Langages informatiques - Python

Université Panthéon-Sorbonne – EMS

Enseignant : Matthieu Jacq - matthieu.jacq@univ-paris1.fr

Objectif de la séance

1. Prendre en main l'éditeur de code VSCode.
2. Bien comprendre le fonctionnement d'un notebook Jupyter.
3. Voir les bases de la programmation en python :
   - Les commentaires
   - Les variables
   - Les fonctions
   - Quelques opérateurs de calcul
   - Les structures de contrôle :
     - conditions
     - boucles

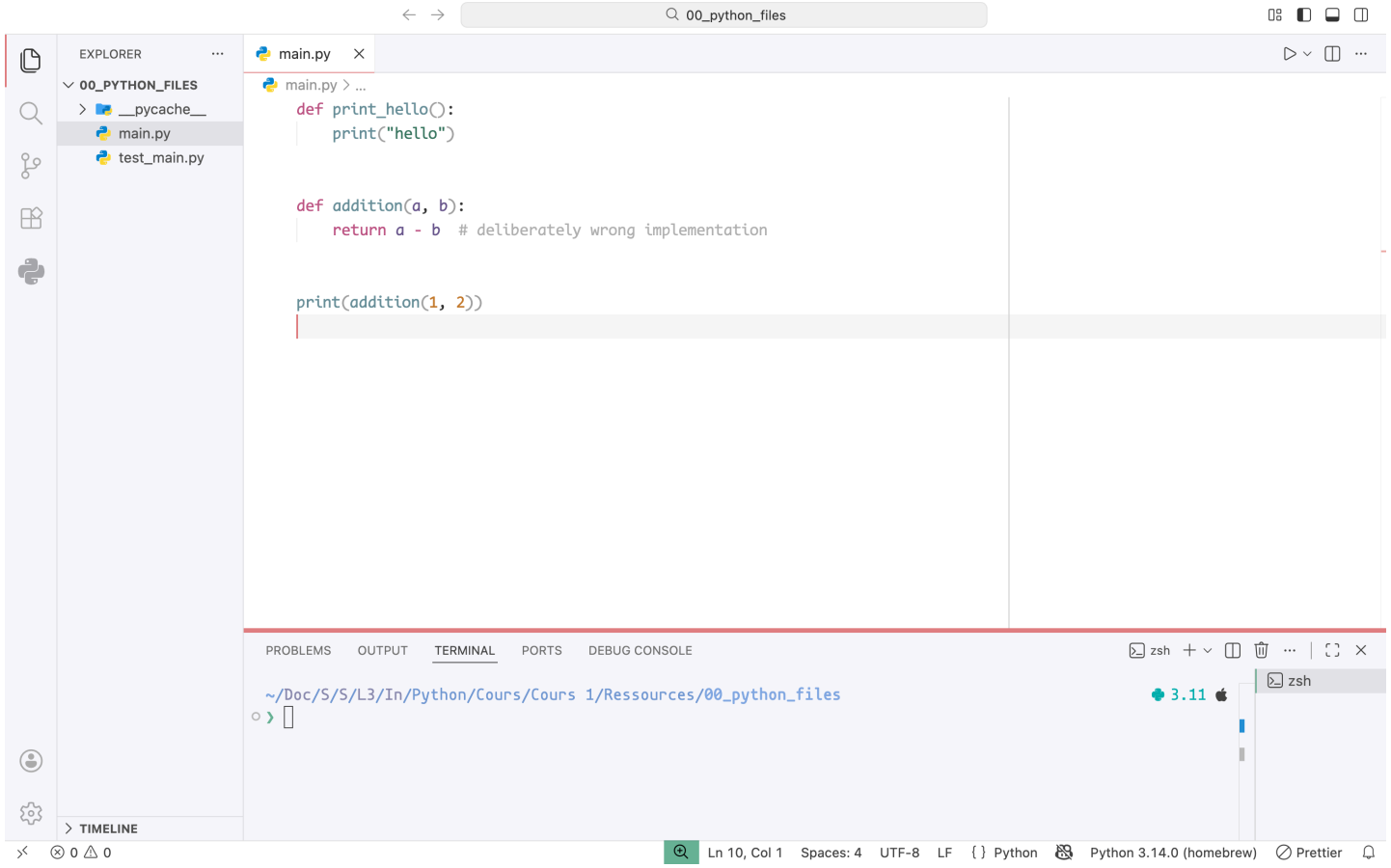
Prise en main de VSCode

Les bases de VSCode – Opérations courantes

- 📂 Ouvrir un dossier : `File` -> `Open Folder...`
- 📄 Ouvrir un fichier : `File` -> `Open File...`
- ✏️ Éditer un fichier : cliquer sur le fichier dans l'explorateur de fichiers à gauche
- 💾 Sauvegarder le fichier : `File` -> `Save` (ou `Ctrl+S` sur Windows/Linux, `Cmd+S` sur MacOS)
  - ⚠️ si vous modifiez un fichier, un indicateur `•` apparaissant dans l'onglet du fichier pour prévenir que vous avez des modifications non sauvegardées
- ❎ Fermer le fichier : `File` -> `Close` (ou `Ctrl+W` sur Windows/Linux, `Cmd+W` sur MacOS)
- 🔍 Rechercher dans le fichier : `Edit` -> `Find` (ou `Ctrl+F` sur Windows/Linux, `Cmd+F` sur MacOS)

Naviguer dans VSCode

VSCode est organisé en plusieurs panneaux :

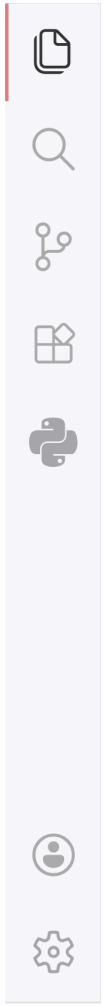


Naviguer dans VSCode

VSCode est organisé en plusieurs panneaux :

- La **barre d'activité** (généralement à gauche) : permet d'accéder à l'explorateur de fichiers, à la recherche, au contrôle de version, aux extensions, etc.
- Le **panneau latéral** affiche l'activité sélectionnée : l'explorateur de fichiers, la recherche, le contrôle de version, les extensions, etc.
- La **barre de titre** en haut : permet de rechercher un fichier ou une commande
- La **barre d'onglets** en haut : permet de naviguer entre les fichiers ouverts
- Le **panneau central** : permet d'éditer les fichiers
- Le **panneau inférieur** : permet d'afficher le terminal, les erreurs, les messages de debug, etc.
- La **barre d'état** en bas : affiche des informations sur l'état de VSCode (interpréteur python sélectionné, branche git, etc.)

Naviguer dans VSCode : la barre d'activité



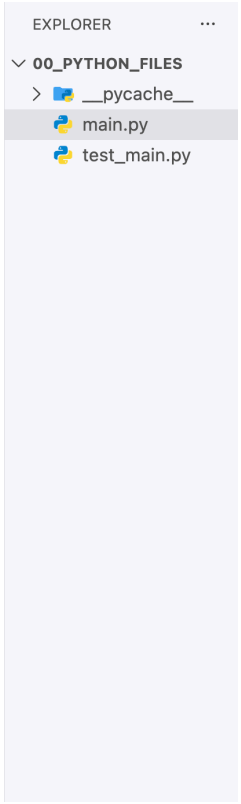
De haut en bas (les icônes peuvent varier selon les extensions installées et être réorganisées) :

- **Explorer** (icône de dossier) : permet de naviguer dans les fichiers et dossiers
- **Rechercher** (icône de loupe) : permet de rechercher du texte dans les fichiers
- **Contrôle de version** (icône de branche) : permet de gérer le code source avec git
- **Extensions** (icône de blocs carrés) : permet de gérer les extensions installées
- **Environnement python** (icône de serpent, si l'extension Python Environments est installée puis activée dans les réglages) : permet de gérer les versions, environnements et librairies python

Les deux dernières icônes (en bas) permettent de gérer les paramètres et les comptes (GitHub, etc.).

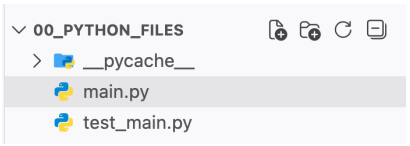
Naviguer dans VSCode : le panneau latéral

Le panneau latéral lorsque l'onglet **Explorer** est sélectionné dans la barre d'activité :



- Affiche les fichiers et dossiers du projet ouvert
- Permet de créer, supprimer, renommer des fichiers et dossiers
- Permet d'ouvrir un fichier en cliquant dessus

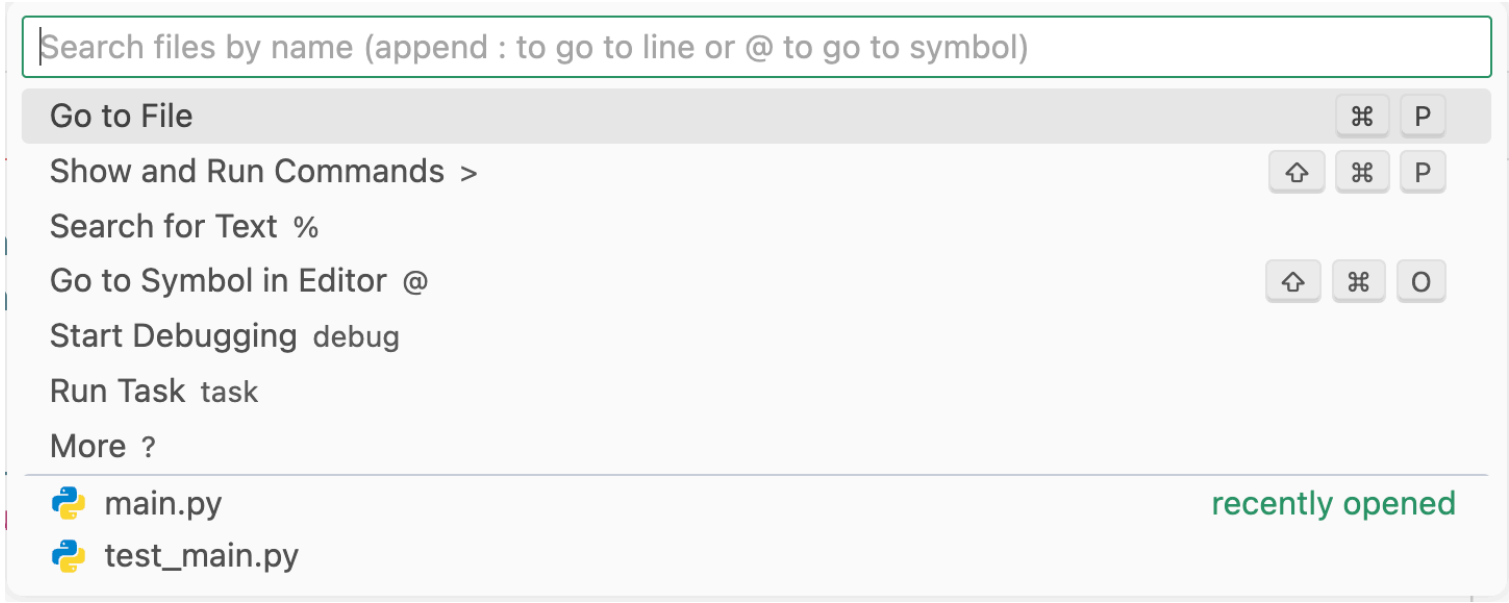
Au survol de l'explorateur, des icônes apparaissent pour effectuer des actions rapides (nouveau fichier, nouveau dossier, rafraîchir, etc.) :



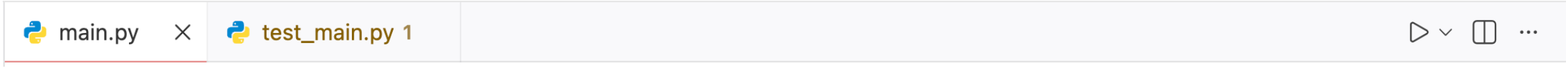
> Vous pouvez également faire un clic droit sur les fichiers et dossiers pour accéder à un menu contextuel avec des actions supplémentaires.

Naviguer dans VSCode : la barre de titre

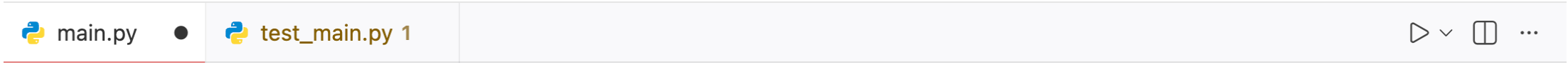
Se transforme en barre de recherche lorsque l'on clique dessus. Permet de rechercher des fichiers ou des commandes.



Naviguer dans VSCode : la barre d'onglets



- Affiche les fichiers ouverts sous forme d'onglets
- Permet de naviguer entre les fichiers ouverts en cliquant sur les onglets
- Permet de fermer un onglet en cliquant sur la croix `x` à droite
- Permet de réorganiser les onglets en les faisant glisser

> ⚠️ Surtout vous pouvez voir si un fichier a des modifications non sauvegardées grâce à l'indicateur `•` qui apparaît dans l'onglet du fichier.
> Ex ici, le fichier `main.py` a des modifications non sauvegardées :
>
> 
>

Naviguer dans VSCode : le panneau central

C'est le panneau principal où l'on édite les fichiers et écrit du code. Il occupe la majeure partie de l'interface.

Naviguer dans VSCode : le panneau inférieur

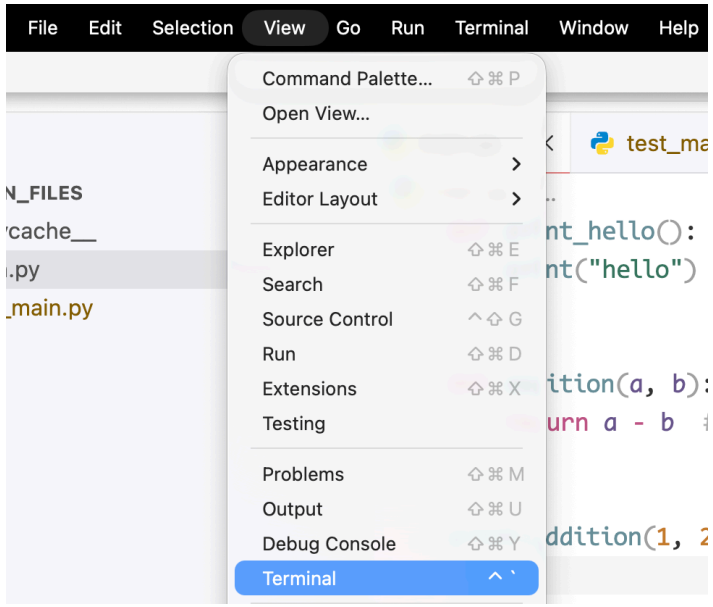
Permet d'afficher le **terminal**, les erreurs, les messages de debug, etc.

- Pour utiliser le terminal, il faut que l'onglet `Terminal` soit sélectionné dans le panneau inférieur.
- Le nom du terminal actif est affiché dans un onglet en haut du panneau inférieur (`zsh` dans la capture ci-dessous).
- Si plusieurs terminaux sont ouverts, une barre latérale (à droite) permet de naviguer entre eux.
- Pour créer un nouveau terminal, il suffit de cliquer sur le bouton `+` dans la barre d'onglets. Si plusieurs types de terminaux sont disponibles (bash, powershell, cmd, etc.), une petite flèche ▽ à côté du bouton `+` permet de choisir le type de terminal à ouvrir.

> Dans ce cours, lorsque nous rentrerons des commandes dans un terminal, nous utiliserons sans distinction l'interpréteur de commandes `zsh` ou `bash` ou `git bash` (disponible par défaut sur macOS et linux et disponible sur windows après l'installation de git).

Naviguer dans VSCode : le panneau inférieur

👁 Pour afficher / cacher le panneau inférieur : `View` -> `Terminal`



Naviguer dans VSCode : la barre d'état

- Affiche des informations sur l'état de VSCode (interpréteur python sélectionné, branche git, etc.)
- 🔍 Affiche et permet de modifier le niveau de **zoom** actuel
- 🐍 Permet de changer rapidement d'interpréteur python en cliquant sur la version de python affichée (ici `Python 3.14.0`)

Les bases de VSCode

Autres opérations courantes :

- ➕ Ouvrir une nouvelle fenêtre : `File` -> `New Window` (ou `Ctrl+Shift+N` sur Windows/Linux, `Cmd+Shift+N` sur MacOS)
- 🔧 Ouvrir les paramètres : `File` -> `Preferences` -> `Settings` (ou `Ctrl` + `,` sur Windows/Linux, `Cmd` + `,` sur MacOS)

Les notebooks Jupyter

Qu'est-ce qu'un notebook Jupyter ?

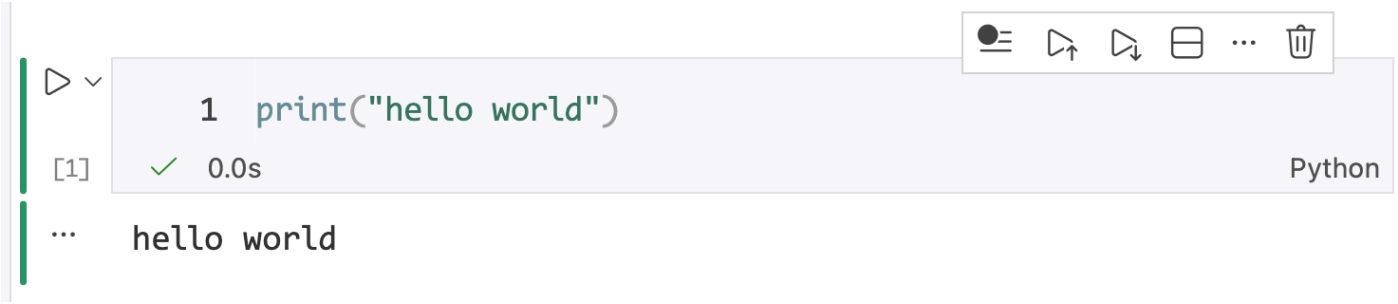
- Un notebook Jupyter est un environnement interactif pour écrire et exécuter du code
- Il permet de combiner du code, du texte formaté (Markdown), des images, des graphiques, etc. dans un même document
- Chaque notebook est composé de cellules qui peuvent contenir du code ou du texte
- Les notebooks Jupyter sont largement utilisés en science des données, apprentissage automatique, analyse de données, etc.

Jupyter : les 2 types de cellules (code et markdown)

1. Cellules de code : contiennent du code python qui peut être exécuté

```
[ ]   1  print("hello world")
                                                          Python
```

On peut exécuter une cellule de code en appuyant sur `Shift + Enter` ou en cliquant sur le bouton `Run` (▶) qui apparaît à gauche de la cellule lorsqu'on la survole avec la souris. Le résultat de l'exécution s'affiche juste en dessous de la cellule.



Jupyter : les 2 types de cellules (code et markdown)

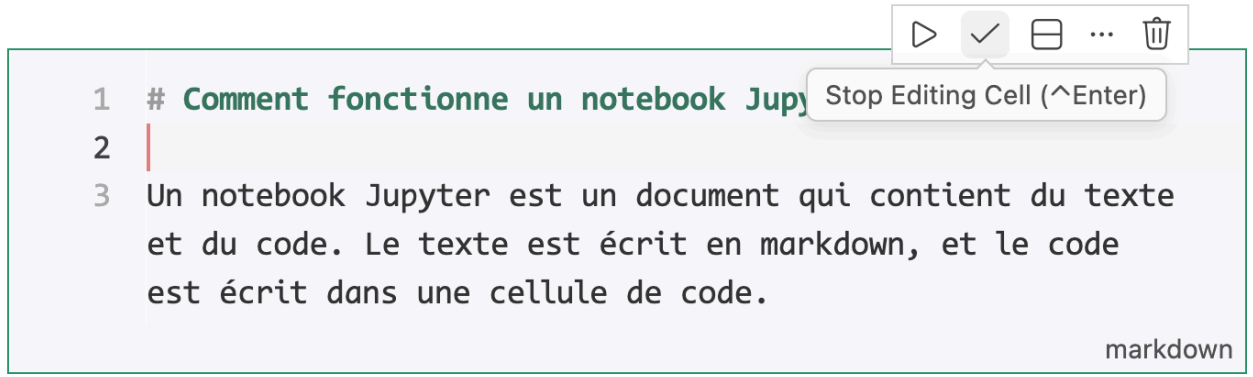
2️⃣ Cellules de texte (Markdown) : contiennent du texte formaté en markdown pour la documentation, les explications, etc.

```markdown
# Comment fonctionne un notebook Jupyter

Un notebook Jupyter est un document qui contient du texte et du code. Le texte est écrit en markdown, et le code est écrit dans une cellule de code.
```

Afficher la cellule de texte en mode édition / rendu

- Pour éditer une cellule de texte, double-cliquez dessus ou sélectionnez-la et appuyez sur `Enter`
- Pour afficher le rendu formaté, appuyez sur `Shift + Enter` ou cliquez sur le bouton `Validate` (✓) qui apparaît en haut de la cellule lorsqu'on la survole avec la souris.



Exemple de rendu d'une cellule de texte (Markdown)

Comment fonctionne un notebook Jupyter

Un notebook Jupyter est un document qui contient du texte et du code. Le texte est écrit en markdown, et le code est écrit dans une cellule de code.

Pour exécuter une cellule de code, il suffit de cliquer sur la cellule et d'appuyer sur `Shift + Enter`. Cela exécutera le code et affichera le résultat en dessous de la cellule. Il est également possible d'exécuter une cellule de code en cliquant sur le bouton ▷ à gauche de la cellule.

Syntaxe Markdown de base

- Titres : `# Titre 1`, `## Titre 2`, `### Titre 3`, etc.
- Texte en gras : `**texte en gras**`
- Texte en italique : `*texte en italique*`
- Listes à puces : `- élément 1`, `- élément 2`, etc.
- Listes numérotées : `1. élément 1`, `2. élément 2`, etc.
- Mise en forme du code à l'intérieur d'une ligne : `` `code` ``

Syntaxe Markdown (suite)

- Citations : `> Ceci est une citation`
- Liens : `[texte du lien](URL)`
- Images : `![texte alternatif](URL de l'image)`
- Blocs de code : utilisez des backticks triples (` ``` `) avant et après le code, avec le nom du langage optionnellement après les backticks d'ouverture (ex : ` ```python `)

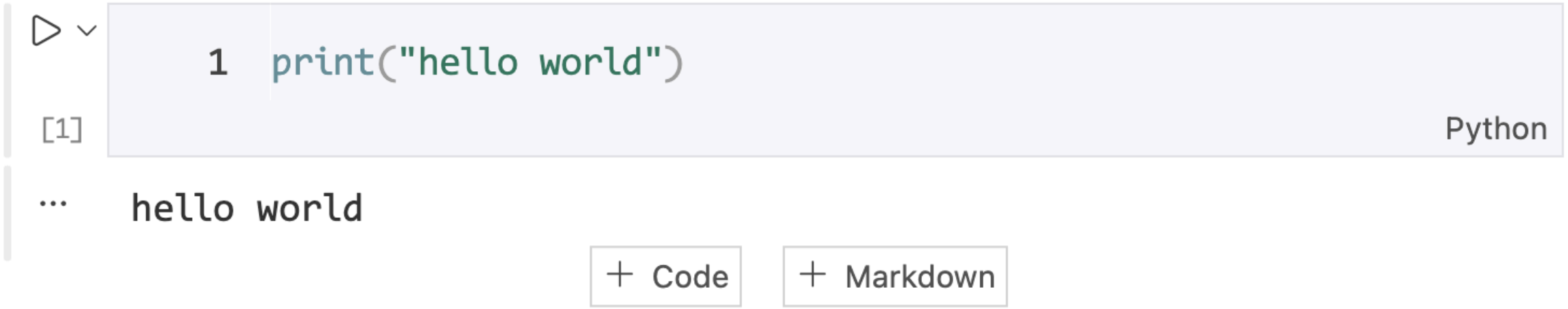
Ex :

````
```python
print("Hello, World!")
```
````

Insérer une cellule dans un notebook Jupyter

Survolez une cellule avec la souris pour faire apparaître les boutons d'insertion de cellule :

- Cliquez sur le bouton `+ Code` pour insérer une cellule de code python en dessous de la cellule courante
- Cliquez sur le bouton `+ Markdown` pour insérer une cellule de texte en markdown en dessous de la cellule courante



Choisir l'interpréteur python dans un notebook Jupyter

- Un notebook Jupyter utilise un **interpréteur python** pour exécuter le code
- Pour choisir l'interpréteur python, cliquez sur le sélecteur d'interpréteur en haut à droite du notebook (`Select Kernel` ou le nom de l'interpréteur actuel, ici `Python 3.11.5`)

> Avant la sélection de l'interpréteur :
>
> 
>

> Après la sélection :
>
> 
>

Pour fonctionner, l'interpréteur doit avoir le package `ipykernel` installé. Si ce n'est pas le cas, installez-le avec la commande :

```
pip install ipykernel
```

Restart & Run All dans un notebook Jupyter

Deux autres boutons importants en haut du notebook :

- Le bouton `Restart` (🔄) permet de redémarrer l'interpréteur python
  - Cela réinitialise toutes les variables et l'état du notebook
  - Utile pour s'assurer que le code fonctionne correctement depuis le début
  - Utile également en cas de problème avec l'interpréteur
- Le bouton `Run All` (▶️) permet d'exécuter toutes les cellules du notebook dans l'ordre

✅ S'assurer que le notebook fonctionne correctement

Au début des fichiers que nous utiliserons dans ce cours, vous trouverez souvent les deux blocs qui permettent de s'assurer que le notebook fonctionne correctement (il n'est pas utile que vous les compreniez pour l'instant) :

1️⃣ Vérifier que l'interpréteur python est bien sélectionné

```
print("✅ Python works!")
from sys import version
print(version)
```

Doit afficher quelque chose comme :

```
✅ Python works!
3.11.5 (tags/v3.11.5:1f2c4d3, Sep  5 2023, 15:22:19) [MSC v.1936 64 bit (AMD64)]
```

✅ S'assurer que le notebook fonctionne correctement (suite)

2️⃣ Vérifier que les packages nécessaires sont installés

```
import ipytest
ipytest.autoconfig()
ipytest.clean()
def test_all_good():
    assert "🐍" == "🐍"
ipytest.run()
```

Doit afficher quelque chose comme :

```
.
1 passed in 0.00s

<ExitCode.OK: 0>
```

✅ S'assurer que le notebook fonctionne correctement (suite)

> Si vous avez une erreur d'importation à l'étape précédente, cela signifie que le package `ipytest` n'est pas installé dans l'interpréteur python sélectionné. Installez-le avec la commande :

```
pip install ipytest
```

OU

```
pip3 install ipytest
```

Le langage python

Les commentaires en python

Un commentaire est une ligne de texte dans le code qui n'est **pas exécutée par l'ordinateur**

- Les commentaires servent à expliquer le code aux humains qui le lisent
- En python, on crée un commentaire en commençant la ligne avec le symbole `#`
- Exemple :

```
# Ceci est un commentaire
x = 5  # Ceci est aussi un commentaire
print(x)  # Affiche la valeur de x
```

> Il est également possible de faire des commentaires sur plusieurs lignes avec des **strings multilignes** (triple guillemets `"""` ou triple apostrophes `'''`).

Les variables en python

- Une variable est un nom qui référence une valeur en mémoire
- On crée une variable en lui assignant une valeur avec le symbole `=`
- On peut ensuite utiliser cette variable dans des expressions
- Exemples :

```
x = 5
y = 10
somme = x + y
print(somme)  # Affiche 15
```

Nommer les variables

- Les noms de variables doivent **commencer par une lettre ou** un underscore `_`
- Ils peuvent contenir des lettres, des chiffres et des underscores
- Ils sont **sensibles à la casse** (`maVariable` et `mavariable` sont deux variables différentes)
- Il est recommandé d'utiliser des noms descriptifs et en minuscules, avec des underscores pour séparer les mots (ex : `ma_variable`). Cette convention est appelée **snake_case**.

Outre ces considérations, il est important de choisir des **noms de variables clairs et significatifs** pour améliorer la lisibilité du code.

> "There are only two hard things in Computer Science: cache invalidation and naming things." – Phil Karlton

Les types de base en python

Les types de base les plus courants sont :

- `int` : nombres entiers (ex : `5`, `-3`, `42`)
- `float` : nombres à virgule flottante (ex : `3.14`, `-0.001`, `2.0`)
- `str` : chaînes de caractères (ex : `"Bonjour"`, `'Python'`)
- `bool` : valeurs booléennes (`True`, `False`)

🔍 On peut vérifier le type d'une variable avec la fonction `type()` :

```
x = 5
y = 10.0
z = "Hello"
print(type(x))  # Affiche <class 'int'>
print(type(y))  # Affiche <class 'float'>
print(type(z))  # Affiche <class 'str'>
```

> Nous verrons ces types de données plus en détail plus tard dans le cours. Nous verrons également d'autres types de données (listes, dictionnaires, etc.)

Les f-strings en python

Les f-strings (formatted string literals) permettent d'insérer des expressions dans des chaînes de caractères. C'est une façon pratique de formater des chaînes et nous nous en servirons souvent pour afficher des messages.

- On crée une f-string en préfixant la chaîne avec la lettre `f`
- On insère des expressions entre accolades `{}` dans la chaîne
- Exemple :

```
nom = "Alice"
age = 30
message = f"Bonjour, je m'appelle {nom} et j'ai {age} ans."
print(message)  # Affiche "Bonjour, je m'appelle Alice et j'ai 30 ans."
```

Les fonctions en python : définition

- Une fonction est un bloc de code réutilisable qui effectue une tâche spécifique
- On définit une fonction avec le **mot-clé** `def`, suivi du **nom de la fonction**, de **parenthèses** (avec d'éventuels **paramètres** séparés par des virgules) et d'un **deux-points**
- Le code de la fonction est **indenté** sous la définition

```
def bonjour():
    print("Bonjour tout le monde!")
```

```
def saluer(nom):
    message = f"Bonjour, {nom}!"
    return message
```

Les fonctions en python : appel

On appelle une fonction en utilisant son **nom** suivi de **parenthèses** contenant d'éventuels **arguments**.

En reprenant les exemples précédents :

```
bonjour()  # Appelle la fonction bonjour et affiche "Bonjour tout le monde!"
```

```
message = saluer("Alice")  # Appelle la fonction saluer avec l'argument "Alice"
print(message)  # Affiche "Bonjour, Alice!"
```

Les fonctions : valeur de retour

Les fonctions peuvent retourner des valeurs avec le mot-clé `return`

> Lorsqu'une fonction atteint une instruction `return`, elle termine son exécution et renvoie la valeur spécifiée

```
def addition(a, b):
    return a + b
```

```
resultat = addition(5, 3)  # Appelle la fonction addition avec les arguments 5 et 3
print(resultat)  # Affiche 8
```

Les fonctions : précisions

> Les fonctions **peuvent être définies n'importe où** dans le code, et elles peuvent être appelées avant ou après leur définition.

> Le **retour** d'une fonction est **facultatif**. Si aucune valeur n'est retournée, la fonction retourne `None`.

Les opérateurs de calcul en python

Pour vos premiers exercices, vous aurez peut-être besoin de faire des opérations de calcul. Les opérateurs de calcul courants sont :

- `+` : addition
- `-` : soustraction
- `*` : multiplication
- `/` : division
- `//` : division entière (arrondi à l'entier inférieur, ex : `7 // 2` donne `3`)
- `%` : modulo (reste de la division entière, ex : `7 % 2` donne `1`)
- `**` : puissance

> Nous verrons ces opérateurs plus en détail plus tard dans le cours.

Les structures de contrôle : conditions

Les conditions permettent d'exécuter du code de manière conditionnelle en fonction de la valeur d'une expression booléenne :

- On utilise les mots-clés `if`, `elif` (else if) et `else` pour créer des structures conditionnelles.
- Le code **indenté** sous chaque mot-clé est exécuté si la condition correspondante est vraie.

Exemple :

```
x = 10
if x > 0:
    print("x est positif")
elif x < 0:
    print("x est négatif")
else:
    print("x est zéro")
```

Conditions : opérateurs de comparaison

Les opérateurs de comparaison courants sont :

- `==` : égal à
- `!=` : différent de
- `<` : inférieur à
- `>` : supérieur à
- `<=` : inférieur ou égal à
- `>=` : supérieur ou égal à

Ces opérateurs retournent des valeurs booléennes (`True` ou `False`, la majuscule est importante) qui sont utilisées dans les conditions. Exemple :

```
a = 5
b = 10
print(a < b)  # Affiche True
print(a == b)  # Affiche False
```

Conditions : opérateurs logiques

Les opérateurs logiques permettent de combiner plusieurs conditions. Les principaux opérateurs logiques sont :

- `and` : vrai si les deux conditions sont vraies
- `or` : vrai si au moins une des conditions est vraie
- `not` : inverse la valeur de vérité d'une condition

Exemple :

```
x = 5
y = 10
if x < y and y > 0:
    print("Les deux conditions sont vraies.")
if x < y or y < 0:
    print("Au moins une des conditions est vraie.")
if not (x < y):
    print("La condition est fausse.")
```

Conditions : l'instruction `pass`

- L'instruction `pass` est une instruction vide qui ne fait rien
- Elle est utilisée comme un **espace réservé** dans les blocs de code où une instruction est requise, mais où aucune action n'est nécessaire pour le moment
- Exemple :

```
x = 5
if x > 0:
    pass  # TODO: Ajouter le code pour le cas où x est positif
else:
    print("x est zéro ou négatif")
```

> Cela permet de garder la structure du code intacte sans provoquer d'erreur de syntaxe.

Les boucles en python

Les boucles permettent d'exécuter un bloc de code plusieurs fois. Les deux types de boucles les plus courants en python sont :

- La boucle `for` : itère sur une séquence (liste, chaîne de caractères, etc.)
- La boucle `while` : exécute le code tant qu'une condition est vraie.

```python
# Boucle for
employees = ["Alice", "Bob", "Charlie"]
for employee in employees:
    print(employee) # Affiche chaque nom d'employé
```

```python
# Boucle while
i = 0
while i < 5:
    print(i)
    i = i + 1  # Affiche les nombres de 0 à 4
```

Boucles : ✍️ syntaxe de la boucle `for`

- La boucle `for` itère sur une séquence (liste, chaîne de caractères, etc.)
- On utilise le mot-clé `for`, suivi d'une **variable temporaire**, du mot-clé `in`, puis de la séquence à parcourir, et enfin d'un deux-points `:`
- Le code **indenté** sous la boucle est exécuté pour chaque élément de la séquence

```
fruits = ["pomme", "banane", "cerise"]
for fruit in fruits:
    print(fruit)  # Affiche chaque fruit
```

Boucles : la fonction `range()`

- La fonction `range()` génère une séquence de nombres entiers
- On peut l'utiliser avec une boucle `for` pour itérer sur une plage de nombres

```
for i in range(5):
    print(i)  # Affiche les nombres de 0 à 4
```

Il existe plusieurs façons d'utiliser `range()` :

- `range(n)` génère les nombres de `0` à `n-1` ⚠️.
- `range(a, b)` génère les nombres de `a` à `b-1`. (ex : `range(2, 5)` génère `2, 3, 4`)
- `range(a, b, step)` génère les nombres de `a` à `b-1` avec un pas de `step`. (ex : `range(2, 10, 2)` génère `2, 4, 6, 8`)

Boucles : ✍️ syntaxe de la boucle `while`

- La boucle `while` exécute le code tant qu'une condition est vraie
- On utilise le mot-clé `while`, suivi de la condition, et d'un deux-points `:`
- Le code **indenté** sous la boucle est exécuté tant que la condition est vraie
- ⚠️ Il est important de modifier la condition à l'intérieur de la boucle pour **éviter les boucles infinies**

```
i = 0
while i < 5:
    print(i)
    i = i + 1  # Incrémente i pour éviter une boucle infinie
```

Boucles : l'instruction `break` et `continue`

- L'instruction `break` permet de sortir immédiatement d'une boucle, même si la condition n'est pas encore fausse
- L'instruction `continue` permet de passer à l'itération suivante de la boucle sans exécuter le reste du code dans la boucle pour l'itération en cours

```
# Exemple avec break
for i in range(10):
    if i == 5:
        break  # Sort de la boucle lorsque i vaut 5
    print(i)  # Affiche les nombres de 0 à 4
```

```
# Exemple avec continue
for i in range(10):
    if i == 5:
        continue  # Passe à l'itération suivante lorsque i vaut 5
    print(i)  # Affiche les nombres de 0 à 4 et de 6 à 9
```