installer on doit passer par la commande pip.

Dans une console en mode administrateur on écrit :

```
pip install wxpython
```

Interface graphique wxPython

- **Structure de base**

Une app wxPython typique comprend :

- Une classe App dérivée de wx.App.
- Une ou plusieurs fenêtres principales (Frame).
- Des panels (wx.Panel) pour organiser le contenu (contient les widgets).
- Des sizers organisent la position et la taille des widgets à l'intérieur des panels (ou d'une fenêtre).
- Des widgets (boutons, textes, listes, etc.).
- Des événements et leurs gestionnaires (callbacks).

Les événements sont au cœur de wxPython.

- Chaque widget déclenche des événements (clic, saisie, fermeture, etc.).
- On connecte (bind) un événement à une méthode.

Les sizers sont des gestionnaires de mise en page.

- wx.BoxSizer : organise les widgets en ligne (horizontal) ou en colonne (vertical).
- wx.GridSizer : organise en grille.
- wx.FlexGridSizer : grille flexible.
- wx.GridBagSizer : grille avancée (plus de possibilité que wxFlexGridSizer).

Interface graphique wxPython

- **Les widgets**

wx.Frame	Fenêtre principale
wx.Panel	Conteneur de widgets, zone de dessin
wx.Button	Bouton cliquable
wx.StaticText	Texte statique (non modifiable)
wx.TextCtrl	Zone de texte (entrée ou affichage de texte)
wx.CheckBox	Case à cocher
wx.RadioButton	Bouton radio (sélection exclusive)
wx.ComboBox	Liste déroulante avec possibilité de saisie
wx.ListBox	Liste déroulante simple
wx.ListCtrl	Liste avancée avec colonnes
wx.Slider	Curseur (glissière)
wx.Gauge	Barre de progression
wx.Choice	Menu déroulant simple
wx.StaticBitmap	Image statique
wx.DatePickerCtrl	Sélecteur de date
wx.TimePickerCtrl	Sélecteur d'heure
wx.MenuBar	Barre de menus
wx.Menu	Menu déroulant
wx.ToolBar	Barre d'outils
wx.StatusBar	Barre de statut en bas de fenêtre
wx.Notebook	Onglets (pour organiser des panneaux)
wx.TreeCtrl	Affichage arborescent
wx.SplitterWindow	Fenêtre divisée en deux panneaux redimensionnables
wx.HyperlinkCtrl	Lien cliquable hypertexte

Interface graphique wxPython

- **Conseils et bonnes pratiques**

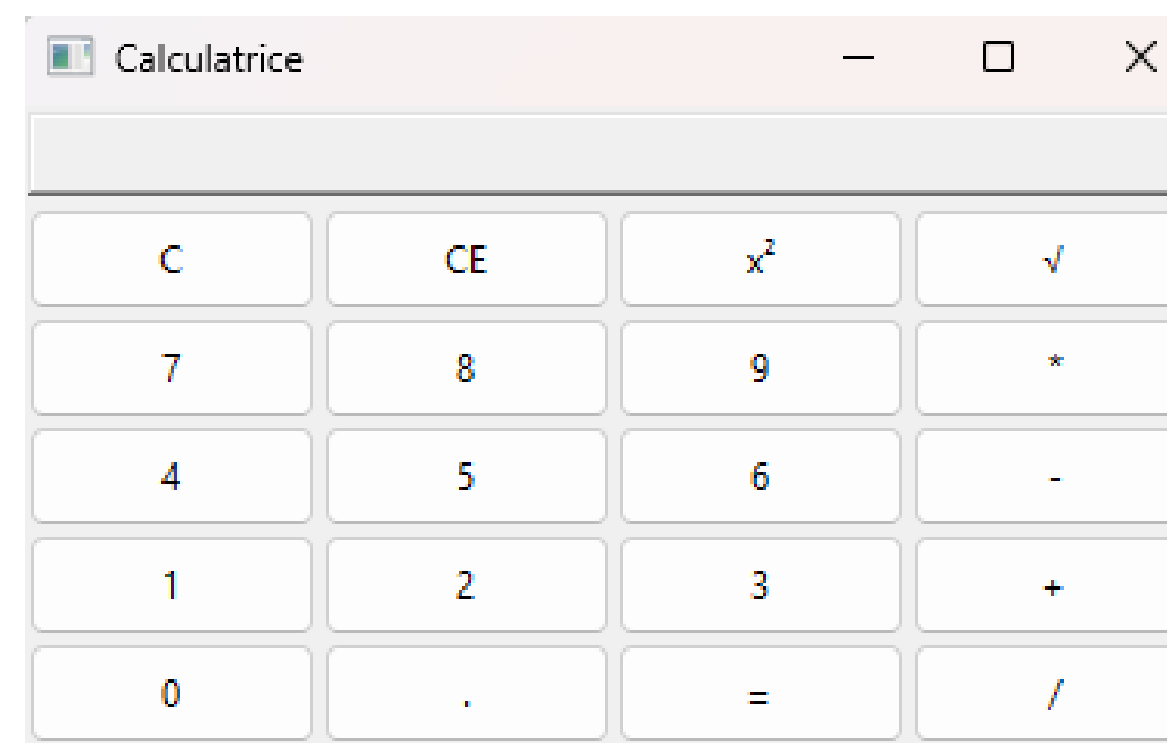
- Toujours utiliser des wx.Panel à l'intérieur des wx.Frame pour un meilleur contrôle des sizers.
- Utiliser les sizers pour gérer la mise en page plutôt que les positions fixes.
- Séparer la logique métier de l'interface (ex : classes différentes).
- Utiliser wx.MessageBox pour les boîtes de dialogue simples.
- Tester l'application sur différents OS pour vérifier le rendu.

- **Ressources utiles**

- Documentation officielle : <https://docs.wxpython.org/>
- Tutoriels : <https://wiki.wxpython.org/Getting%20Started>
- Exemples sur GitHub : <https://github.com/wxWidgets/Phoenix/tree/master/demo>

Exercice « ex_wxPython.py »

- Enoncé :
 - Faire l'apparence d'une calculatrice simple.
 - Exemple d'affichage :



ex_wx_calculatrice

Les tests

Test Unitaire

Nous pouvons faire des tests unitaires avec le module unittest.

Les tests sont regroupés dans des classes de test qui doivent obligatoirement héritées de la classe `unittest.TestCase`.

Généralement, on groupe dans une classe les tests ayant la même classe ou le même module comme point d'entrée. Les tests sont représentés par des méthodes dont le nom commence par `test`.

Pour exécuter une classe de test, il faut utiliser la fonction `unittest.main()`.

Il est possible d'exécuter des instructions avant et après chaque test pour allouer et désallouer des ressources nécessaires à l'exécution des tests. On redéfinit respectivement pour cela les méthodes `setUp()` et `tearDown()`.

Test Unitaire

Pour réaliser les assertions, une classe de test utilise des méthodes d'assertion tel que :

<code>assertEqual(a, b)</code>	→ <code>a == b</code>
<code>assertNotEqual(a, b)</code>	→ <code>a != b</code>
<code>assertTrue(x)</code>	→ <code>bool(x)</code> est vrai
<code>assertFalse(x)</code>	→ <code>bool(x)</code> est faux
<code>assertIs(a, b)</code>	→ <code>a is b</code>
<code>assertIsNot(a, b)</code>	→ <code>a is not b</code>
<code>assertIsNone(x)</code>	→ <code>x is None</code>
<code>assertIsNotNone(x)</code>	→ <code>x is not None</code>
<code>assertIn(a, b)</code>	→ <code>a in b</code>
<code>assertNotIn(a, b)</code>	→ <code>a not in b</code>
<code>assertIsInstance(a, b)</code>	→ <code>isinstance(a, b)</code>
<code>assertNotIsInstance(a,b)</code>	→ <code>not isinstance(a, b)</code>
<code>assertRaises(x)</code>	→ <code>except(x)</code> est vrai

Toutes ces méthodes acceptent le paramètre optionnel `msg` pour passer un message d'erreur à afficher si l'assertion échoue.

mon_projet_unittest

Couverture de code

Une couverture de test, nous permet de connaître le pourcentage de notre code qui est testé. C'est une métrique très utile qui peut vous aider à évaluer la qualité de votre suite de tests. Cela nous donne une idée des parts d'ombre qui peuvent subsister dans notre projet !

Une bonne couverture de test, supérieure à 80 %, est signe d'un projet bien testé et auquel il est plus facile d'ajouter de nouvelles fonctionnalités.

Les outils de couverture de code utilisent un ou plusieurs critères pour déterminer quelle ligne de code a été vérifiée lors de l'exécution de votre suite de tests.

Couverture de code

Il existe 5 façons de mesurer la couverture de code :

- La couverture de ligne : C'est sans doute la méthode la plus utilisée dans les outils de couverture de code. Elle permet de compter le nombre de lignes qui ont été testées.
- La couverture des instructions : Cette méthode peut être confondue avec la couverture de ligne, mais c'est un peu différent. En effet, elle permet de différencier les instructions qu'il y a dans une ligne, et de vérifier si elles sont toutes testées.
- La couverture de branche : Cette méthode compte le nombre de structures de contrôle qui ont été exécutées. Par exemple, la condition if constitue une structure de contrôle, ainsi if et else comptent pour deux branches. Cette métrique permet de vérifier qu'on a pris en compte toutes les possibilités du code.
- La couverture des conditions : Contrairement à la couverture de branche, cette métrique prend en compte l'ensemble des sous-expressions booléennes.
- La couverture des fonctions : Ici, on va compter le nombre de fonctions ou méthodes qui ont été appelées dans le code. Cependant, cette métrique ne prend pas en compte le nombre de lignes dans la fonction.

Couverture de code

- Pour effectuer ces tests nous allons installer la bibliothèque coverage.py en écrivant :

```
pip install coverage
```

Puis le module pytest-cov en tapant :

```
pip install pytest-cov
```

- Syntaxe pour tester un projet situé dans un dossier :

```
pytest --cov=dossier_du_projet
```

- Pour générer un rapport html dans un dossier nommé htmlcov situé dans le dossier du projet testé :

```
pytest --cov=dossier_du_projet --cov-report html
```

- Pour ne pas tester certains fichier ou dossier nous devons créer un fichier .coveragerc et indiquer à l'intérieur de celui-ci ce qu'on ne veut pas tester :

```
[run]  
omit = chemin/*
```

Module pytest

pytest est un framework de test open source pour Python. Il permet :

- d'écrire des tests simples avec peu de code.
- de gérer des tests complexes avec des fixtures, hooks, plugins, etc.
- de produire des rapports clairs et lisibles.

- **Pourquoi l'utiliser ?**

- Syntaxe claire, sans classe obligatoire.
 - Découverte automatique des fichiers/tests.
 - Compatibilité avec unittest et nose.
 - Très extensible via des plugins (pytest-cov, pytest-django, etc.).

- **Installation**

- Dans une fenêtre invite de commande en mode administrateur :

```
pip install pytest
```

- **Exécution des tests**

- Allez dans le dossier du projet via l'invite de commande et exécuter la commande `pytest`. Cela testera tout les fichiers Python qui commence par « `test_` ». Les classes à tester doivent commencer par « `Test` » et les fonctions par « `test_` ».

Module pytest

- **Couverture de code**

Pour faire un rapport de couverture de code avec pytest, on installe d'abord le module « pytest-cov » via la commande dans une fenêtre d'invite de commande en mode administrateur :

```
pip install pytest-cov
```

Ensuite il faut se placer dans le dossier racine du projet et exécuter la commande suivante :

```
pytest --cov=src .\tests
```

.\tests = le dossier des fichiers de tests dans le projet (optionnel selon le nom du dossier).

Pour obtenir un rapport html (un dossier htmlcov est créé) :

```
pytest --cov=src --cov-report=html .\tests
```

Pour ouvrir le rapport html à partir de la console :

```
start htmlcov/index.html
```

Pour obtenir un rapport dans la console et un fichier texte :

```
pytest --cov=src --cov-report=term --cov-report=term-missing > coverage.txt
```

- **Fichiers optionnels du projet**

Le fichier « requirements.txt » sert à connaître les modules utilisés dans le projet.

Le fichier « conftest.py » permet de mettre des données réutilisables pour tous les fichiers de tests.

Module DocTests

Les DocTests en Python permettent d'écrire des tests directement dans la docstring d'une fonction, d'une classe ou d'un module.

L'idée est simple : la documentation montre des exemples d'utilisation, et Python peut les exécuter automatiquement pour vérifier que le résultat attendu correspond bien.

Pour lancer les tests, il y a 2 méthodes principales. La première par la console avec la ligne de commande suivante :

```
python -m doctest -v mon_fichier.py
```

Le -v est pour verbose, affiche tous les tests et leur statut.

La deuxième méthode c'est de lancer les tests directement dans le code du fichier en important le module doctest puis en exécutant la méthode testmod().

Les résultats doivent correspondre exactement (y compris espaces et format).

Les exceptions peuvent être testées et on peut utiliser « ... » pour ignorer une partie du message.

Avantages

Parfait pour tester de petits exemples.
Sert à la fois de documentation et de test.
Intégré à Python.

Inconvénients

Peu adapté si beaucoup de cas ou complexe.
Vérifie uniquement l'exemple dans la docstring.

doctests.py

Module Pylint

Pylint est un outil d'analyse statique de code source pour Python (il ne voit pas les divisions par zéro possible par exemple). Son principal objectif est d'identifier des problèmes potentiels, des erreurs de programmation, des non-conformités aux conventions de codage et des pratiques non recommandées dans le code Python. Pour utiliser Pylint nous devons installer le module en exécutant la ligne de commande suivante :

```
pip install pylint
```

En mode console, il suffira d'écrire la ligne de commande suivante :

```
pylint nom_fichier.py
```

Un rapport s'affiche dans la console pour indiquer la totalité des erreurs et avertissements.

Dans Visual Studio Code, il faut installer l'extension du même nom pour obtenir les mêmes messages directement dans l'IDE.

Remarque :

Il existe d'autres modules permettant de faire les mêmes choses, tel que :

Flake8, MyPy, autopep8, BlackFormatter ou yapf

Module PyChecker

PyChecker est un outil équivalent à Pylint, mais il n'est compatible qu'avec Python v2. Pour installer le package, il faut exécuter la ligne de commande suivante :

```
pip install PyChecker
```

Une fois PyChecker installé, vous devez configurer VSCode pour l'utiliser. Pour ce faire, vous pouvez ajouter une configuration dans votre fichier settings.json. Pour accéder à ce fichier, ouvrez le menu "Fichier" dans VSCode, sélectionnez "Préférences", puis "Paramètres".

Ajoutez une section pour la configuration de PyChecker dans settings.json :

```
"python.linting.pycheckerEnabled": true,  
"python.linting.pycheckerPath": "/chemin/vers/pychecker"
```

Assurez-vous de remplacer "/chemin/vers/pychecker" par le chemin réel vers l'exécutable PyChecker.

Exercice « ex_unittest.py »

- Enoncé :
 - Importer la fonction add du module "a_tester.py"
 - Les tests suivants doivent être couverts :
 - L'addition de deux entiers positifs.
 - L'addition d'un entier et de zéro.
 - L'addition de deux entiers négatifs.
 - L'addition d'un entier négatif et d'un entier positif.

ex_unittest

La programmation parallèle

Programmation parallèle

- **Multithreading**

Le multithreading est une fonctionnalité du langage Python qui nous permet de créer et d'exécuter des threads (des tâches) en parallèle dans un même processus. Les threads sont des unités d'exécution qui se partagent le même espace mémoire. Ils font parti d'un même processus et peuvent se partager des variables, etc... Cela peut considérablement accélérer l'exécution de votre programme même si cela peut amener des défis de coordination. Attention, 2 threads de 2 processus distinct sont isolés l'un de l'autre et ne partagent rien. Par contres en cas de plantage de l'un, cela n'a pas d'effet sur l'autre.

Ces défis de coordination font partie de ce qui rend l'écriture de programmes parallèles plus difficile que celle de simples programmes séquentiels.

L'exécution parallèle augmente le débit global d'un programme, ce qui permet de décomposer les tâches importantes pour les accomplir plus rapidement ou d'accomplir plus de tâches dans un temps donné.

Certains problèmes informatiques sont si vastes ou complexes qu'il n'est pas pratique, ou pas possible, de les résoudre avec un seul ordinateur.

Le calcul parallèle nécessite du matériel en parallèle avec plusieurs processeurs pour exécuter différentes parties d'un programme en même temps.

Il est fortement conseillé d'utiliser des noms pour les threads au lieu de th1, th2 etc... pour une meilleure maintenabilité du code.

`thread_simple.py`

Programmation parallèle

• Processeur

SISD	Single Instruction, Single Data	Ordinateur séquentiel doté d'une seule unité de traitement.
SIMD	Single Instruction, Multiple Data	Les processeurs exécutent la même instruction à un moment donné, mais ils peuvent chacun opérer sur des éléments de données différents.
MISD	Multiple Instruction, Single Data	Chaque unité de traitement exécute indépendamment sa propre série d'instructions, mais tous ces processeurs travaillent sur le même flux de données.
MIMD	Multiple Instruction, Multiple Data	Chaque unité de traitement peut exécuter une série différente d'instructions et, en même temps, chacun de ces processeurs peut travailler sur un ensemble différent de données.

• Mémoire

Shared Memory	Tous les processeurs ont accès à la même mémoire dans le cadre d'un espace d'adressage global. L'architecture la plus utilisée est le SMP où tous les processeurs ont un accès égal à la mémoire.
Distributed Memory	Chaque processeur possède sa propre mémoire locale avec son propre espace d'adressage. Le concept d'espace d'adressage global n'existe donc pas.

Programmation parallèle

- **MultiProcessing vs MultiThreading**

Lorsqu'un ordinateur exécute une application, l'instance du programme qui s'exécute est appelée un processus. Un processus est constitué du code du programme, de ses données et d'informations sur son état. Chaque processus est indépendant, possède son propre espace d'adressage et sa propre mémoire. Un ordinateur peut avoir des centaines de processus actifs en même temps, et le travail d'un système d'exploitation est de les gérer tous. Les processus communiquent via des IPC (Inter Processus Communication) comme de la mémoire partagée (via configuration de l'OS), les pipes (fichiers) ou le réseau. À l'intérieur de chaque processus, il existe un ou plusieurs sous-éléments plus petits appelés threads. Les threads sont les unités de base que le système d'exploitation gère, et il alloue du temps sur le processeur pour les exécuter réellement.

Il est possible d'écrire des programmes parallèles qui utilisent plusieurs processus travaillant ensemble vers un objectif commun, ou des programmes qui utilisent plusieurs threads dans un seul processus.

Les threads qui appartiennent à un même processus se partagent l'espace d'adressage du processus, ce qui leur donne accès aux mêmes ressources et à la même mémoire, y compris le code exécutable et les données du programme. Mais cela peut aussi créer des problèmes si les threads ne coordonnent pas leurs actions. Le partage des ressources entre des processus distincts n'est pas aussi facile que le partage entre les threads d'un même processus. Parce que chaque processus existe dans son propre espace d'adressage.

Programmation parallèle

- **Concurrence vs Exécution parallèle**

La concurrence désigne la capacité d'un algorithme ou d'un programme à être décomposé en différentes parties qui peuvent être exécutées indépendamment les unes des autres sans affecter le résultat final.

La concurrence concerne la manière dont un programme est structuré et la composition de processus s'exécutant indépendamment.

Concurrence	Parallélisme
Structure d'un programme capable de traiter plusieurs choses à la fois.	Exécution simultanée, c'est-à-dire le fait de faire plusieurs choses à la fois.

La concurrence permet à un programme de s'exécuter en parallèle, à condition de disposer du matériel nécessaire, mais un programme concurrent n'est pas intrinsèquement parallèle.

Programmation parallèle

- **Global Interpreter Lock**

L'utilisation de threads pour gérer des tâches concurrentes en Python est assez simple. Cependant, l'interpréteur Python ne permettra pas à ces threads concurrents de s'exécuter simultanément et en parallèle, en raison d'un mécanisme appelé Global Interpreter Lock ou GIL (verrou global de l'interpréteur).

Le GIL est un mécanisme de Python qui empêche plusieurs threads Python de s'exécuter en même temps. Cela signifie que si votre programme est écrit pour avoir 10 threads concurrents, un seul d'entre eux peut s'exécuter à la fois pendant que les neuf autres attendent leur tour.

Il y a eu plusieurs propositions visant à éliminer le GIL de Python en raison de son impact négatif sur les performances parallèles, mais dans la plupart des cas, les avantages du GIL l'emportent sur ses inconvénients. Il existe d'autres implémentations de Python qui n'utilisent pas le GIL :

Jython, qui est basé sur Java,
IronPython, qui est basé sur .NET,
et PyPy-STM

Mais Python est l'interpréteur par défaut et le plus populaire, de sorte que le GIL demeure un élément litigieux de Python pouvant avoir un impact négatif sur les performances multithread si votre application est liée au processeur et qu'elle passe la plupart de son temps à effectuer des calculs intensifs pour le processeur. La version 3.13 (fin 2024) permettra de désactiver le GIL de façon expérimentale.

Voir <https://docs.python.org/3/library/threading.html>

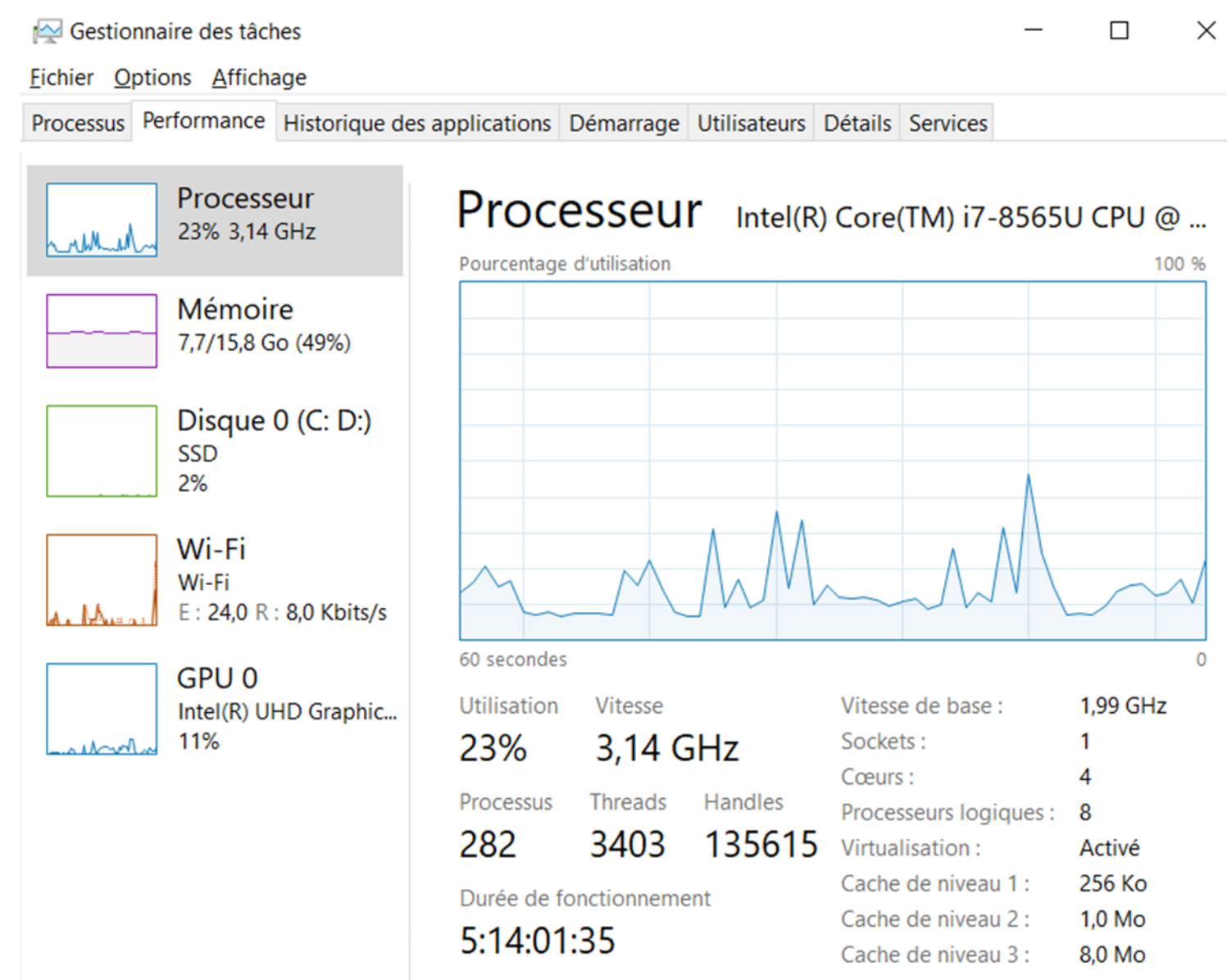
Programmation parallèle

• Multithreading avec Python

On utilise le module threading de Python pour créer plusieurs threads simultanés.

<https://docs.python.org/3/library/threading.html>

Pour avoir un aperçu sur l'utilisation des ressources, le gestionnaire des tâches peut être utilisé.



multiple_threads.py

Programmation parallèle

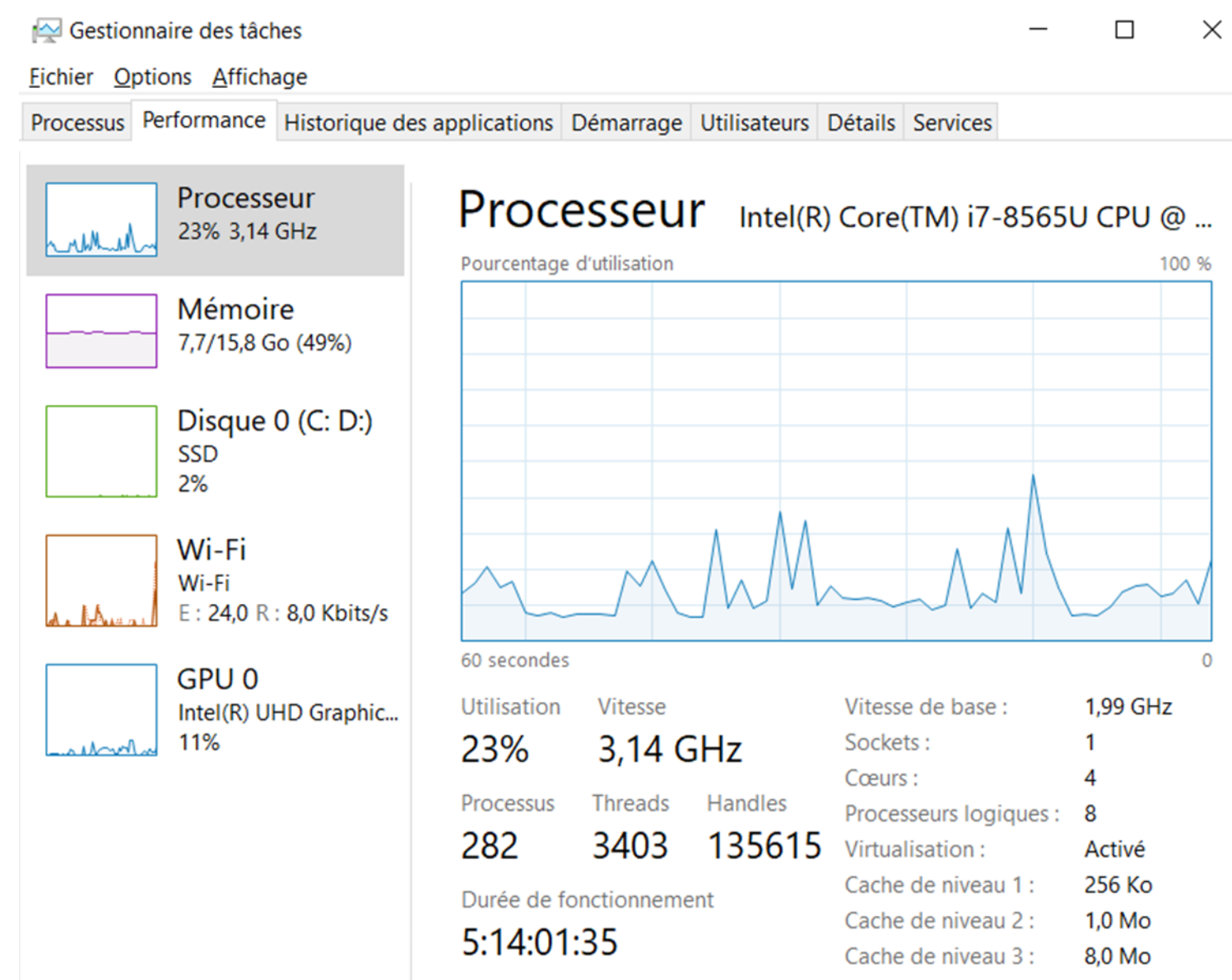
• Multiprocessing avec Python

Pour tirer parti des processeurs multiples et réaliser une véritable exécution parallèle en Python, plutôt que de structurer notre programme pour utiliser plusieurs threads, nous devons utiliser plusieurs processus.

Le module multiprocessing de Python fournit une API pour la création de processus supplémentaires qui ressemble beaucoup au module threading.

Pour avoir un aperçu sur l'utilisation des ressources, le gestionnaire des tâches peut être utilisé.

`multiple_processus.py`



Programmation parallèle

- **L'ordonnanceur**

Les threads ne s'exécutent pas simplement quand ils le veulent. Un ordinateur peut avoir des centaines de processus avec des milliers de threads qui veulent tous avoir leur tour pour fonctionner sur une poignée de processeurs.

Le système d'exploitation comprend un ordonnanceur qui contrôle quand les différents threads et processus obtiennent leur tour d'exécution sur le CPU. L'ordonnanceur (scheduler) permet à plusieurs programmes de s'exécuter simultanément sur un seul processeur. Lorsqu'un processus est créé et prêt à être exécuté, il est chargé en mémoire et placé dans la file d'attente des processus prêts.

ordonnanceur.py

Programmation parallèle

- **Data Race**

L'un des principaux défis de l'écriture de programmes concurrents consiste à identifier les dépendances possibles entre les threads pour s'assurer qu'ils n'interfèrent pas entre eux et ne causent pas de problèmes. Les Data Race sont un problème courant qui peut se produire lorsque :

- Deux ou plusieurs threads accèdent simultanément au même emplacement en mémoire.
- Au moins un de ces threads écrit à cet emplacement pour en modifier la valeur.

On utilise les techniques de synchronisation pour protéger un programme contre les Data Race. Chaque fois que plusieurs threads lisent et écrivent simultanément une ressource partagée, cela crée un comportement incorrect potentiel, comme un effacement de données, mais cela peut être empêché en identifiant et en protégeant les sections critiques du code.

Une section critique ou une région critique est une partie d'un programme qui accède à une ressource partagée, telle qu'une mémoire structurée en données ou un périphérique externe, et qui peut ne pas fonctionner correctement si plusieurs threads y accèdent simultanément.

Une incrémentation par exemple, est une section critique, parce que c'est une opération qui englobe trois sous opérations : Lecture, Modification, Affectation.

Programmation parallèle

- **Mutex (Lock)**

- Mécanisme pour implémenter l'exclusion mutuelle.
- Un seul thread ou processus peut être en possession du verrou (lock) à la fois.
- Limite l'accès à une section critique.

Comme les threads peuvent être bloqués et coincés en attendant qu'un thread de la section critique finisse de s'exécuter, il est important de garder la section de code protégée par le mutex aussi courte que possible.

Le mutex permet de synchroniser les ressources.

Remarque :

En python il existe le mot clé `with` qui correspond à l'utilisation d'un gestionnaire de contexte (context manager)

Il utilise les dunders `__enter__` (acquisition de ressource) et `__exit__` (libération de ressource) de la classe appelée. Cela assure la libération de la ressource même en cas de plantage dans le `with`.

`mutex.py`

Programmation parallèle

- **Sémaphore**

Un sémaphore est un mécanisme de synchronisation qui peut être utilisé pour contrôler l'accès aux ressources partagées, comme un mutex.

Contrairement à un mutex, un sémaphore peut permettre à plusieurs threads d'accéder à la ressource en même temps.

Un sémaphore inclut un compteur (un mécanisme de jeton) pour suivre le nombre de fois où il a été acquis ou libéré. Tant que la valeur du compteur du sémaphore est positive, n'importe quel thread peut acquérir le sémaphore, ce qui diminue la valeur du compteur. Si le compteur atteint zéro, les threads qui essaient d'acquérir le sémaphore seront bloqués et placés dans une file d'attente pour attendre qu'il soit disponible.

Si les valeurs possibles d'un sémaphore sont limitées à zéro ou un, zéro représentant un état verrouillé et un représentant un état déverrouillé. On parle d'un sémaphore binaire et il correspond beaucoup à un mutex.

`semaphore.py`

Programmation parallèle

• Mutex vs Sémaphore

Mutex	Sémaphore
Un mutex utilise un mécanisme de verrouillage, c'est-à-dire que si un processus veut utiliser une ressource, il verrouille la ressource, l'utilise et la libère ensuite.	Un sémaphore utilise un mécanisme de signalisation où les méthodes wait() et signal() sont utilisées pour montrer si un processus libère ou prend une ressource.
Un mutex est un objet.	Un sémaphore est une variable entière.
Un mutex ne peut être déverrouillé que par le même thread qui l'a verrouillé à l'origine.	Un sémaphore peut être acquis et libéré par différents threads. N'importe quel thread peut incrémenter la valeur du sémaphore en le libérant ou tenter de décrémenter sa valeur en l'acquérant.
Un objet mutex permet à plusieurs threads de processus d'accéder à une seule ressource partagée, mais un seul à la fois.	Le sémaphore permet à plusieurs threads de processus d'accéder à l'instance finie de la ressource jusqu'à ce qu'elle soit disponible.

Exercice « ex_threads.py »

- Enoncé :
 - On veut écrire un programme qui : Lance 3 threads en parallèle.
 - Chaque thread doit afficher son nom et compter de 1 à 5, avec une pause d'1 seconde entre chaque nombre.
 - Le programme principal doit attendre la fin de tous les threads avant d'afficher "Programme terminé".

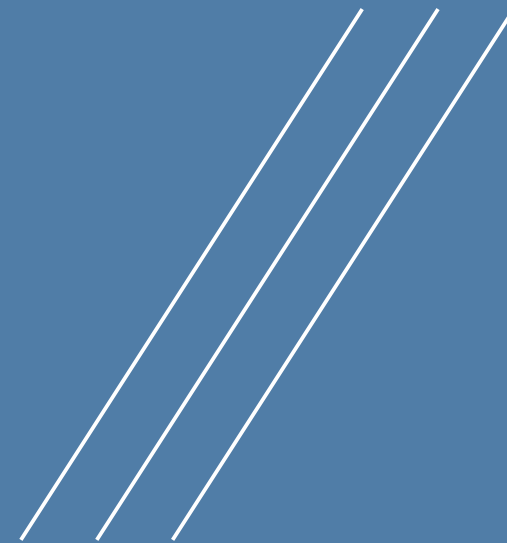
ex_threads

Remerciements

Remerciements

- Votre formateur Xavier TABUTEAU vous remercie d'avoir participé à cette formation.
- Seriez-vous intéressé par une autre formation ?
- Si oui, laquelle ?

PYTHON



Présenté par
Xavier TABUTEAU