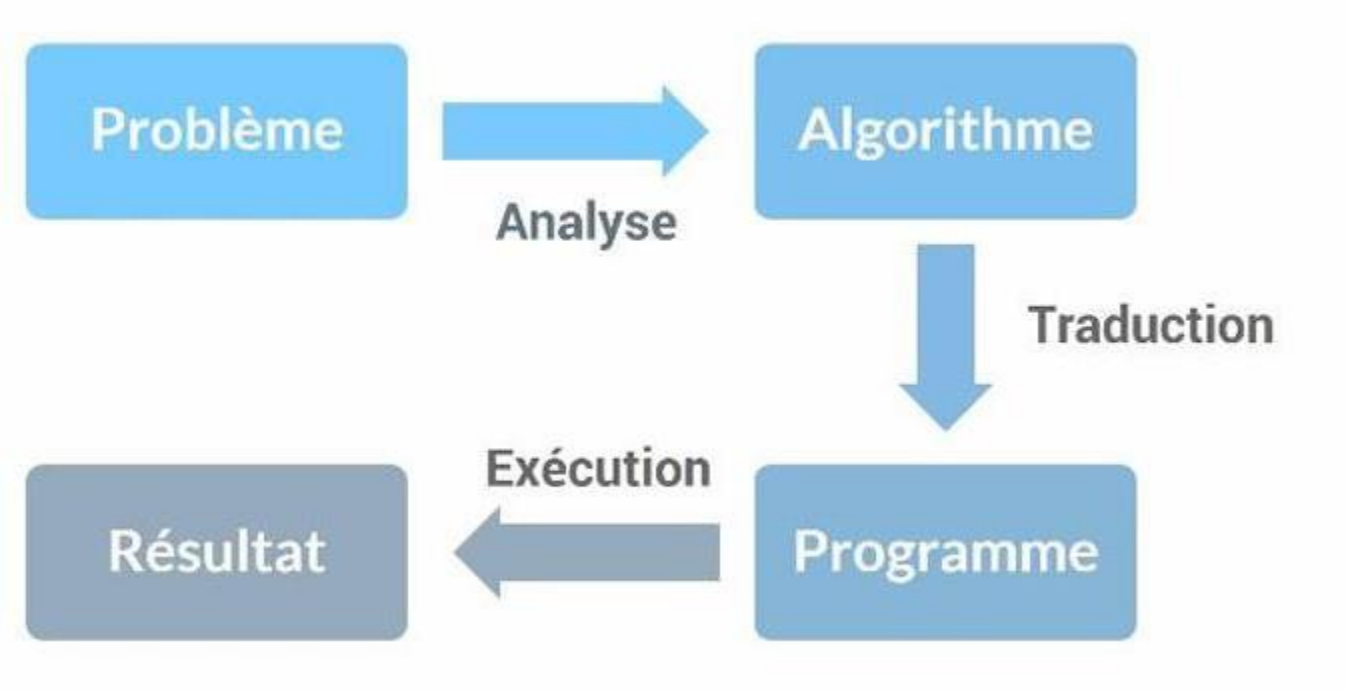
Programmation en Langage Python

Pr. Abdelali El Gourari

Email: a.elgourari.ced@uca.ac.ma

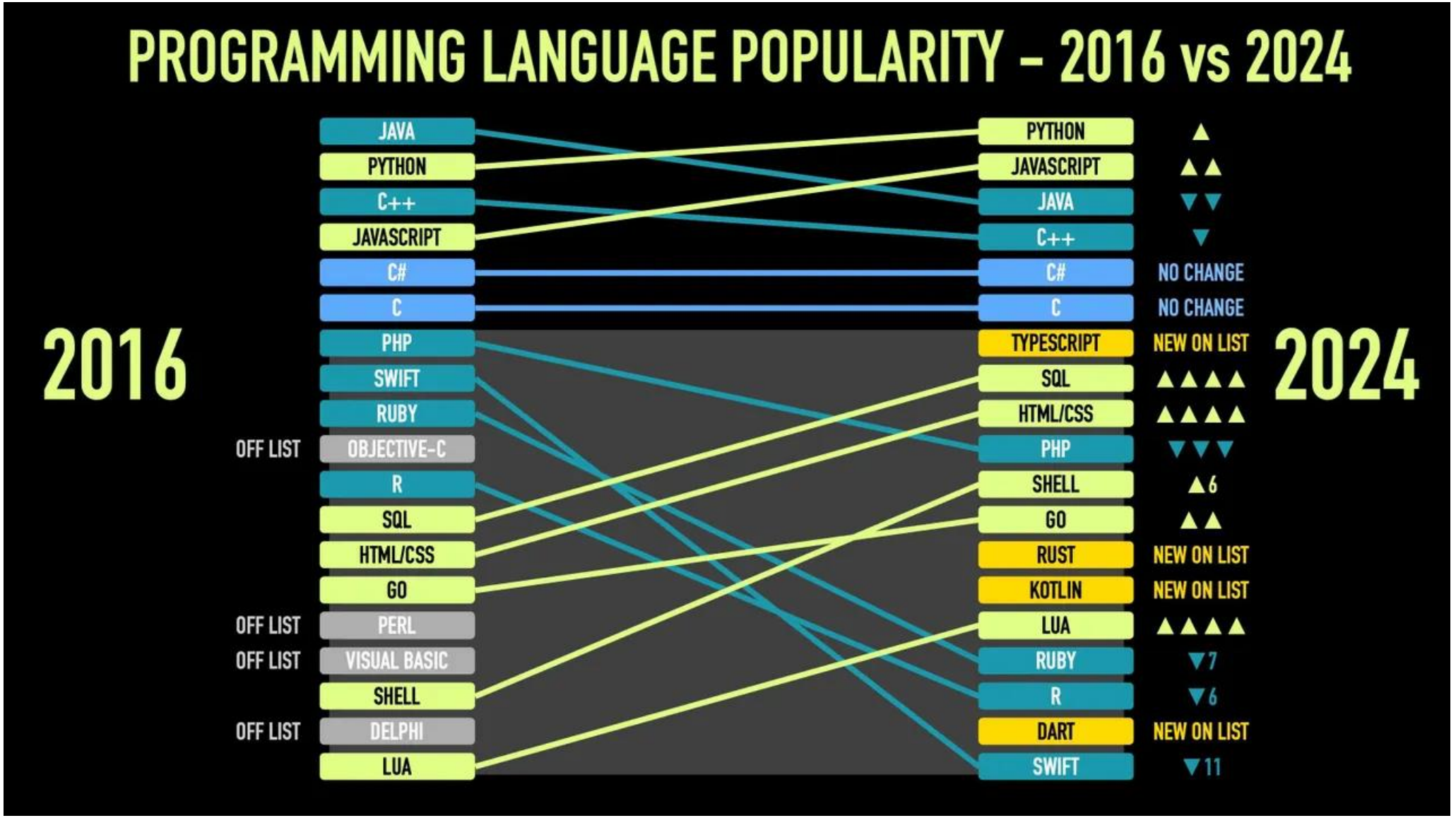
Phases de résolution d’un problème en programmation



Pourquoi Python

Populaire, haut-niveau,
à usage général, gratuit et open-source

Le plus populaire depuis 2024



https://www.google.com/search?sca_

Python, un langage haut niveau

C++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

JAVA

```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

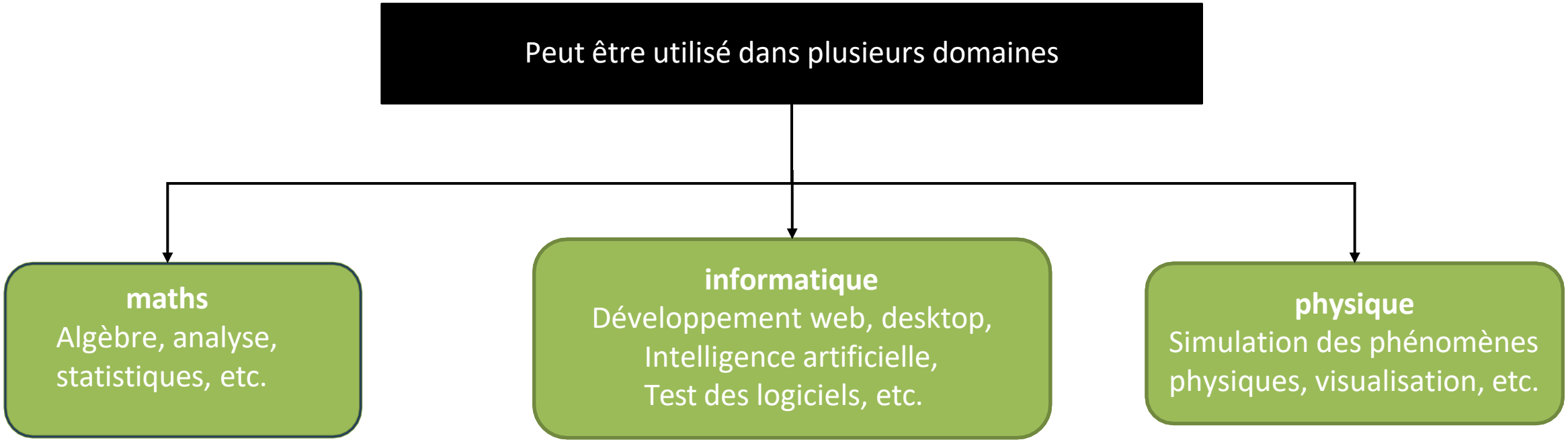
PYTHON

```
>>> print("Hello World!")
Hello World!
```

un langage haut niveau : proche du langage humain que du langage machine

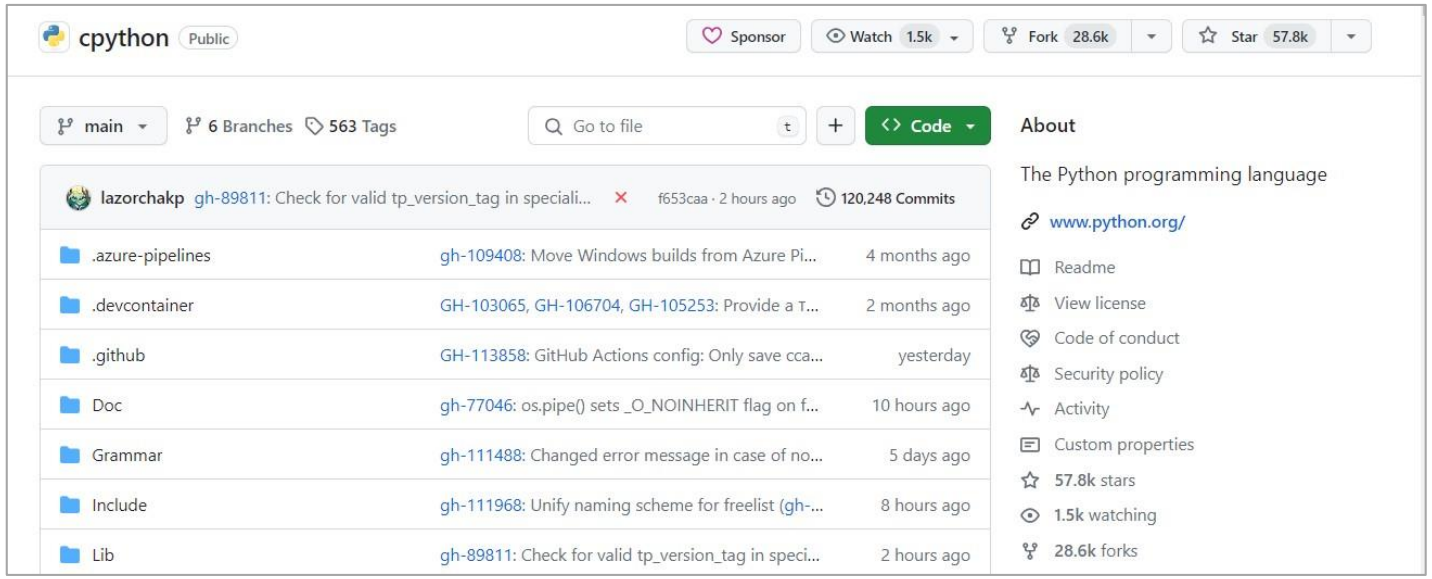
À usage général

Peut-importe la spécialité dans laquelle vous allez continuer ➔ Python va vous aider



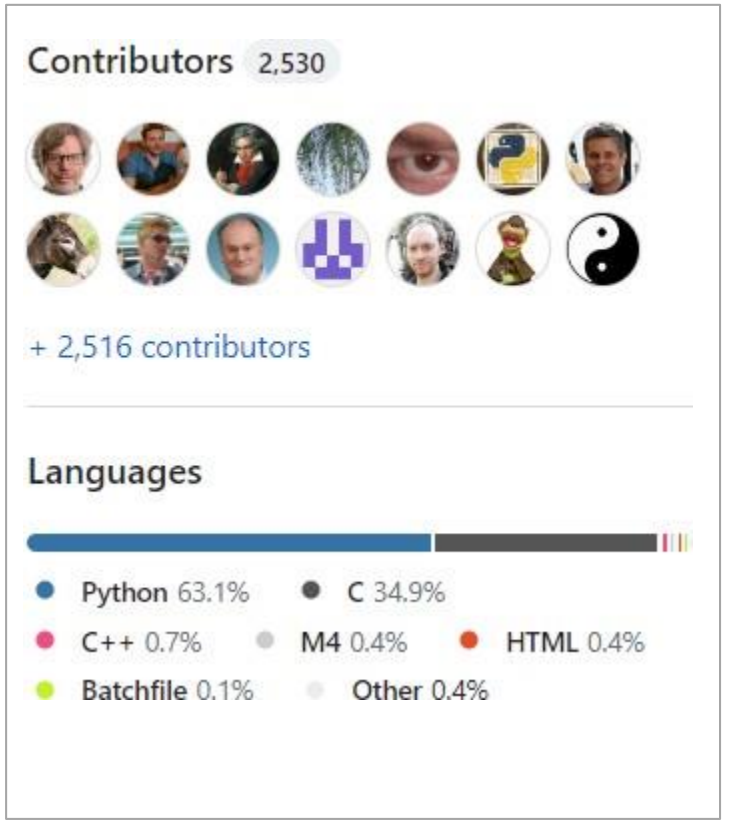
Open source

Tout le monde peut voir / consulter le code de Python lui-même



https://github.com/python/cpython

Guido Van Rossum,
Créateur du langage Python



Exécution

Interprété (Python)

Compilé (C, C++)

Un peu des deux (Java)

Compilé VS interprété

Processus de compilation :

- Le code source est traduit en langage machine ou en un code intermédiaire par un compilateur avant l'exécution.
- Ce processus aboutit à un fichier exécutable autonome ou un binaire qui peut être directement exécuté par le matériel de l'ordinateur.

Exécution :

- Le code compilé s'exécute directement sur le matériel de l'ordinateur, ce qui conduit généralement à une exécution plus rapide.
- L'étape de compilation se produit avant l'exécution du programme.

Exemples de langages compilés :

- **C**, C++, Fortran.

Avantages :

- Généralement une exécution plus rapide car le code est déjà traduit en langage machine.
- Le binaire compilé peut être distribué sans révéler le code source.

Inconvénients :

- Cycle de développement plus long car la compilation est une étape distincte. (Pour voir le résultat du code, on doit compilé)
- Des fichiers binaires spécifiques peuvent être nécessaires pour chaque système d'exploitation différent.

Processus d'interprétation :

- Le code source est exécuté ligne par ligne ou déclaration par déclaration par un interpréteur.
- L'interpréteur traduit et exécute le code à la volée, sans produire de fichier exécutable autonome.

Exécution

- Le code est exécuté au moment de l'exécution, et les modifications apportées au code peuvent être immédiatement testées sans étape de compilation séparée.

Exemples :

- **Python**, JavaScript, PHP.

Avantages :

- Cycle de développement plus court car il n'y a pas besoin d'une étape de compilation distincte.
- Le code peut être facilement modifié et testé pendant le développement.

Inconvénients :

- Généralement une exécution plus lente par rapport aux langages compilés.
- Nécessite la présence de l'interpréteur sur le système cible.
- Peut exposer le code source, car il est souvent distribué dans sa forme originale.

Compilé VS interprété

Analogie avec un livre

Compilé : c'est comme si vous avez un livre en anglais, et pour le lire, vous traduisez tout le livre en arabe (la langue que vous comprenez), et puis vous lisez depuis la version arabe.

Vous pouvez imaginez maintenant pourquoi c'est très rapide lors de l'exécution, car la lecture se fait depuis la version la plus proche de ce qu'on comprend.

interprété : c'est comme si vous avez un livre en anglais, et pour le lire, vous traduisez la première phrase, vous l'exécuter, vous traduisez la deuxième phrase, vous l'exécuter, etc. jusqu'à finir tout le livre, et puis vous supprimez les traductions des lignes.

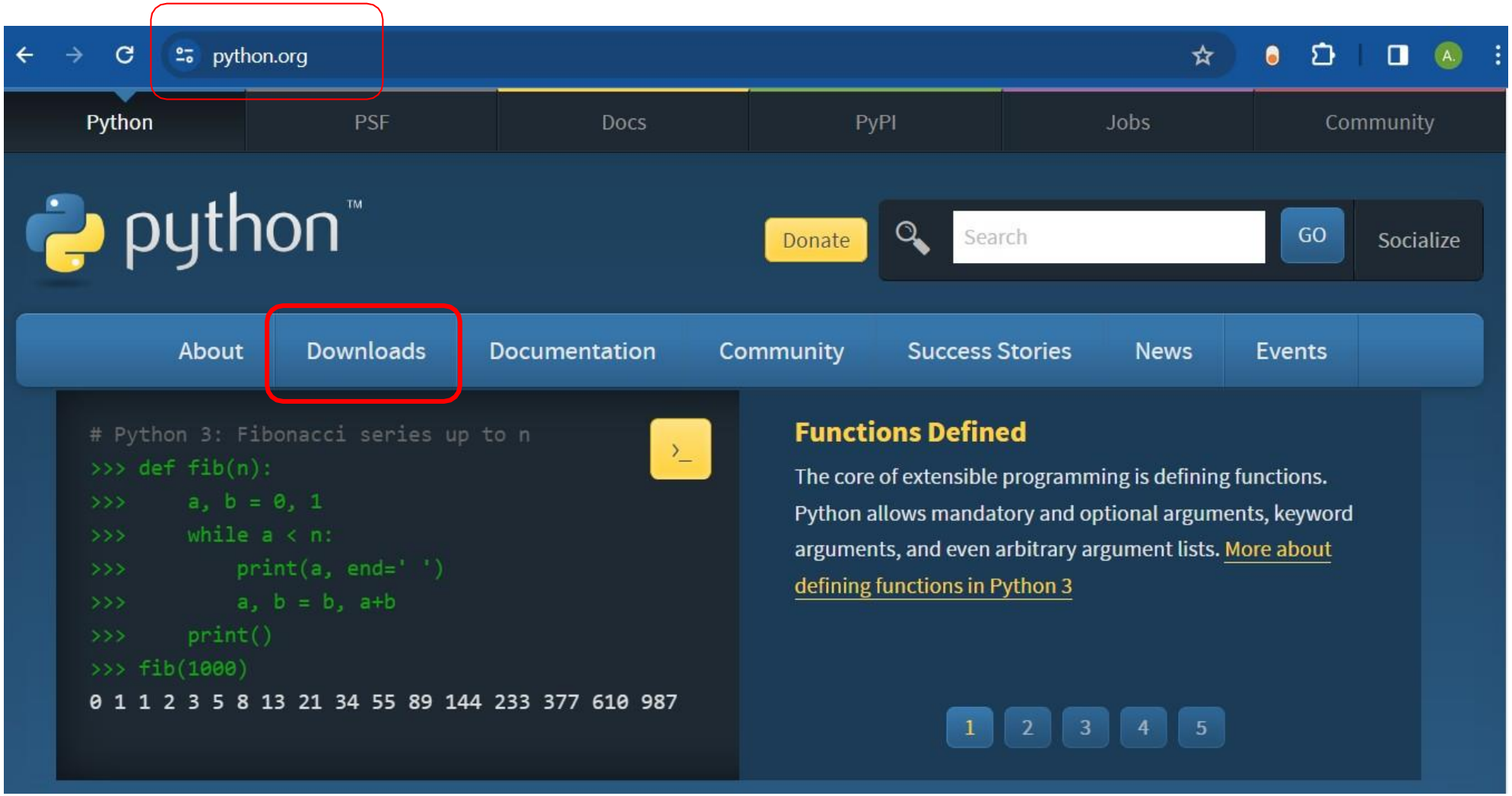
Du coup, à chaque fois on veut faire l'exécution, on doit passer par la traduction, d'où le temps d'exécution relativement long.

Mais c'es plus facile à débuguer (trouver et éliminer les erreurs) car on peut voir l'exécution de chaque ligne du code de manière isolée.

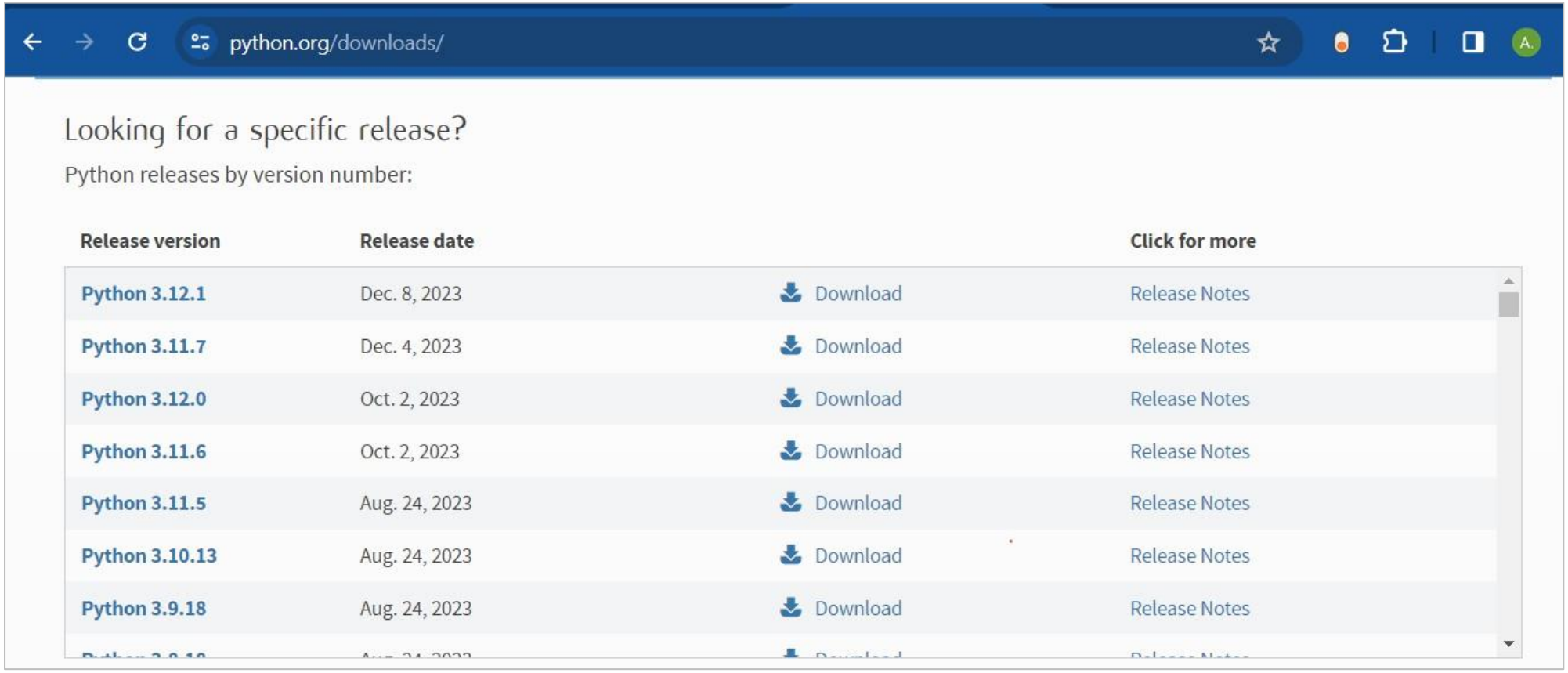
Installation

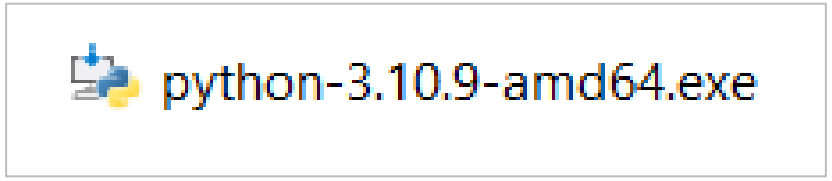
Télécharger Python.exe, version,
Étapes d’installation,
IDLE,
IDE modernes

Télécharger Python.exe

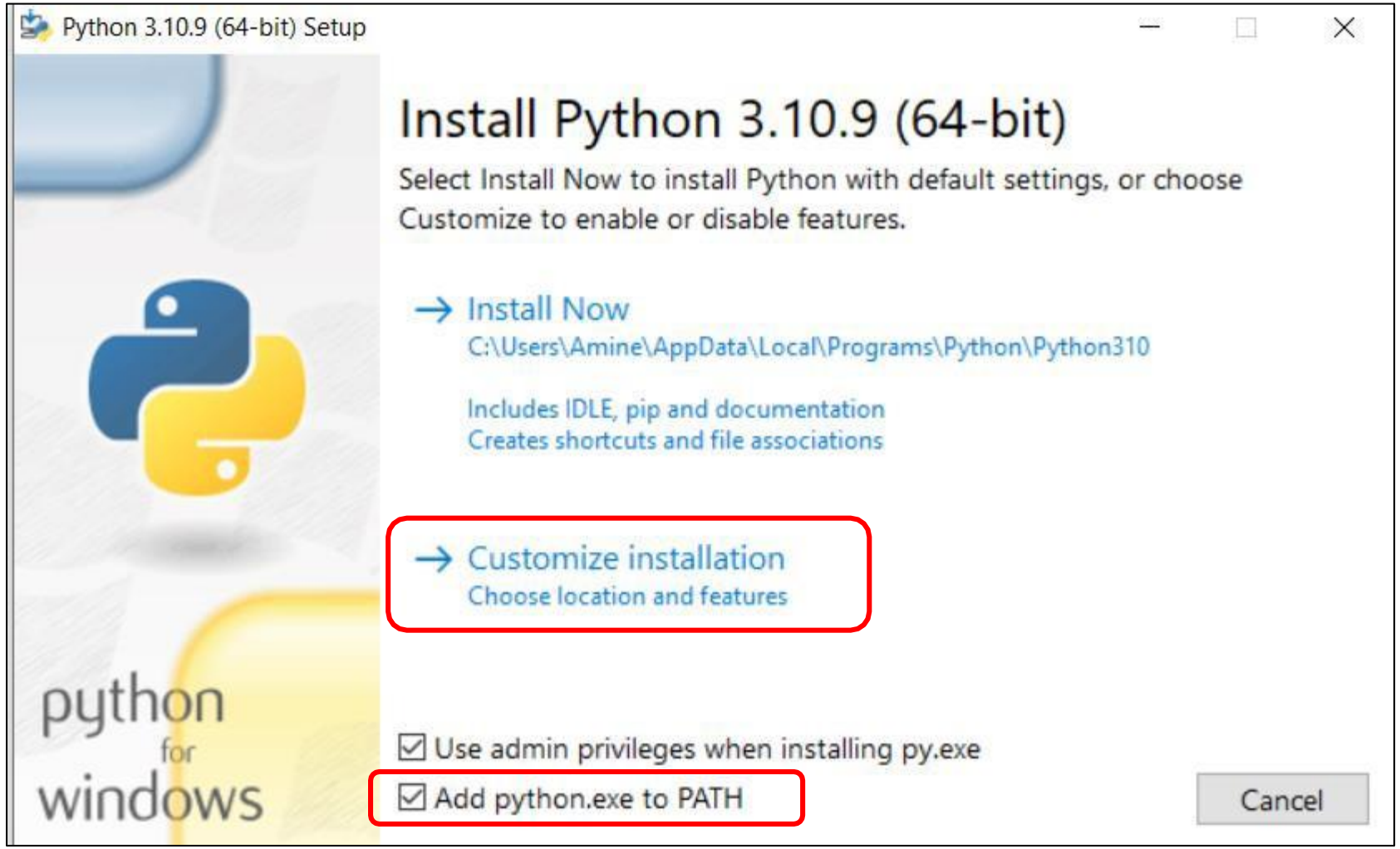


Télécharger et cliquer sur le « .exe »

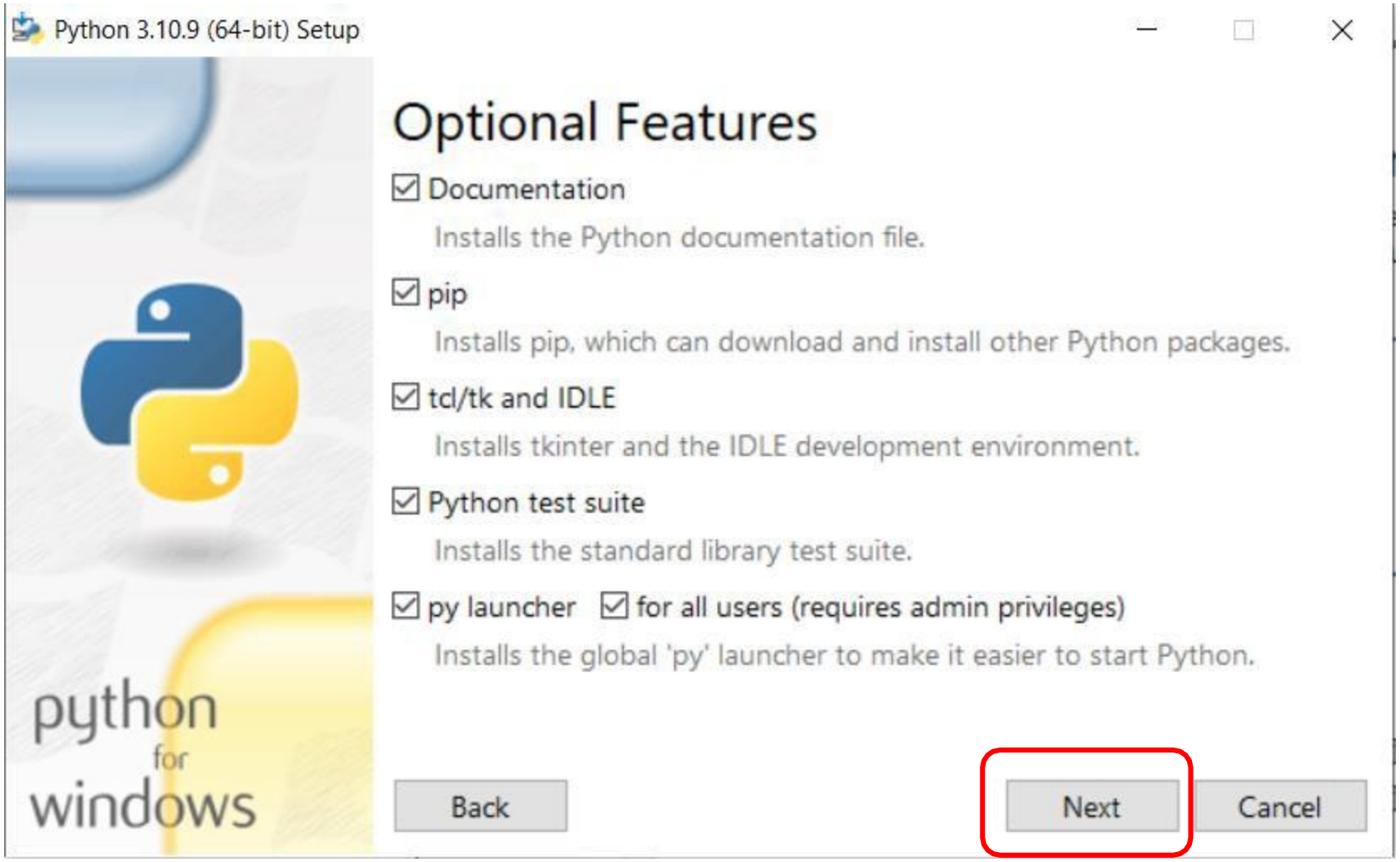


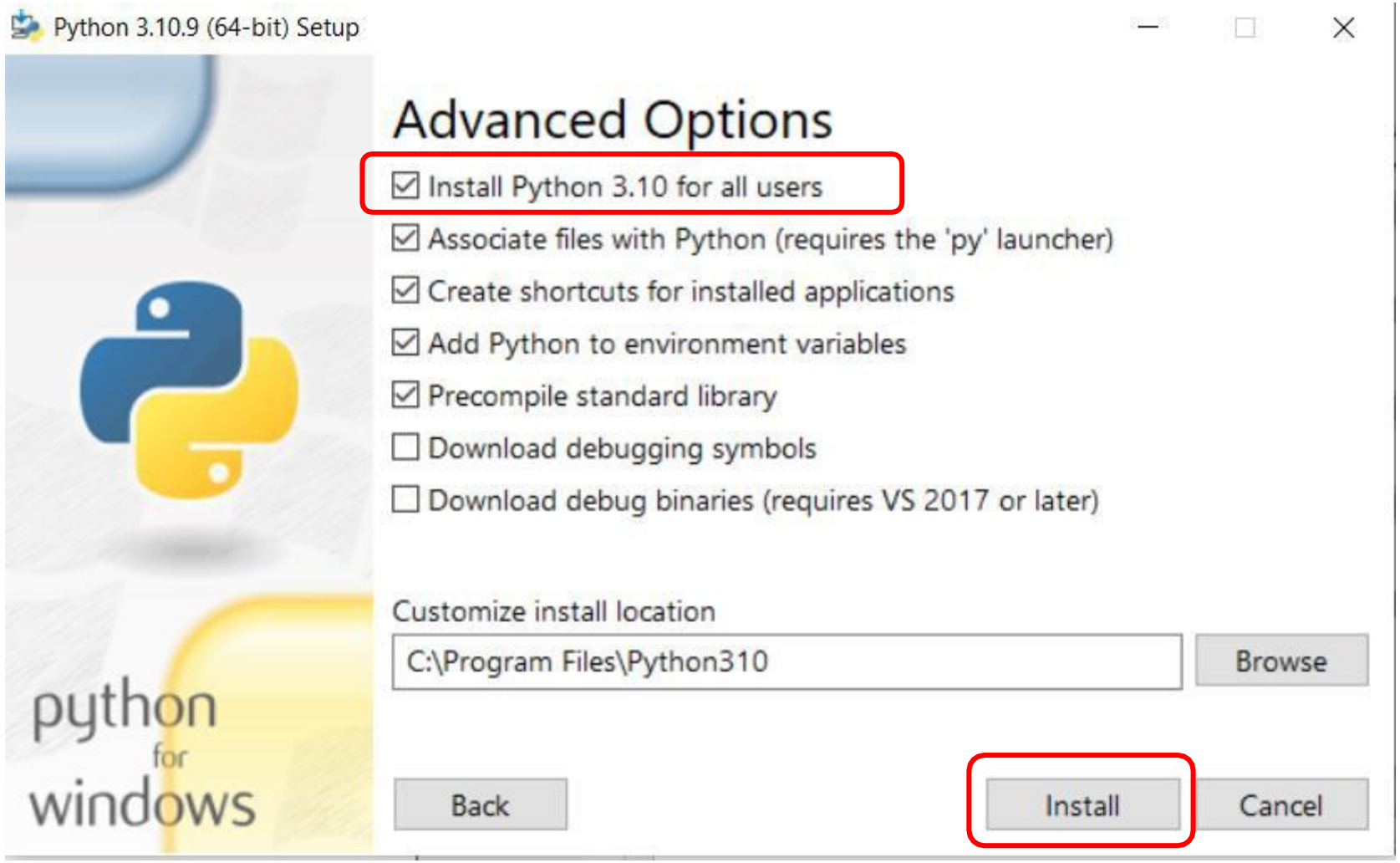


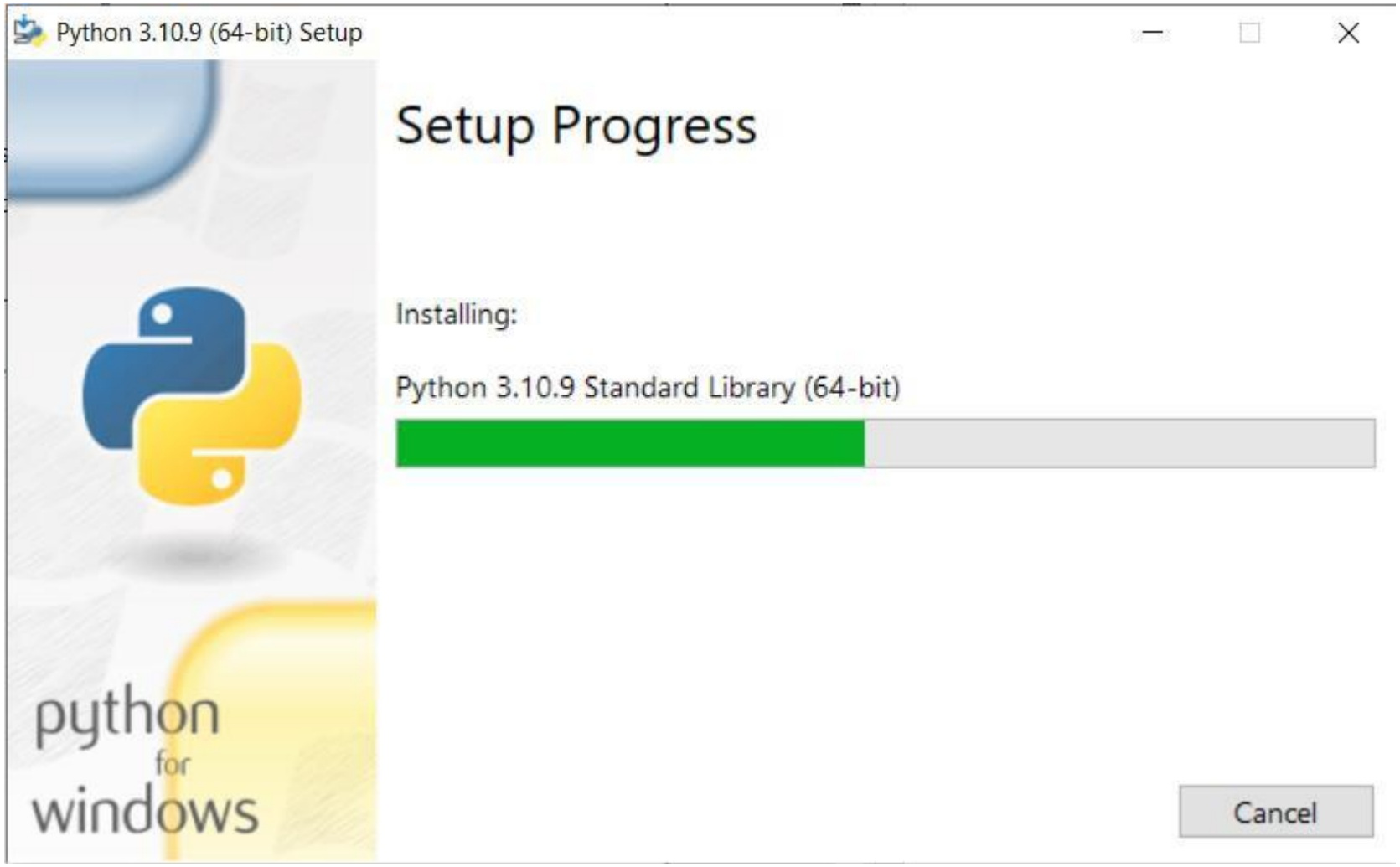
Cocher « Add Python to PATH »
Et cliquer sur « Customize installation »

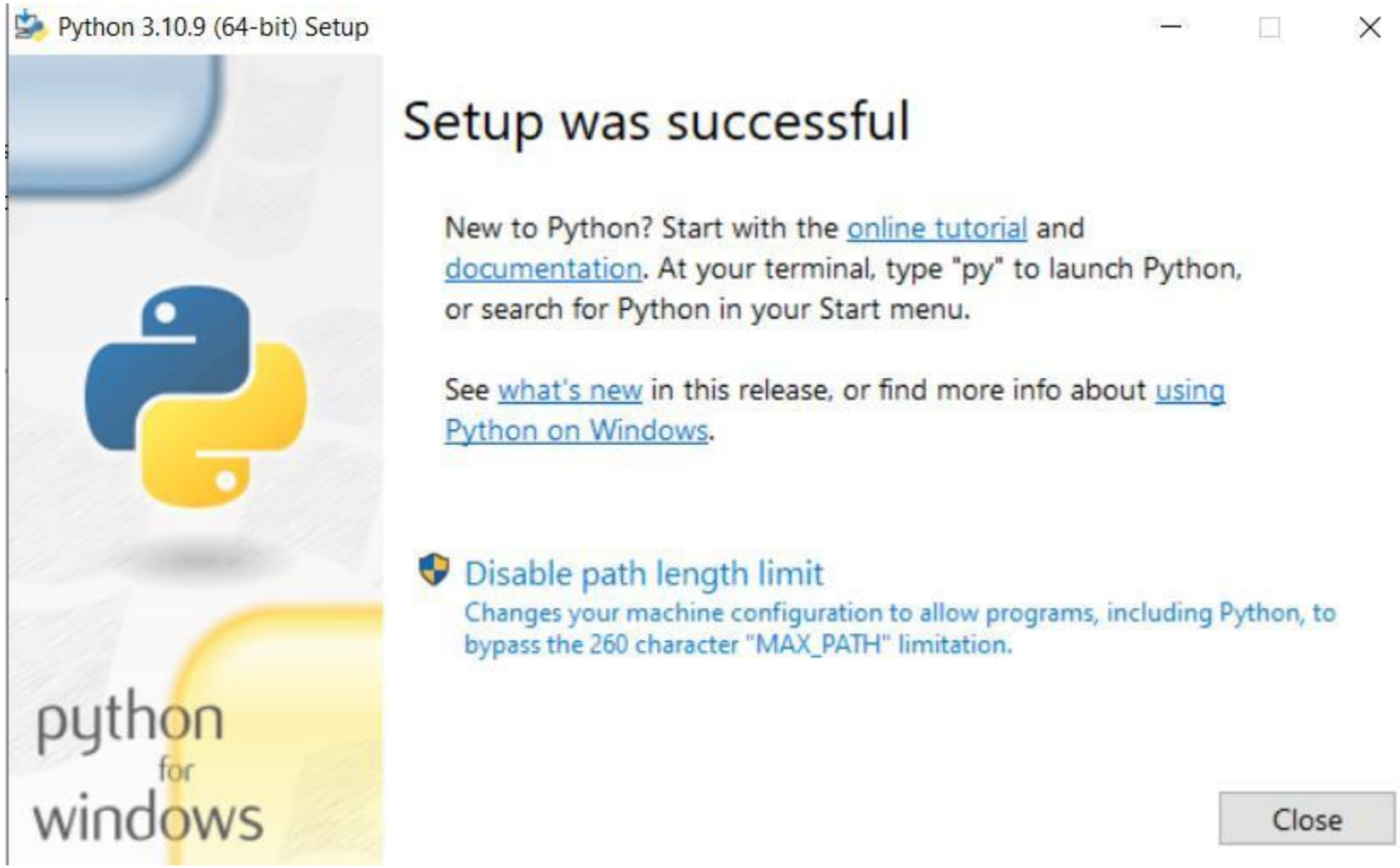


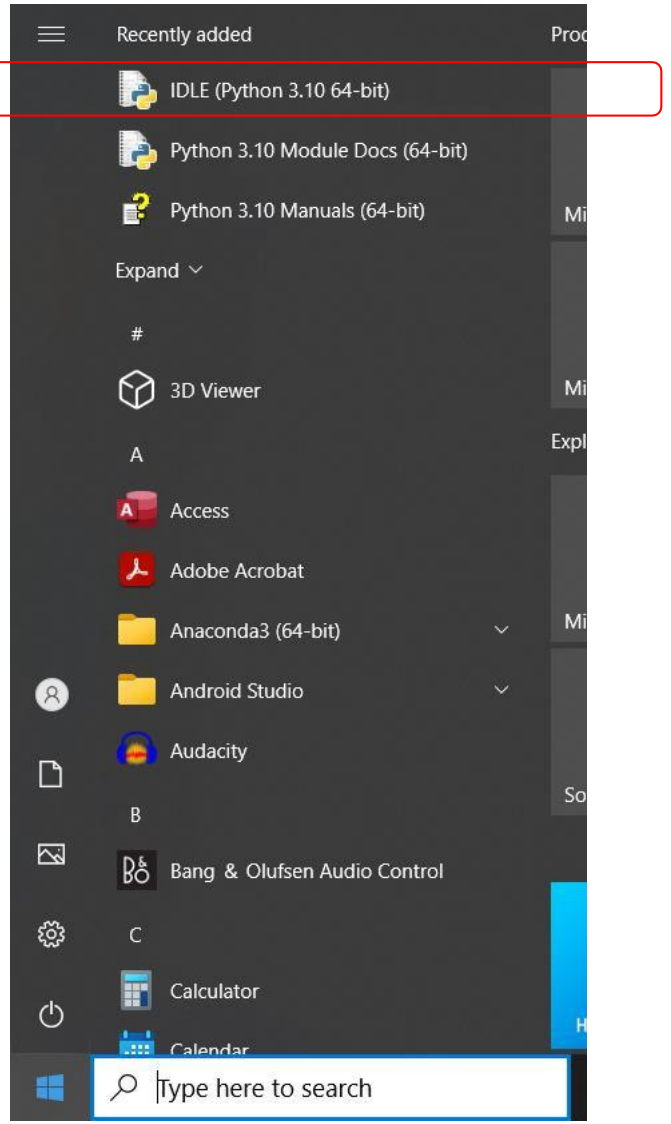
Télécharger et installer Python.exe

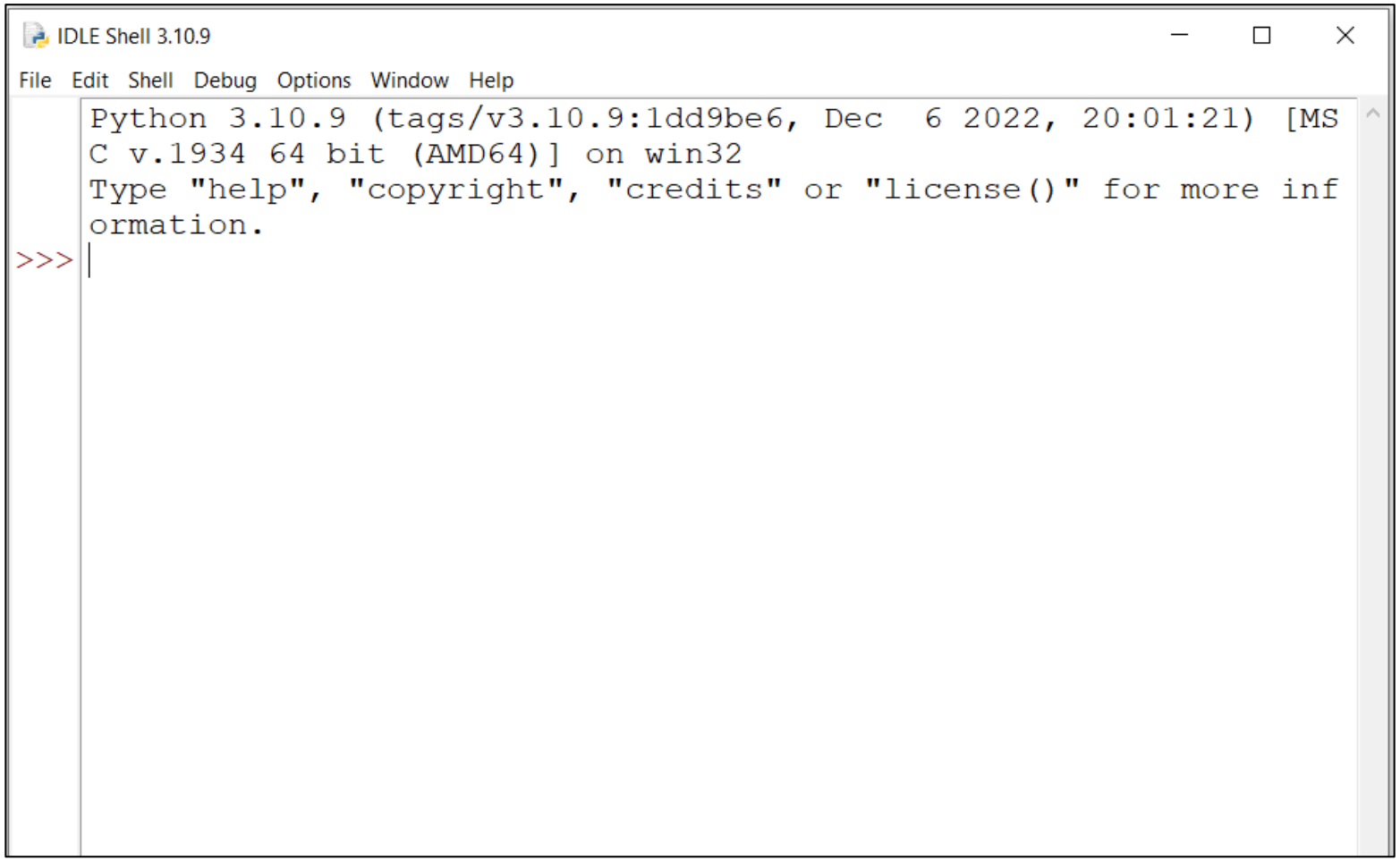


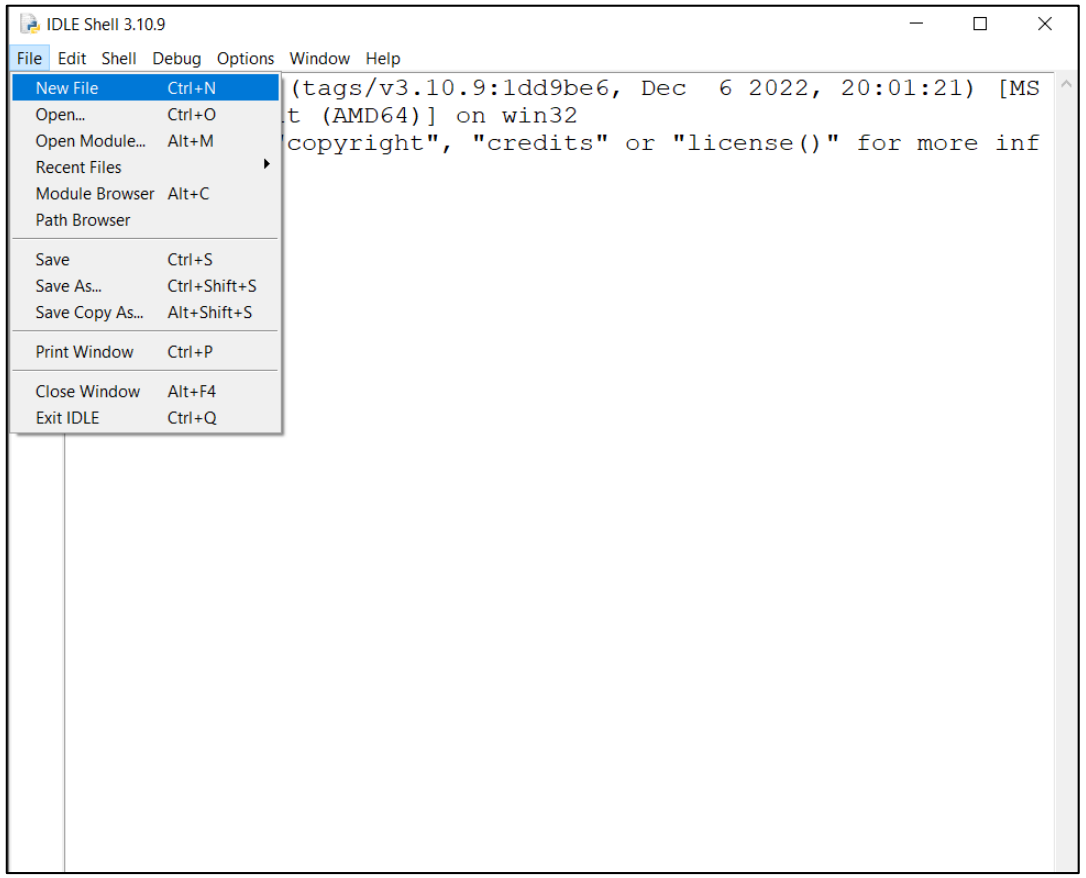

Python 3.10.9 (64-bit) Setup
Advanced Options
Install Python 3.10 for all users
Associate files with Python (requires the 'py' launcher)
Create shortcuts for installed applications
Add Python to environment variables
Precompile standard library
Download debugging symbols
Download debug binaries (requires VS 2017 or later)
Customize install location
C:\Program Files\Python310
Browse
python
for
windows
Back
Install
Cancel

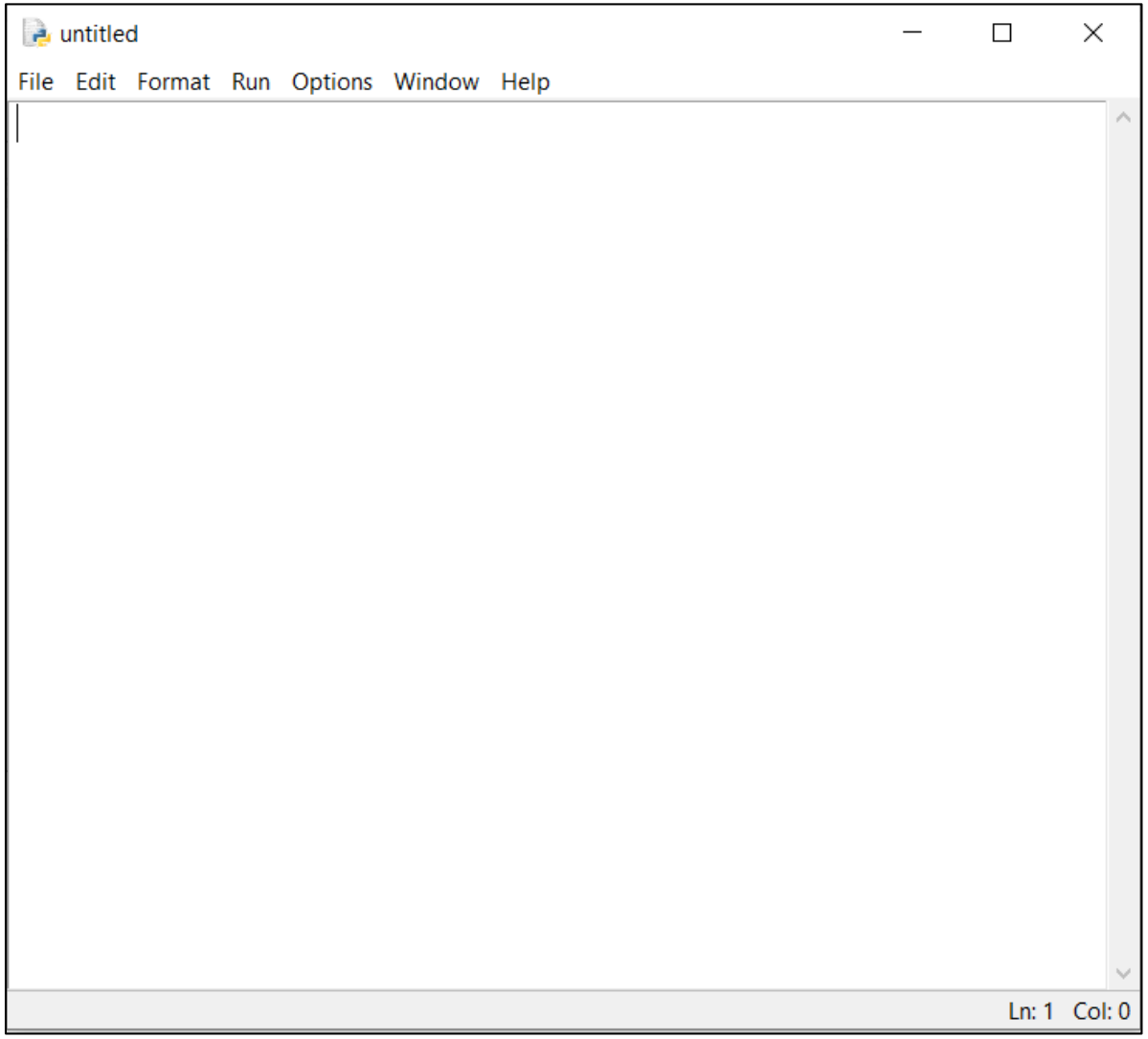

Python 3.10.9 (64-bit) Setup
Setup Progress
Installing:
Python 3.10.9 Standard Library (64-bit)
python
for
windows
Cancel


Python 3.10.9 (64-bit) Setup
Setup was successful
New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.
See what's new in this release, or find more info about using Python on Windows.
Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.
python
for
windows
Close

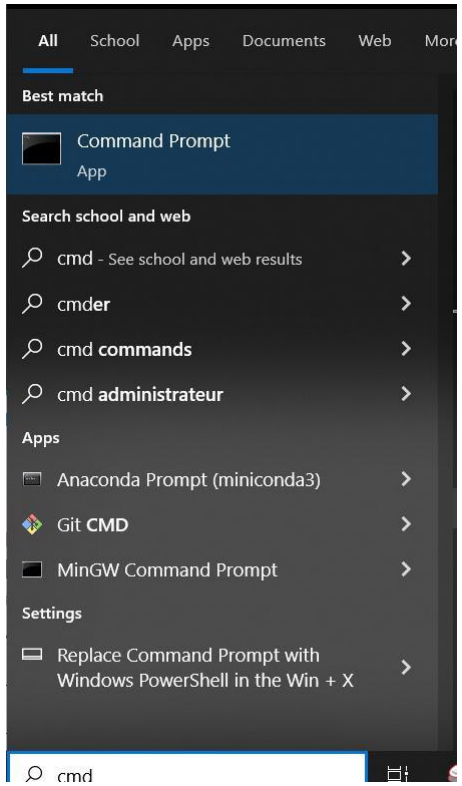

Recently added
IDLE (Python 3.10 64-bit)
Python 3.10 Module Docs (64-bit)
Python 3.10 Manuals (64-bit)
Expand
#
3D Viewer
A
Access
Adobe Acrobat
Anaconda3 (64-bit)
Android Studio
Audacity
B
Bang & Olufsen Audio Control
C
Calculator
Type here to search

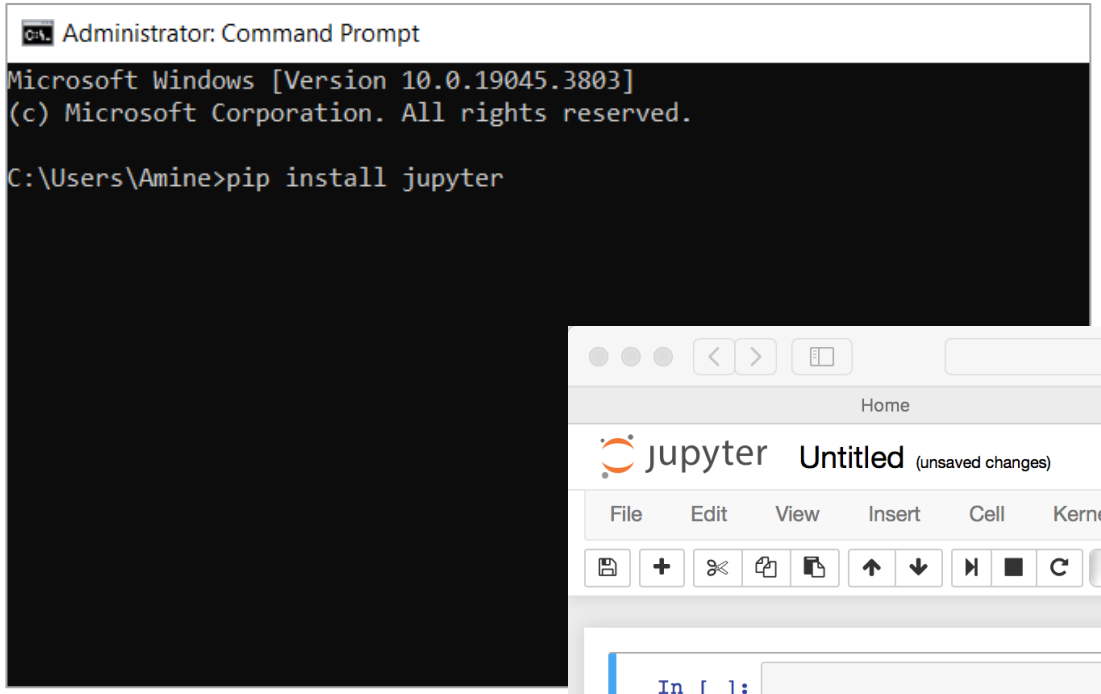

IDLE Shell 3.10.9
File Edit Shell Debug Options Window Help
Python 3.10.9 (tags/v3.10.9:1dd9be6, Dec 6 2022, 20:01:21) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

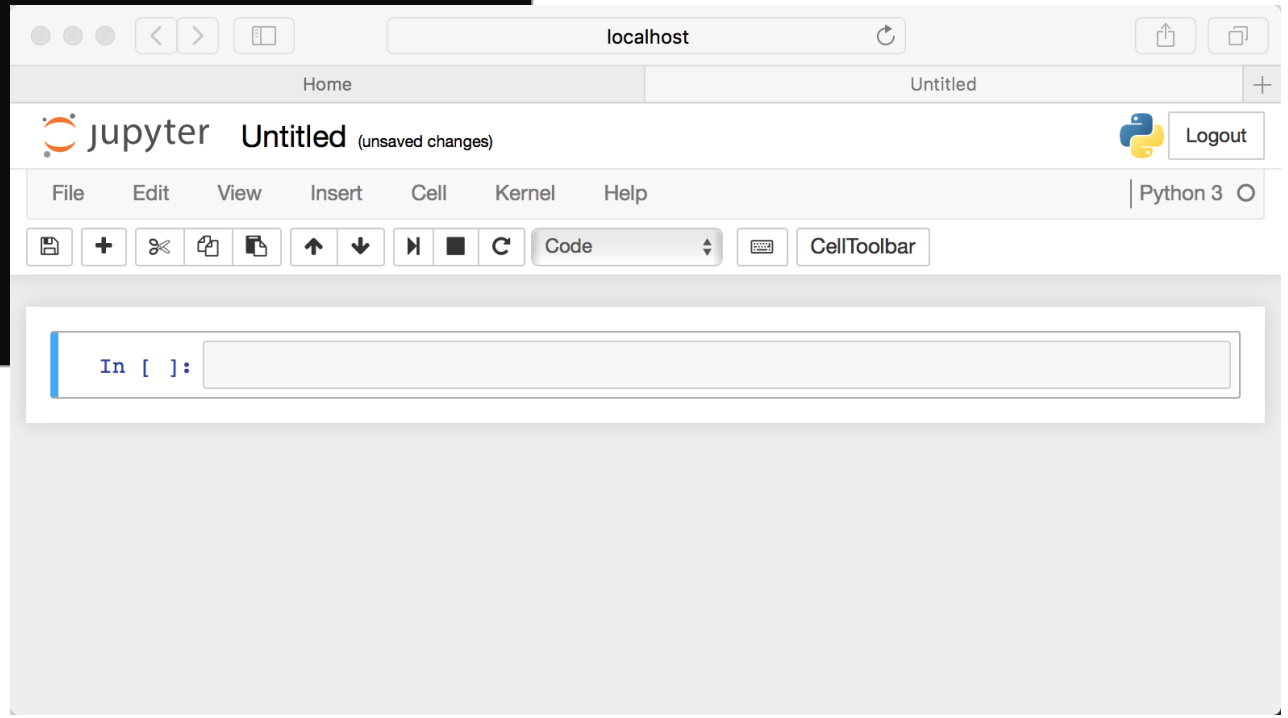

IDLE Shell 3.10.9
File Edit Shell Debug Options Window Help
New File Ctrl+N
Open... Ctrl+O
Open Module... Alt+M
Recent Files
Module Browser Alt+C
Path Browser
Save Ctrl+S
Save As... Ctrl+Shift+S
Save Copy As... Alt+Shift+S
Print Window Ctrl+P
Close Window Alt+F4
Exit IDLE Ctrl+Q
(tags/v3.10.9:1dd9be6, Dec 6 2022, 20:01:21) [MS
t (AMD64)] on win32
copyright", "credits" or "license()" for more inf


untitled
File Edit Format Run Options Window Help
Ln: 1 Col: 0

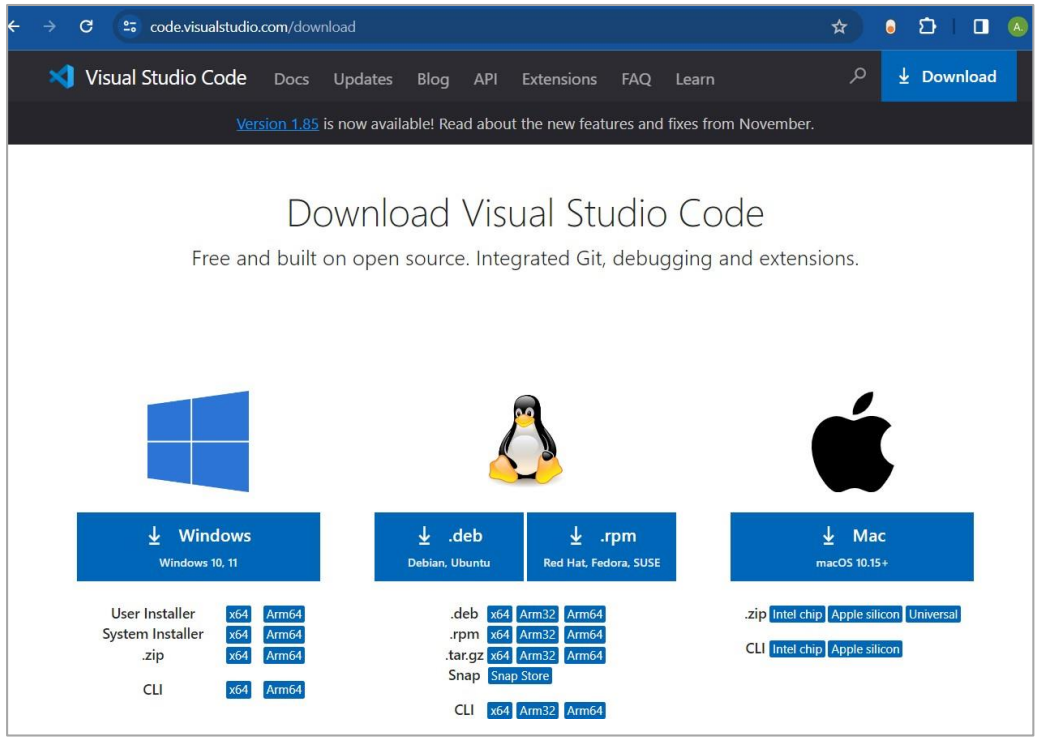
Autres IDE modernes
Jupyter notebook

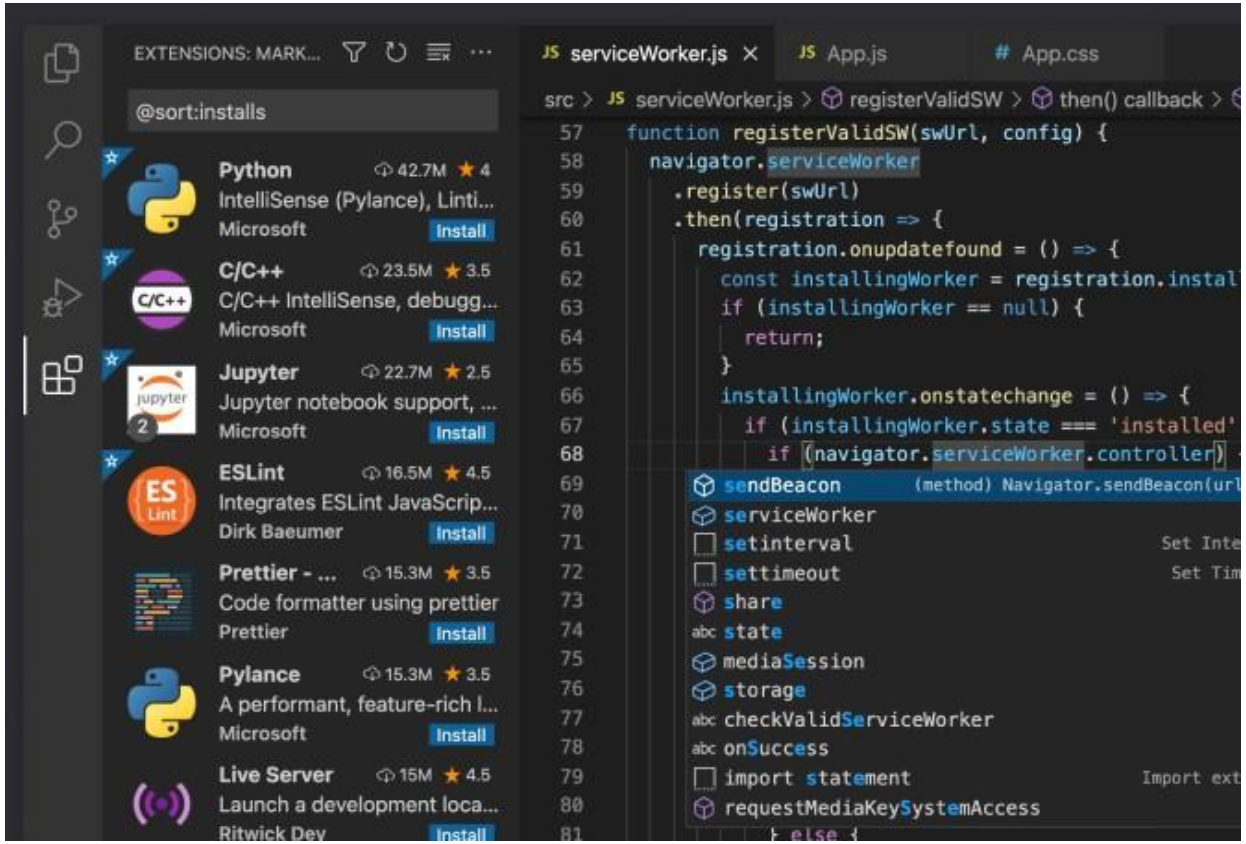






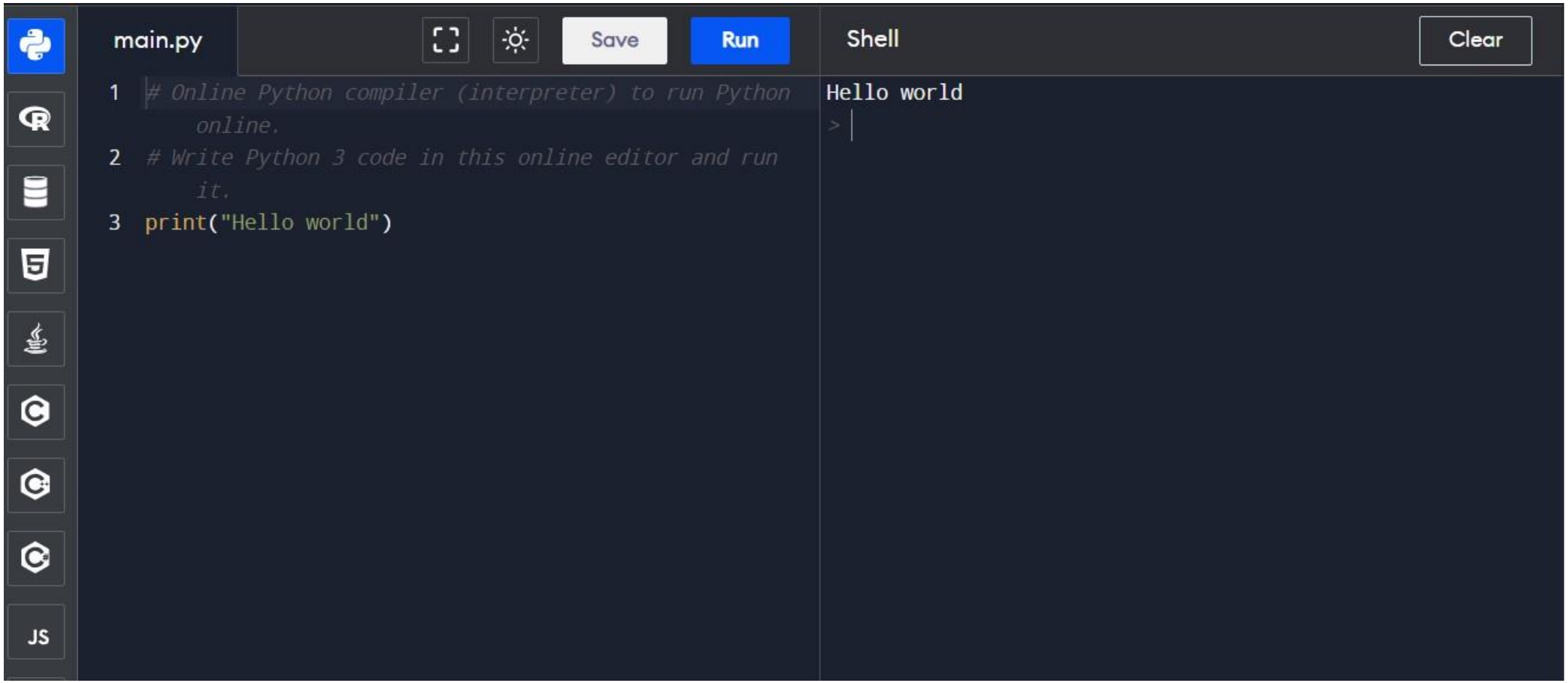
Autres IDE modernes
Visual studio Code





Sinon, IDE en ligne

Pas besoin d'installation

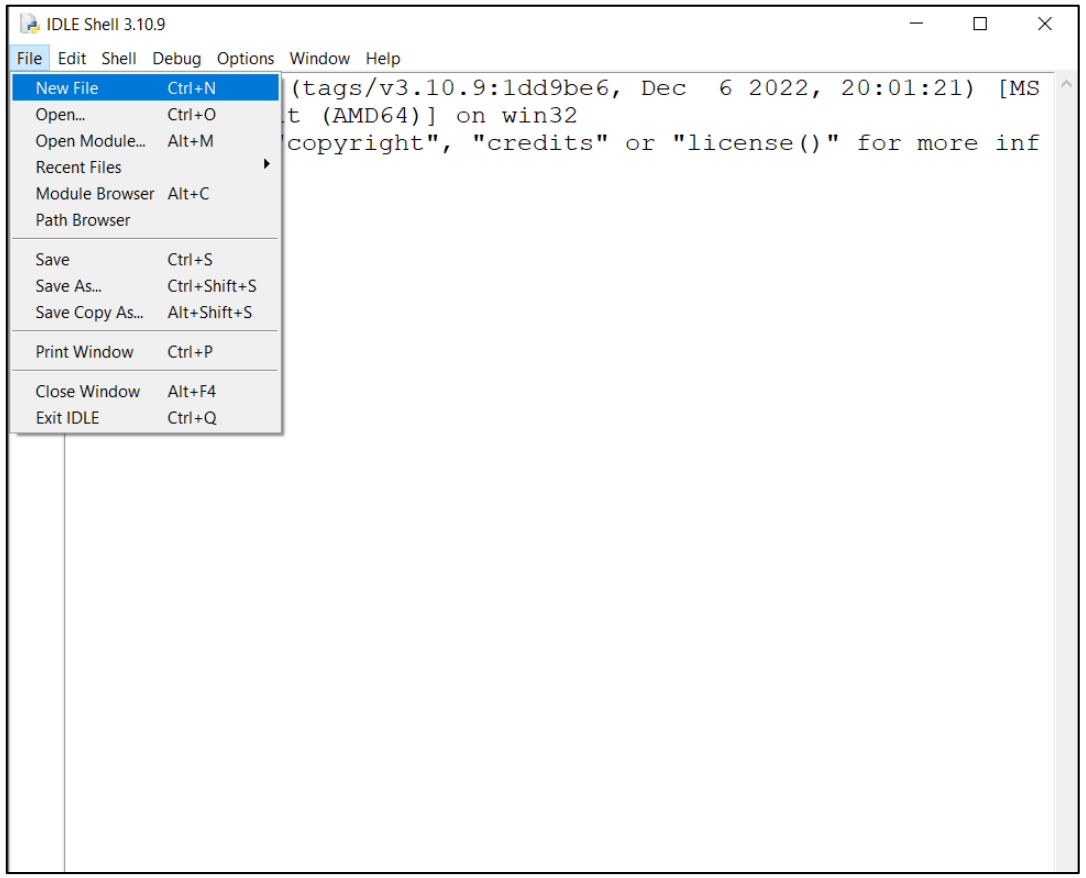


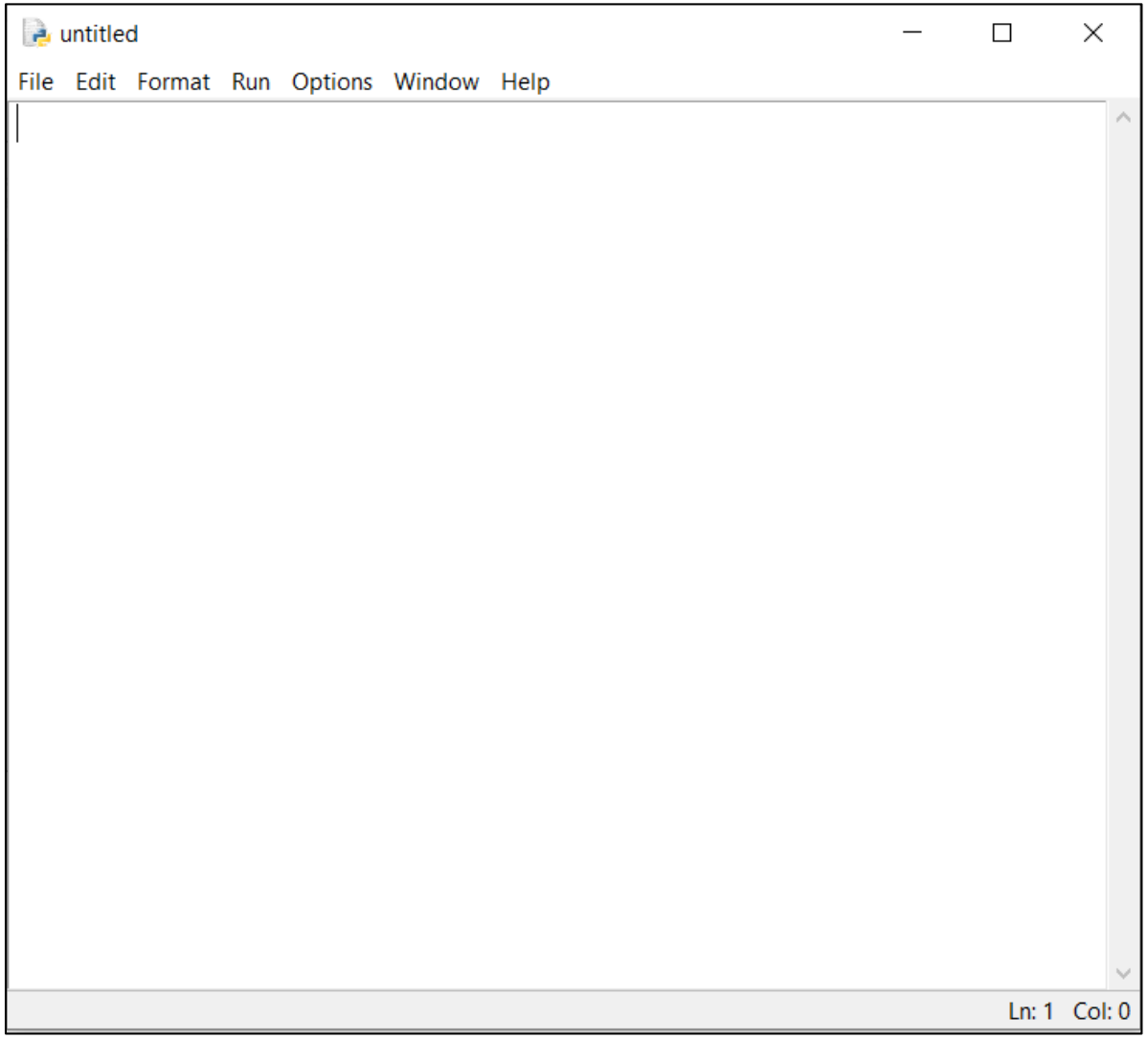
Programiz

https://www.programiz.com/python-programming/online-compiler/

Votre premier programme Python

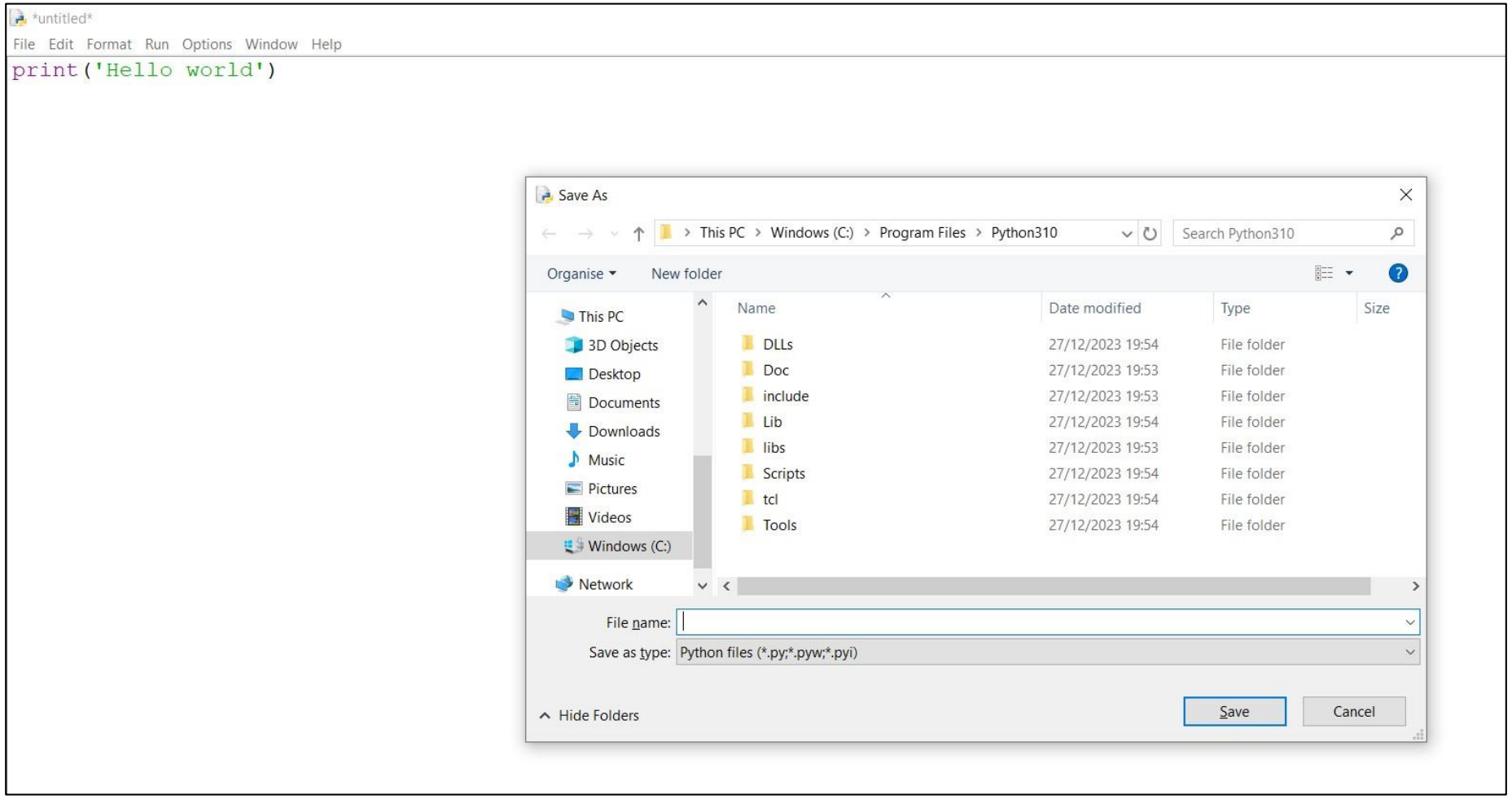
```
print('Bonjour')
```

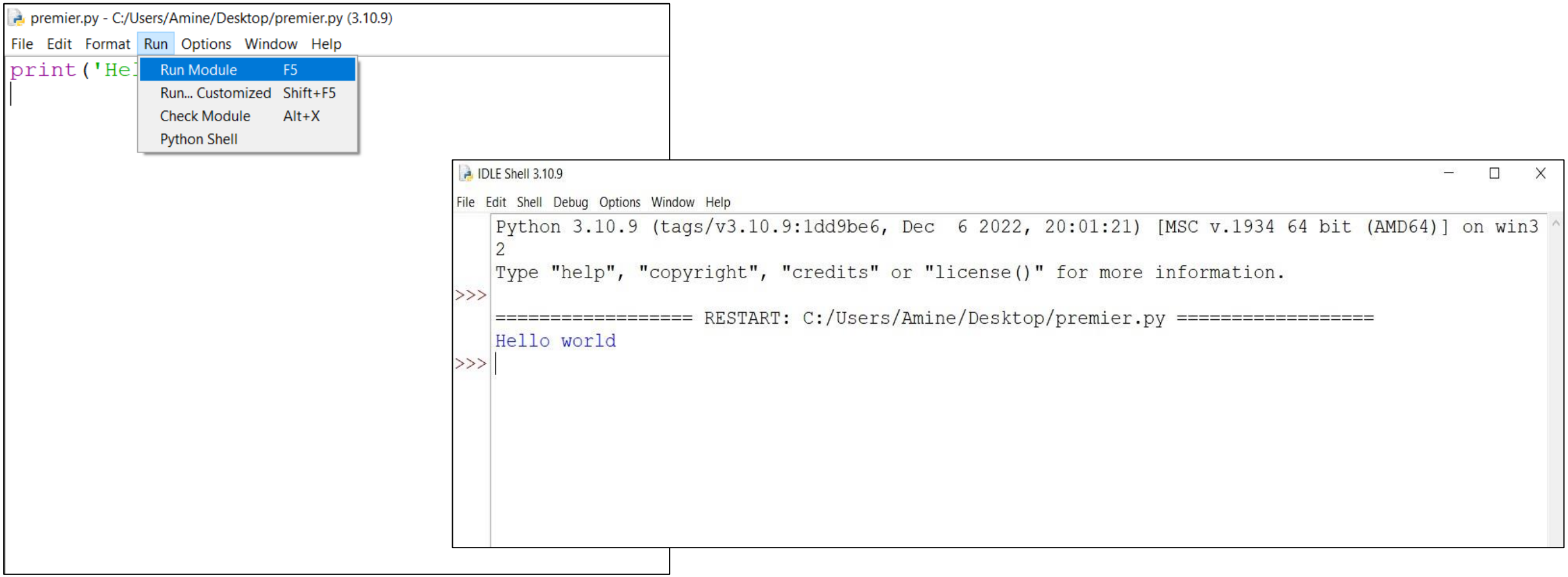

IDLE Shell 3.10.9
File Edit Shell Debug Options Window Help
New File Ctrl+N
Open... Ctrl+O
Open Module... Alt+M
Recent Files
Module Browser Alt+C
Path Browser
Save Ctrl+S
Save As... Ctrl+Shift+S
Save Copy As... Alt+Shift+S
Print Window Ctrl+P
Close Window Alt+F4
Exit IDLE Ctrl+Q
(tags/v3.10.9:1dd9be6, Dec 6 2022, 20:01:21) [MS
t (AMD64)] on win32
copyright", "credits" or "license()" for more inf


untitled
File Edit Format Run Options Window Help
Ln: 1 Col: 0

print(‘Hello word’)

Sauvegarder le fichier et exécuter pour voir le résultat




premier.py - C:/Users/Amine/Desktop/premier.py (3.10.9)
File Edit Format Run Options Window Help
print('He
Run Module F5
Run... Customized Shift+F5
Check Module Alt+X
Python Shell
IDLE Shell 3.10.9
File Edit Shell Debug Options Window Help
Python 3.10.9 (tags/v3.10.9:1dd9be6, Dec 6 2022, 20:01:21) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/Amine/Desktop/premier.py ==========
Hello world
>>>

Les Variables en Python

- Les variables sont des conteneurs permettant de stocker des valeurs de données.

- Python n'a pas de commande pour déclarer une variable.
- Une variable est créée au moment où vous lui attribuez une valeur pour la première fois.

```
x = 5
y = “Ali"
print(x)
print(y)
```

- Les variables n'ont pas besoin d'être déclarées avec un type particulier et peuvent même changer de type après avoir été définies.

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

- Si vous souhaitez spécifier le type de données d'une variable, cela peut être fait à l'aide d'un casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

- Vous pouvez obtenir le type de données d'une variable avec la fonction type().

```
x = 5
y = “Ali"
print(type(x))
print(type(y))
```

- Les variables de chaîne peuvent être déclarées en utilisant des guillemets simples ou doubles :

```
x = “Ali"
# is the same as
x = ‘Ali'
```

- Sensible aux majuscules et minuscules: Les noms de variables sont sensibles à la casse.

```
a = 4
A = “Abdelali"
#A will not overwrite a
```

- Noms légaux des variables :

```
myvar = "Abdelali"
my_var = "Abdelali"
_my_var = "Abdelali"
myVar = "Abdelali"
MYVAR = "Abdelali"
myvar2 = "Abdelali"
```

- Noms de variables illégaux :

```
2myvar = "Abdelali"
my-var = "Abdelali"
my var = "Abdelali"
```

- Python vous permet d'attribuer des valeurs à plusieurs variables sur une seule ligne :

```
x, y, z = "Ali", "Ahmed", "Sara"
print(x)
print(y)
print(z)
```

- Et vous pouvez attribuer la *même* valeur à plusieurs variables sur une seule ligne :

```
x = y = z = "Ali"
print(x)
print(y)
print(z)
```

- Si vous avez une collection de valeurs dans une liste, un tuple, etc. Python vous permet d'extraire les valeurs dans des variables. C'est ce qu'on appelle le déballage .

```
Names = ["Ali", "Ahmed", "Sara"]
x, y, z = Names
print(x)
print(y)
print(z)
```

Variables de sortie

- La fonction print() en Python est souvent utilisée pour générer des variables.

```
x = " My name is Ali"
print(x)
```

- Dans la fonction print(), vous générez plusieurs variables, séparées par une virgule :

- Vous pouvez également utiliser l' **+** opérateur pour générer plusieurs variables :

```
x = " My name "
y = "is "
z = “Ali"
print(x + y + z)
```

```
x = “My name"
y = "is"
z = “Ali"
print(x, y, z)
```

- La somme des variables :

```
x = 5
y = 10
print(x + y)
```

- Dans la fonction **print()**, lorsque vous essayez de combiner une chaîne et un nombre avec l' **+** opérateur, Python vous donnera une erreur :

```
x = 5
y = “Ali"
print(x + y)
```

- La meilleure façon de générer plusieurs variables dans la fonction print() est de les séparer par des virgules, qui prennent même en charge différents types de données :

```
x = 5
y = “Ali"
print(x, y)
```

Types de données intégrés

En programmation, le type de données est un concept important.

Les variables peuvent stocker des données de différents types, et différents types peuvent faire différentes choses.

Python possède les types de données suivants intégrés par défaut, dans ces catégories :

Type de texte :	**str**
Types numériques :	**int, float, complex**
Types de séquences :	**list, tuple, range**
Type de cartographie :	**dict**
Types d'ensembles :	**set, frozenset**
Type booléen :	**bool**
Types binaires :	**bytes, bytearray, memoryview**
Aucun Type:	**NoneType**

- ***Obtenir le type de données***

Vous pouvez obtenir le type de données de n'importe quel objet en utilisant la fonction **type()** :
Python possède les types de données suivants intégrés par défaut, dans ces catégories :

```
x = 5
print(type(x))
```

```
x = 5    # int

print(type(x))
print(type(y))
print(type(z))
```

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

- ***Nombre aléatoire***

Python a un module intégré appelé **random** qui peut être utilisé pour créer des nombres aléatoires :

```
import random

print(random.randrange(1, 10))
```

```
x = int(1)   # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3


x = str("s1") # x will be 's1'
y = str(2)    # y will be '2'
z = str(3.0)  # z will be '3.0'
```

```
x = float(1)     # x will be 1.0
y = float(2.8)   # y will be 2.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
```

- ***Obtenir le type de données***

Les chaînes en Python sont entourées soit de guillemets simples, soit de guillemets doubles.
'bonjour' est la même chose que **"bonjour"** .

Vous pouvez utiliser trois guillemets doubles :

```
x = 5
a = """Abdelali El Gourari,
en informatique spécialisé en intelligence
artificielle et systèmes embarqués,
Ses principaux intérêts de recherche"""
print(a)
```

```
a = ''Abdelali El Gourari,
en informatique spécialisé en intelligence
artificielle et systèmes embarqués,
Ses principaux intérêts de recherche'''
print(a)
```

Récupérez le caractère à la position 1 (rappelez-vous que le premier caractère a la position 0) :

```
a = "Hello, World!"
print(a[1])
```

La len() fonction renvoie la longueur d'une chaîne :

```
a = "Hello, World!"
print(len(a))
```

- ***Vérifier la chaîne***

Vérifiez si « **free** » est présent dans le texte suivant :

```
txt = "The best things in life are free!"
print("free" in txt)
```

Imprimer uniquement si « **free** » est présent :
Vous pouvez utiliser trois guillemets doubles :

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Vérifiez si « **cher** » n'est pas présent dans le texte suivant :

```
txt = "The best things in life are free!"
print(" cher " not in txt)
```

Imprimer uniquement si « **cher** » n'est pas présent :

```
txt = "The best things in life are free!"
if " cher " not in txt:
  print("No, ' cher ' is NOT present.")
```

- ***Découpage de chaînes***

Obtenez les personnages de la position 2 à la position 5:

```
b = "Hello, World!"
print(b[2:5])
```

Obtenez les personnages du début à la position 5:

```
b = "Hello, World!"
print(b[:5])
```

Récupérez les personnages de la position 2 jusqu'à la fin :

```
b = "Hello, World!"
print(b[2:])
```

- ***Indexation négative***

Utilisez des index négatifs pour démarrer la tranche à partir de la fin de la chaîne :

Obtenez les personnages :
De : « o » dans «World ! » (position -5)
À, mais non inclus : « d » dans «World ! » (position -2) :

```
b = "Hello, World!"
print(b[-5:-2])
```

- ***Modifier les chaînes***

La méthode upper() renvoie la chaîne en majuscules :

```
a = "Hello, World!"
print(a.upper())
```

La méthode lower() renvoie la chaîne en minuscules :

```
a = "Hello, World!"
print(a.lower())
```

La méthode strip() supprime tout espace blanc du **début** ou de la **fin** :

```
a = " Hello, World! "
print(a.strip()) #### "Hello, World!"
```

La méthode replace() remplace une chaîne par une autre chaîne :

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

La méthode split() divise la chaîne en sous-chaînes si elle trouve des instances du séparateur :

```
a = "Hello, World!"
print(a.split(",")) #### ['Hello', ' World!']
```

- ***Concaténation de chaînes***

Pour concaténer ou combiner deux chaînes, vous pouvez utiliser l'opérateur **+**.

Fusionner variable aavec variable bdans variable c:

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

Pour ajouter un espace entre eux, ajoutez un " ":

```
a = "Hello"
b = "World"
c = a + " " + b
print(c)
```

- ***Format de chaîne***

Comme nous l'avons appris dans le chapitre Variables Python, nous ne pouvons pas combiner des chaînes et des nombres comme ceci :

!!!!!

```
age = 28
txt = "My name is Ali, I am " + age
print(txt)
```

Mais nous pouvons combiner des chaînes et des nombres en utilisant des chaînes f ou la méthode format() !

F-String a été introduit dans Python 3.6 et constitue désormais le moyen privilégié de formater les chaînes.

Pour spécifier une chaîne comme une chaîne **f**, placez simplement un **f** devant le littéral de chaîne et ajoutez des accolades **{}** comme espaces réservés pour les variables et autres opérations.

Créer une chaîne f :

```
age = 28
txt = f"My name is Ali, I am {age}"
print(txt)
```

```
price = 100
txt = f"The price is {price} DH"
print(txt)
```

Un modificateur est inclus en ajoutant deux points :suivis d'un type de formatage légal, comme **.2f** ce qui signifie un nombre à virgule fixe avec **2** décimales :

Afficher le prix avec 2 décimales :

```
price = 100
txt = f"The price is {price:.2f} DH"
print(txt)
```

Effectuez une opération mathématique dans l'espace réservé et renvoyez le résultat :

```
txt = f"The price is {16 * 100} DH"
print(txt)
```

Python prend en charge les conditions logiques habituelles des mathématiques :

- ✓ Égal à : `a == b`
- ✓ Pas égal à : `a != b`
- ✓ Inférieur à : `a < b`
- ✓ Inférieur ou égal à : `a <= b`
- ✓ Supérieur à : `a > b`
- ✓ Supérieur ou égal à : `a >= b`

Ces conditions peuvent être utilisées de plusieurs manières, le plus souvent dans des instructions **if** et des **boucles**.

```
a = 10
b = 100
if b > a:
  print("b est supérieur a")
```

Instruction **If**, sans indentation (générera une erreur) :

```
a = 10
b = 100
if b > a:
print(" b est supérieur a ") # Error
```

Le mot-clé `elif` est la manière dont Python dit « si les conditions précédentes n'étaient pas vraies, alors essayez cette condition ».

```
a = 10
b = 10
if b > a:
  print(" b est supérieur a ")
elif a == b:
  print("a et b sont égaux")
```

```
a = 100
b = 10
if b > a:
  print(" b est supérieur a ")
elif a == b:
  print(" a et b sont égaux ")
else:
  print(" a est supérieur b ")
```

Instruction if sur une ligne :

```
if a > b: print(" a est supérieur b ")
```

Instruction **if** sur une ligne :

```
a = 10
b = 100
print(“Ali") if a > b else print(“Ahmed")
```

Instruction **if** **et** **else** sur une ligne, avec 3 conditions :

```
a = 10
b = 100
print("Abdelali") if a > b else print(“Ahmed") if a == b else print(“Sara")
```

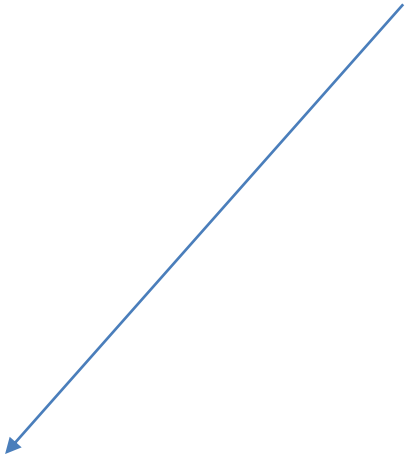
Le mot clé and est un opérateur logique et est utilisé pour combiner des instructions conditionnelles :

Testez si a est supérieur à b, **ET** si c est supérieur à a:

```
a = 100
b = 10
c = 200
if a > b and c > a:
  print("les deux conditions sont vraies")
```

Le mot clé **or** est un opérateur logique et est utilisé pour combiner des instructions conditionnelles :

Testez si a est supérieur à b, **OU** si a est supérieur à c:



```
a = 100
b = 10
c = 200
if a > b or a > c:
  print("au moins une des conditions est vraie")
```

Le mot clé not est un opérateur logique et est utilisé pour inverser le résultat de l'instruction conditionnelle :

Tester si a n'est **PAS** supérieur à b:

```
a = 10
b = 100
if not a > b:
  print("a n'est pas supérieur à b ")
```

Vous pouvez avoir des instructions if à l'intérieur de if d'instructions, c'est ce qu'on appelle des instructions imbriquées if .

```
x = 30

if x > 10:
  print("Au-dessus de dix")
  if x > 20:
    print("et aussi au-dessus de 20")
  else:
    print("mais pas au dessus de 20")
```

Les instructions **If** ne peuvent pas être vides, mais si pour une raison quelconque vous avez une instruction **if** sans contenu, insérez l'instruction **pass** pour éviter d'obtenir une erreur.

```
a = 10
b = 100

if b > a:
  pass
```

Les instructions If ne peuvent pas être vides, mais si pour une raison quelconque vous avez une instruction if sans contenu, insérez l'instruction pass pour éviter d'obtenir une erreur.

Une boucle for est utilisée pour parcourir une séquence (qui est soit une liste, un tuple, un dictionnaire, un ensemble ou une chaîne).
Cela ressemble moins au mot-clé for dans d'autres langages de programmation et fonctionne davantage comme une méthode d'itérateur que l'on trouve dans d'autres langages de programmation orientés objet.
Avec la boucle for, nous pouvons exécuter un ensemble d'instructions, une fois pour chaque élément d'une liste, d'un tuple, d'un ensemble, etc.

Imprimez chaque nom dans une liste de fruits :

```
names = [“abdelali", “ahmed", “sara"]
for x in names:
  print(x)
```

Parcourez les lettres du mot « **abdelali** » :

```
for x in “abdelali":
  print(x)
```

Sortir de la boucle quand a c'est « abdelali" :

```
names = [“ahmed", “abdelali", “sara"]
for x in names:
  print(x)
  if x == “abdelali":
    break
```

Sortez de la boucle lorsque x c'est "**abdelali**", mais cette fois la pause vient avant l'impression :

```
names = ["ahmed", “abdelali", “sara"]
for x in names:
  if x == “abdelali":
    break
  print(x)
```

Avec l' instruction continue , nous pouvons arrêter l'itération actuelle de la boucle et continuer avec la suivante :

N'imprimez pas **abdelali**:

```
names = [“ahmed", “abdelali", “sara"]
for x in names:
  print(x)
  if x == “abdelali":
    continue
```

```
names = ["ahmed", “abdelali", “sara"]
for x in names:
  if x == “abdelali":
    break
  print(x)
```

Les Boucles en Python: La fonction range()

Si vous devez itérer sur une suite de nombres, la fonction native range() est faite pour cela. Elle génère des suites arithmétiques :

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

0
1
2
3
4
```

On appelle de tels objets des **itérables**, c'est-à-dire des objets qui conviennent à des fonctions ou constructions qui s'attendent à quelque chose duquel elles peuvent tirer des éléments, successivement, jusqu'à épuisement. Nous avons vu que l'instruction **for** est une de ces constructions, et un exemple de fonction qui prend un itérable en paramètre est **sum()** :

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Le dernier élément fourni en paramètre ne fait jamais partie de la liste générée ; **range(10)** génère une liste de 10 valeurs, dont les valeurs vont de 0 à 9. Il est possible de spécifier une valeur de début et une valeur d'incrément différentes (y compris négative pour cette dernière, que l'on appelle également parfois le « pas ») :

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Pour itérer sur les indices d'une séquence, on peut combiner les fonctions **range()** et **len()** :

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

- ***Boucles imbriquées***

Une boucle imbriquée est une boucle à l'intérieur d'une boucle.

La « boucle interne » sera exécutée une fois pour chaque itération de la « boucle externe » :

```
names = [“Abdelali", “Ahmed", “Jalal"]
description = [“is small", “is happy", “is smart"]

for x in names:
  for y in description:
    print(x, y)
```

les boucles **for** ne peuvent pas être vides, mais si pour une raison quelconque vous avez une boucle **for** sans contenu, insérez l'instruction **pass** pour éviter d'obtenir une erreur.

```
for x in [0, 1, 2]:
  pass
```

```
while True:
    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```

- ***La boucle while***

Avec la boucle while , nous pouvons exécuter un ensemble d'instructions tant qu'une condition est vraie.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

La boucle `while` nécessite que les variables pertinentes soient prêtes, dans cet exemple, nous devons définir une variable d'indexation, `i` , que nous définissons à 1.

Avec l' instruction break , nous pouvons arrêter la boucle même si la condition while est vraie :

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

Avec l' instruction `continue` , nous pouvons arrêter l'itération en cours et continuer avec la suivante :

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

- ***La boucle while***

Avec l'instruction `else` , nous pouvons exécuter un bloc de code une fois lorsque la condition n'est plus vraie :

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

En Python, une fonction est définie à l'aide du mot-clé **def** :

```
def my_function():
  print(“My name is Abdelali")
```

Pour **appeler** une fonction, utilisez le nom de la fonction suivi de parenthèses :

```
def my_function():
  print(" My name is Abdelali ")

my_function()
```

Arguments:

Les informations peuvent être transmises aux fonctions sous forme d'arguments.

Les arguments sont spécifiés après le nom de la fonction, entre parenthèses. Vous pouvez ajouter autant d'arguments que vous le souhaitez, séparez-les simplement par une virgule.

L'exemple suivant contient une fonction avec un argument (**text**). Lorsque la fonction est appelée, nous transmettons un prénom, qui est utilisé à l'intérieur de la fonction pour afficher le nom complet :

```
def my_function(text):
  print(fname + “Power")

my_function(“Spot is")
my_function(“Sleep gives you")
my_function(“Studying needs")
```

Nombre d'arguments

Par défaut, une fonction doit être appelée avec le nombre correct d'arguments. Cela signifie que si votre fonction attend 2 arguments, vous devez appeler la fonction avec 2 arguments, ni plus, ni moins.

Cette fonction attend **2** arguments et obtient **2** arguments :

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

Si vous essayez d'appeler la fonction avec 1 ou 3 arguments, vous obtiendrez une erreur :

Cette fonction attend 2 arguments, mais n'en obtient qu'un :

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

*Arguments arbitraires, ***args***

Si vous ne savez pas combien d'arguments seront passés dans votre fonction, ajoutez un ***** avant le nom du paramètre dans la définition de la fonction.

De cette façon, la fonction recevra un **tuple** d'arguments et pourra accéder aux éléments en conséquence :