

Les types en Python

Les types primitifs

Les types primitifs sont les types de données les plus simples. Ils sont:

- int
- float
- str
- bool
- list
- tuple
- dict

Un **str** est une chaîne de caractères. Il est défini par la liste des caractères entre guillemets.

Supprimez tout ce qu'il y a dans le fichier **main.py** et remplacez-le par le code suivant:

```
prenom = "Jean"
nom = 'Dupont'
phrase = "Bonjour, je m'appelle " + prenom + " " + nom
print(prenom)
print(nom)
print(phrase)
```

Pour exécuter le programme, tapez **uv run main.py** dans votre terminal.

Vous devriez voir le résultat suivant:

```
Jean
Dupont
Bonjour, je m'appelle Jean Dupont
```

Un **int** est un type de données qui représente des nombres entiers.

```
age = 42
print(age)
```

Un **float** est un type de données qui représente des nombres décimaux.

```
prix = 12.99
print(prix)
```

list est un type de données qui représente une liste de valeurs.

```
courses = ["Oeufs", "Lait", "Farine", "Chocolat"]
print(courses)
```

```
['Oeufs', 'Lait', 'Farine', 'Chocolat']
```

Si on souhaite afficher un élément d'une liste, on doit utiliser l'**index** de position de l'élément.

```
courses = ["Oeufs", "Lait", "Farine", "Chocolat"]
# index      0      1      2      3
print(courses[0])
print(courses[1])
```

En informatique, on commence à compter les indices à 0.

On utilise le **#** pour commenter le code.

```
# Cette ligne de code est un commentaire
```

Python nous permet de récupérer le dernier élément d'une liste avec l'index **-1**.

```
courses = ["Oeufs", "Lait", "Farine", "Chocolat"]
#          -4      -3      -2      -1
print(courses[-1])
```

On peut ainsi remonter depuis la fin de la liste jusqu'à la première valeur en utilisant les index négatifs.

On appelle ça du "**sucre syntaxique**".

Le sucre syntaxique désigne une simplification de l'écriture du code, permettant d'exprimer une construction plus complexe de manière plus concise et lisible, sans ajouter de nouvelles fonctionnalités au langage. Habituellement on récupère le dernier élément d'une liste avec la fonction `len()`. la fonction `len()` retourne le nombre d'éléments d'une liste.

```
courses[len(courses) - 1]
```

On peut modifier un élément d'une liste avec l'index de position.

```
courses = ["Oeufs", "Lait", "Farine", "Chocolat"]
courses[1] = "Lait de soja"
print(courses)
```

```
['Oeufs', 'Lait de soja', 'Farine', 'Chocolat']
```

Le type `str` partage des propriétés avec le type `list`. En effet, une chaîne de caractères n'est autre qu'une "liste" de caractères qui se suivent.

Ainsi on peut :

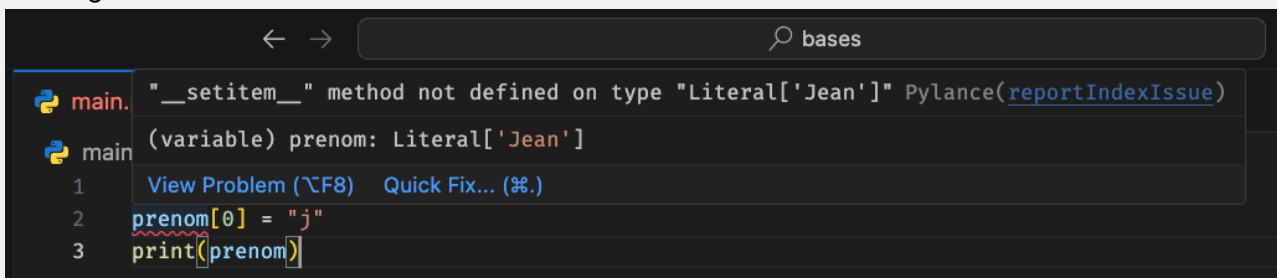
```
prenom = "Jean"
print(prenom[0])
print(prenom[-1])
```

Par contre, à l'instar des listes, on ne peut pas modifier une chaîne de caractères.

```
prenom = "Jean"
prenom[0] = "j"
```

```
Traceback (most recent call last):
  File "...", line 2, in <module>
    prenom[0] = "j"
    ~~~~~^~
TypeError: 'str' object does not support item assignment
```

VsCode est capable de détecter les erreurs de syntaxe et de type. Pour cela il suffit d'ouvrir le `settings.json` avec le raccourci `Ctrl + Maj + P` (ou `Cmd + Maj + P` sur Mac) et taper : `Preferences: Open Settings (JSON)`. D'ajouter la ligne dans les accolades : `"python.analysis.typeCheckingMode": "basic"` Vous devriez voir votre fichier `main.py` passer en rouge, avec la ligne d'erreur. Et si vous survolez sur la ligne d'erreur, vous devriez voir le message suivant :



Nous verrons plus tard pourquoi l'erreur est différente sur vscode et le terminal.

Un **dict** est un type de données qui représente un dictionnaire. Un dictionnaire est un ensemble de paires clé-valeur.

```
infos = {  
    "username": "johndoe2025",  
    "password": "qwerty",  
}
```

On peut accéder à une valeur d'un dictionnaire avec sa clé, toujours en utilisant les crochets.

```
print(infos["username"])
```

Si vous essayez d'accéder à une clé inexistante, vous obtiendrez une erreur.

```
print(infos["inexistante"])
```

```
print(infos['inexistante'])  
~~~~~  
KeyError: 'inexistante'
```

On peut modifier une valeur d'un dictionnaire avec sa clé.

```
infos["username"] = "johndoe2030"  
print(infos)
```

On peut enfin **ajouter** une clé et sa valeur à un dictionnaire.

```
infos["age"] = 42  
print(infos)
```

```
{'username': 'johndoe2030', 'password': 'qwerty', 'age': 42}
```

Un **tuple** est un type de données qui représente une "liste" de valeurs, mais qui **ne peut pas être modifiée**.
On la différencie de la liste en utilisant des **parenthèses**.

```
users = ("Jean", "Dupont", "Jean-Pierre")  
print(users)
```

Tout comme une liste, on peut **accéder** à une valeur d'un tuple en utilisant l'index.

```
print(users[0])  
print(users[-1])
```

Si on essaye de modifier une valeur d'un tuple, on obtiendra une erreur.

```
users[0] = "jean"
```

```
users[0] = "jean"  
~~~~~  
TypeError: 'tuple' object does not support item assignment
```

Là aussi, VsCode vous alerte qu'il y a une erreur sur cette ligne.

Un **set** est un type de données qui représente un ensemble de valeurs uniques

```
nombres = [1,2,2,2,2,3,3,3,4,4,5,5,6,6,7,7,8,8,9,9,0,0,0]  
nombres_uniques = set(nombres)  
print(nombres_uniques)
```

On convertit ici une liste en un set.

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Attention, il faut bien différencier un **set** d'un **dictionnaire**. Car les deux utilisent des parenthèses pour contenir des éléments.

Un **bool** est un type de données qui représente des valeurs booléennes. Ils peuvent être **True** ou **False**.

```
is_admin = True  
has_admin_role = False
```

En programmation, on choisit souvent de nommer les variables booléennes en commençant par **is**, **has** ou par un **verbe conjugué**, parce que cela traduit naturellement une question à laquelle la variable répond par *vrai* ou *faux*.

Par exemple, si l'on écrit `is_admin = True`, on peut lire cette variable comme une phrase : « est-ce un administrateur ? » et la réponse est *oui*. De la même façon, `has_admin_role = False` se lit comme « a-t-il un rôle d'administrateur ? » et la réponse est *non*. On pourrait aussi utiliser des verbes comme *can* (*peut-il*), *should* (*devrait-il*) ou *must* (*doit-il*), ce qui renforce encore la logique de question/réponse.

Cette convention rend le code plus clair et plus lisible : on comprend immédiatement que la variable est un booléen et qu'elle exprime une condition vraie ou fausse, sans avoir besoin de lire le reste du programme.

Ainsi nous pouvons écrire :

```
if is_admin: # Doit être vrai
    print("Il est administrateur")
else:
    print("Il n'est pas administrateur")
```

Littéralement, cette convention d'écriture permet de traduire : **Si il est admin**

Dans une condition, le `if` doit être vrai pour que le code entre le `if` et le `else` s'exécute. Sinon, il exécute le code dans le `else`.

Si on passe la valeur `False` à la variable `is_admin`, on obtiendra le résultat suivant :

```
Il n'est pas administrateur
```

De même dans le code suivant

```
if not has_admin_role:
    print("Il n'a pas de rôle d'administrateur")
else:
    print("Il a un rôle d'administrateur")
```

Le `if` doit être toujours vrai et c'est le cas ici. En effet :

```
not has_admin_role => not False => True
```

L'indentation

En Python, l'indentation n'est pas seulement une convention esthétique : elle fait partie intégrante de la syntaxe du langage. Contrairement à d'autres langages comme C, Java ou JavaScript, qui utilisent des accolades pour délimiter les blocs de code, Python se sert de l'espace visuel laissé en début de ligne pour indiquer la structure logique d'un programme.

```
if not has_admin_role:
    print("Il n'a pas de rôle d'administrateur")
else:
    print("Il a un rôle d'administrateur")
```

Cette approche découle de la philosophie de conception de Python : rendre le code lisible et explicite. Le créateur du langage, Guido van Rossum, souhaitait éliminer la possibilité d'écrire un code correctement exécuté **mais illisible**. En imposant une indentation obligatoire, Python oblige le programmeur à aligner la logique du programme sur sa présentation visuelle. Ainsi, la hiérarchie des instructions devient immédiatement perceptible à la lecture.

Lorsqu'on écrit une condition, une boucle ou une fonction, le bloc de code qui en dépend doit être indenté du même nombre d'espaces ; en général, on utilise **quatre espaces** (ou la touche **Tab** pour aller plus vite). Ce décalage indique à l'interpréteur quels ordres appartiennent à ce bloc. Si l'indentation n'est pas cohérente, Python renvoie une erreur de syntaxe, car il ne peut plus déterminer les limites de chaque structure de contrôle.

Cette exigence renforce la discipline de codage : chaque développeur écrit des blocs visuellement uniformes, ce qui réduit les ambiguïtés et rend la lecture collective du code plus fluide. L'indentation en Python matérialise donc le principe fondamental du langage : la clarté prime sur la complexité.

Les Itérables

Un **itérable** est un objet que l'on peut parcourir élément par élément à l'aide d'une boucle. En Python, les listes, les tuples, les ensembles ou encore les dictionnaires sont des itérables, car on peut utiliser **for element in objet** : pour accéder successivement à chacun de leurs éléments. Derrière cela, un itérable est tout objet qui implémente un mécanisme permettant de produire un **itérateur**, c'est-à-dire une entité qui sait donner le prochain élément jusqu'à épuisement de la séquence.

Concernant la convention de nommage, on met les itérables au **pluriel** pour signaler qu'ils contiennent potentiellement plusieurs éléments. Par exemple, **users** laisse entendre qu'il s'agit d'une collection de plusieurs utilisateurs, tandis que **user** désigne un seul utilisateur. Cette distinction entre singulier et pluriel permet de comprendre immédiatement, en lisant le nom de la variable, si l'on manipule une entité unique ou une collection d'éléments, ce qui améliore la lisibilité et réduit les ambiguïtés dans le code.

Ainsi on peut écrire :

```
for user in users:
    print(user)
```

Même si on peut utiliser **i, j, k**, etc. à la place de **user** pour les itérateurs, il est préférable de nommer les itérables en utilisant des noms plus explicites. En utilisant la version singulière, comme ici **user**

Ou on utilisera un nom explicite comme:

```
for article in courses:  
    print(article)
```

Les codes couleurs de VSCode

Dans Visual Studio Code, les couleurs du texte ne sont pas choisies au hasard : elles servent à distinguer les différents éléments du code et à rendre la lecture plus claire.

Quand on voit un nom écrit en **bleu**, il s'agit généralement d'une **variable**. Une variable stocke une valeur, et on l'utilise telle quelle, sans parenthèses. Par exemple **age** ou **prenom**.

```
age = 42  
prenom = "Jean"
```

Quand le nom apparaît en **jaune**, cela correspond à une **fonction** ou une **méthode**. Une fonction ou une méthode doit être **appelée** pour s'exécuter, et cela se fait avec des parenthèses. Par exemple **print("Bonjour")** ou **prenom.upper()**. Sans les parenthèses, on fait seulement référence à la fonction elle-même, mais on ne l'exécute pas.

```
print(prenom.upper)  
  
### Sortie : <built-in method upper of str object at 0x104368f90>
```

Le **upper** est en jaune, il faut donc l'invoquer avec des parenthèses.

Enfin, en **vert**, on a les **classes**, qui servent de modèles pour créer des objets. Plus tard, lorsqu'on apprendra à manipuler des classes, on verra que pour en créer une instance, il faut aussi utiliser des parenthèses, comme **MonObjet()**.

Ainsi, retenir ces codes couleurs aide à comprendre visuellement la nature de ce que l'on lit : **bleu** pour une variable, **jaune** pour quelque chose qu'il faut appeler avec des parenthèses, et **vert** pour une classe qui sert de plan de construction aux objets.

Fonctions et méthodes : Introduction

Quand on écrit du code, on peut appeler des **fonctions** ou des **méthodes**, et même si la construction technique sera vue plus tard, il est important de comprendre la différence dans l'utilisation.

Une **fonction** est une action que l'on appelle directement par son nom, en lui donnant éventuellement des valeurs à traiter. Par exemple, **print("Bonjour")** affiche du texte à l'écran, et **len([1, 2, 3])** renvoie la taille de la liste. On donne quelque chose à la fonction et elle renvoie un résultat ou effectue une action.

Une **méthode**, en revanche, est une action qui est **attachée** à un objet précis. On l'appelle avec la **notation pointée** (Dot Notation). Par exemple, si on a **prenom = "Alice"**, écrire **prenom.upper()** signifie «

prends cette chaîne de caractères et transforme-la en majuscules ». Ici, la méthode est appliquée spécifiquement à l'objet `prenom`. Donc `upper()` est une méthode de la classe `str`.

On peut donc dire que les fonctions sont générales et s'appliquent à ce qu'on leur transmet, tandis que les méthodes sont des comportements liés à un objet particulier et que l'on invoque directement à partir de cet objet.

Voici une liste d'exemples de **fonctions intégrées** (built-in) en Python :

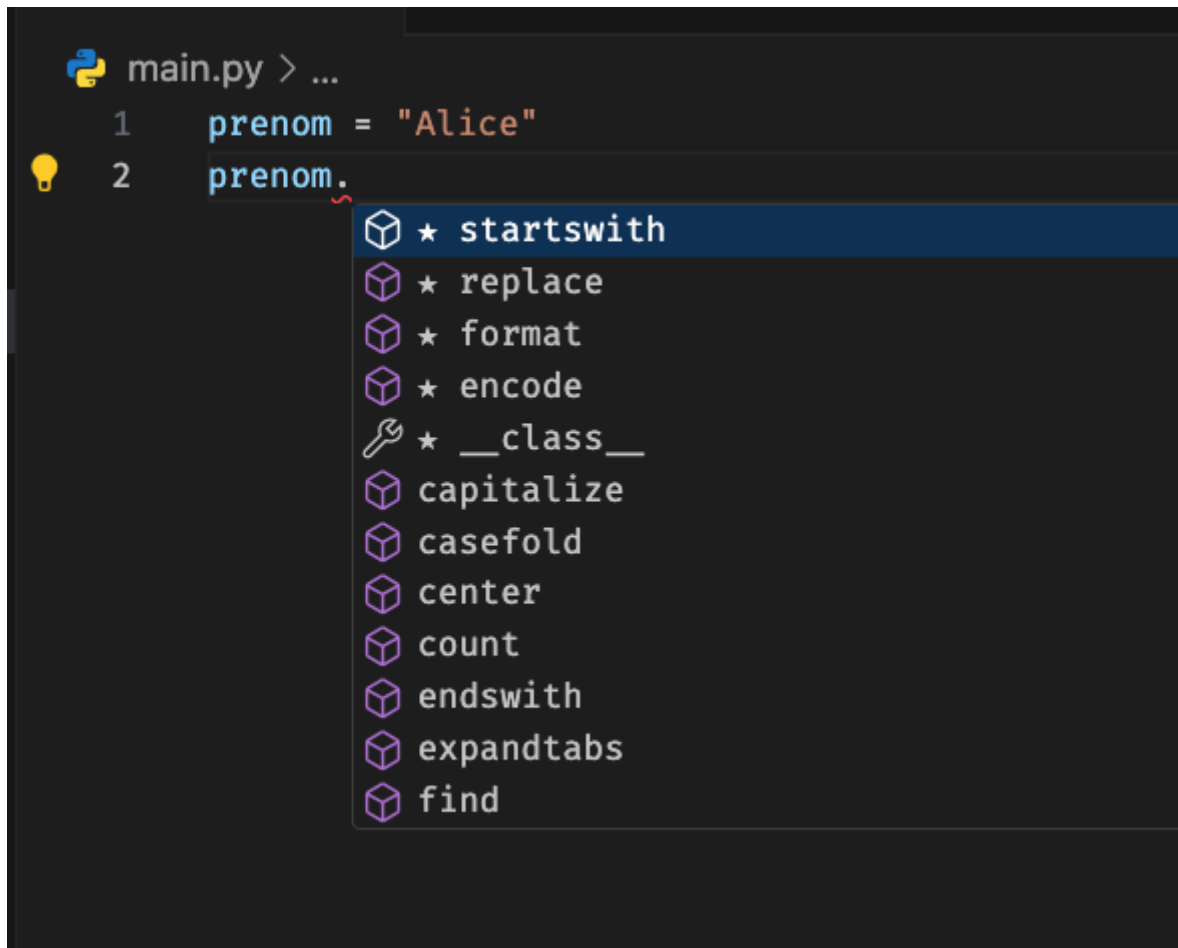
- `print("Bonjour")` : affiche du texte ou une valeur.
- `len([1, 2, 3])` : donne la taille d'une liste, ici `3`.
- `sum([4, 5, 6])` : calcule la somme des éléments, ici `15`.
- `max(7, 2, 9)` : renvoie la plus grande valeur, ici `9`.
- `min(7, 2, 9)` : renvoie la plus petite valeur, ici `2`.
- `abs(-10)` : donne la valeur absolue, ici `10`.
- `round(3.14159, 2)` : arrondit à un certain nombre de décimales, ici `3.14`.
- `sorted([3, 1, 2])` : renvoie une nouvelle liste triée, ici `[1, 2, 3]`.

Ces fonctions ont toutes le même principe : on leur transmet une ou plusieurs valeurs entre parenthèses (des **arguments**), et elles effectuent un calcul ou une action avant de renvoyer un résultat.

Les méthodes de la classe `str`

Chaque type de données possède des méthodes qui s'appliquent à ce type de données.

VSCode est là aussi pour nous aider à comprendre les méthodes de la classe `str`. Quand je mets un point juste après un nom de variable, je vois la liste des méthodes disponibles.



```
main.py > ...
1 prenom = "Alice"
2 prenom.
   * startswith
   * replace
   * format
   * encode
   * __class__
   capitalize
   casefold
   center
   count
   endswith
   expandtabs
   find
```

Donc une méthode est une fonction qui est attachée à un objet précis. Ici, pour la variable `prenom`, on l'appelle avec la notation pointée, et VSCode nous liste les méthodes disponibles.

La liste proposée est différente selon vos dernières utilisations.

Par exemple, ici je peux voir que la première méthode proposée est `startswith()`. Plus haut nous avons vu que les standards d'écriture disaient que tout ce qui commence par `is`, `has` ou un **verbe conjugué** renvoie un booléen. Et c'est le cas ici.

`startswith()` renvoie `True` si le prénom commence par une lettre saisie entre parenthèses, et `False` sinon.

```
print(prenom.startswith("j"))
```

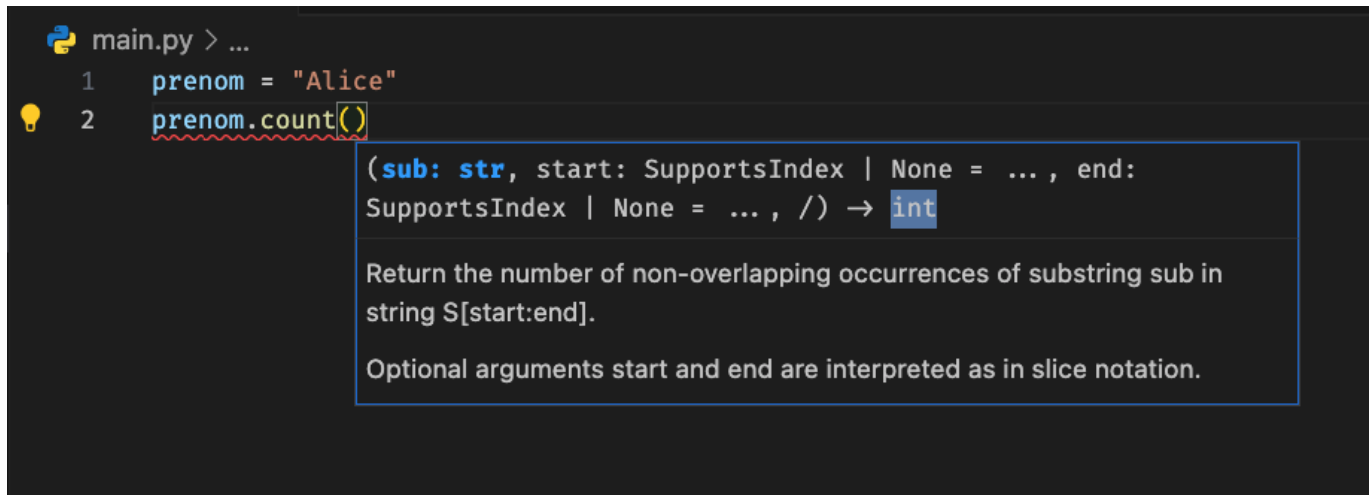
Vous avez d'autres méthodes qui commencent par `is`. Comme :

- `isupper()` dit si la chaîne est en majuscules
- `islower()` dit si la chaîne est en minuscules
- `isdigit()` dit si la chaîne contient des chiffres
- `isalpha()` dit si la chaîne contient des lettres

A partir du moment où la méthode commence par `is`, on comprend facilement que la méthode renvoie un booléen.

Autre point important : vous n'êtes pas obligé de connaître toutes les méthodes de la classe `str` par coeur. Il suffit de vous aider de VSCode. En effet une fois que vous ouvrez les parenthèses, VSCode vous proposera une boîte de dialogue avec l'explication de la méthode.

Par exemple avec la méthode `count()`, dès qu'on ouvre les parenthèses :



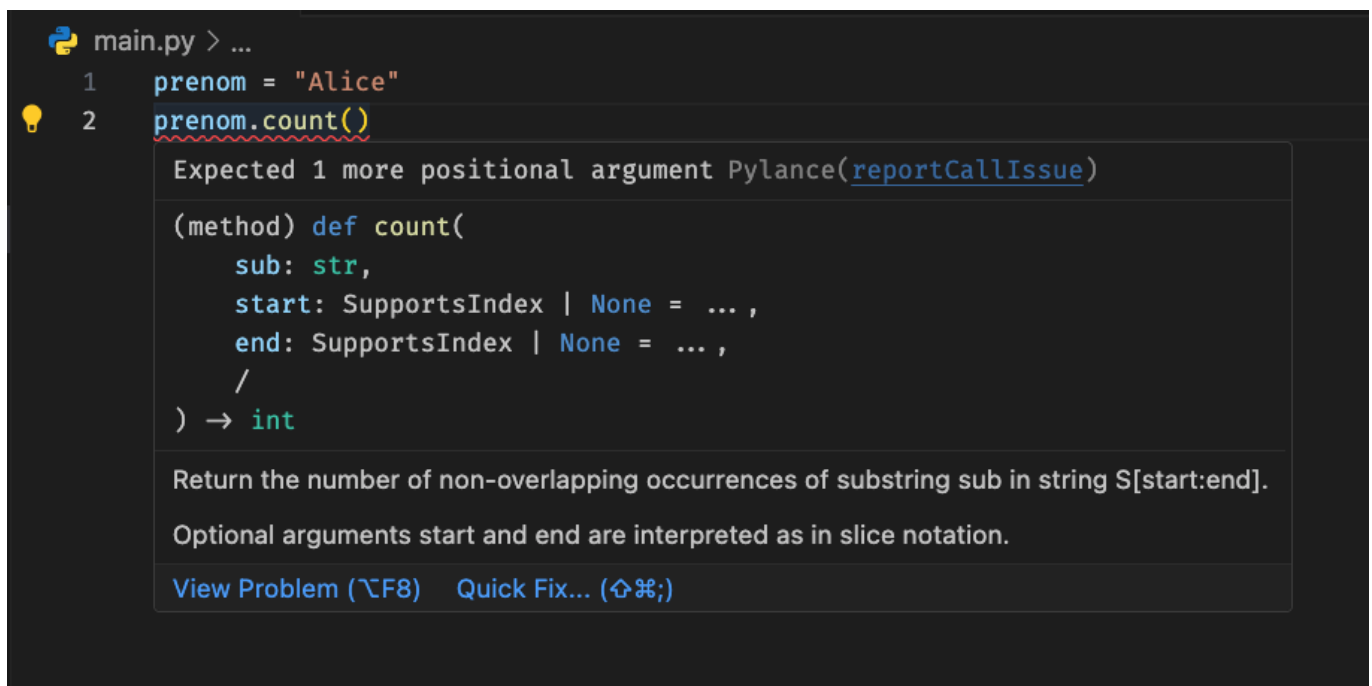
```
main.py > ...
1 prenom = "Alice"
2 prenom.count()
```

(sub: str, start: SupportsIndex | None = ..., end: SupportsIndex | None = ..., /) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

Ou en survolant la méthode `count()`



```
main.py > ...
1 prenom = "Alice"
2 prenom.count()
```

Expected 1 more positional argument Pylance([reportCallIssue](#))

(method) def count(
 sub: str,
 start: SupportsIndex | None = ...,
 end: SupportsIndex | None = ...,
 /
) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

[View Problem](#) (⌘F8) [Quick Fix...](#) (⇧⌘;)

Cette boîte est précieuse car elle donne tous les éléments nécessaires pour comprendre comment utiliser correctement la méthode.

Message d'erreur en haut : « *Expected 1 more positional argument* » indique qu'il manque un argument obligatoire. Ici, la méthode `count()` attend qu'on précise ce qu'on veut compter dans la chaîne de caractères.

Signature de la méthode (Le plus important) :

```
(method) def count(  
    sub: str,
```

```
    start: SupportsIndex | None = ...,  
    end: SupportsIndex | None = ...,  
    ) -> int
```

- C'est une méthode (**method**)
- **sub: str** : premier paramètre obligatoire, une sous-chaîne (**str**) que l'on cherche.
- **start** et **end** : paramètres optionnels qui permettent de limiter la recherche à une portion de la chaîne.
- **-> int** : signifie que la méthode renvoie un entier (le nombre d'occurrences trouvées).

On reconnaît les paramètres optionnels avec le **| None = ...**

Description : La boîte explique en anglais ce que fait la méthode : *Return the number of non-overlapping occurrences of substring sub in string S[start:end]*. Autrement dit, elle renvoie le nombre d'occurrences (sans chevauchement) d'une sous-chaîne dans la chaîne, éventuellement limitée à une portion définie par **start** et **end**.

Ces boîtes de dialogue que VS Code affiche quand on survole une fonction ou une méthode sont comme un petit dictionnaire intégré au langage. Leur importance est grande : elles permettent d'apprendre à utiliser une méthode sans avoir à mémoriser toute la documentation par cœur.

Quand on programme, il est impossible de retenir chaque méthode, ses paramètres et ses subtilités. En revanche, si on sait **lire et décrypter** ces boîtes, on devient autonome :

- On sait immédiatement quels arguments sont obligatoires et lesquels sont optionnels.
- On sait quel type de valeur est attendu (**str**, **int**, etc.) et quel type de résultat sera renvoyé.
- On a une description claire, souvent avec des indices de vocabulaire ou des rappels de syntaxe (par exemple, que **start** et **end** fonctionnent comme dans un découpage de chaîne).

Cela change la manière d'apprendre : au lieu d'essayer de tout retenir par cœur, on développe un **réflexe de lecture**. Chaque fois qu'on oublie comment une méthode fonctionne, il suffit de passer la souris dessus pour retrouver la bonne utilisation.

En pratique, cela veut dire que le plus important n'est pas de connaître toutes les méthodes par avance, mais d'apprendre à lire et comprendre ces indications. C'est comme avoir un guide toujours disponible : si on sait le décrypter, on n'est jamais bloqué et on peut avancer dans le code sans perdre de temps.

D'autres méthodes de la classe **str**

Voici quelques méthodes utiles sur les chaînes de caractères (**str**) en Python et leur utilisation :

- **upper()** : transforme tout en majuscules. Exemple :

```
"alice".upper() # "ALICE"
```

- **lower()** : transforme tout en minuscules. Exemple :

```
"ALICE".lower() # "alice"
```

- `capitalize()` : met seulement la première lettre en majuscule. Exemple :

```
"alice".capitalize() # "Alice"
```

- `strip()` : enlève les espaces au début et à la fin. Exemple :

```
"  Alice  ".strip() # "Alice"
```

- `replace("a", "o")` : remplace un caractère ou un mot par un autre. Exemple :

```
"Alice".replace("A", "O") # "Olice"
```

- `startswith("A")` : teste si la chaîne commence par le caractère ou mot indiqué. Exemple :

```
"Alice".startswith("A") # `True`
```

- `endswith("e")` : teste si la chaîne se termine par le caractère ou mot indiqué. Exemple :

```
"Alice".endswith("e") # `True`
```

- `count("i")` : compte le nombre d'occurrences. Exemple :

```
"Alice".count("i") # `1`
```

- `find("i")` : donne l'indice de la première occurrence trouvée. Exemple :

```
"Alice".find("i") # `2`
```

Les méthodes de la classe `list`

`append(x)` ajoute un élément **en fin** de liste.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.append(6)
print(ma_liste)  # [1, 2, 3, 4, 5, 6]
```

`insert(i, x)` insère un élément à un indice donné.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.insert(2, 99)
print(ma_liste)  # [1, 2, 99, 3, 4, 5]
```

`remove(x)` supprime la première occurrence d'une valeur.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.remove(3)
print(ma_liste)  # [1, 2, 4, 5]
```

`pop(i)` retire et renvoie l'élément à l'indice `i` (par défaut le dernier).

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste.pop())  # 5
print(ma_liste)        # [1, 2, 3, 4]

ma_liste = [1, 2, 3, 4, 5]
print(ma_liste.pop(1)) # 2
print(ma_liste)        # [1, 3, 4, 5]
```

`clear()` vide la liste.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.clear()
print(ma_liste)  # []
```

`index(x)` renvoie l'indice de la première occurrence.

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste.index(4))  # 3
```

`count(x)` compte les occurrences.

```
ma_liste = [1, 2, 3, 4, 5]
print(ma_liste.count(2)) # 1
```

`sort()` trie la liste sur place.

```
ma_liste = [5, 3, 1, 4, 2]
ma_liste.sort()
print(ma_liste) # [1, 2, 3, 4, 5]

ma_liste.sort(reverse=True)
print(ma_liste) # [5, 4, 3, 2, 1]
```

`reverse()` inverse l'ordre des éléments.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.reverse()
print(ma_liste) # [5, 4, 3, 2, 1]
```

`copy()` renvoie une copie superficielle.

```
ma_liste = [1, 2, 3, 4, 5]
copie = ma_liste.copy()
print(copie) # [1, 2, 3, 4, 5]
```

`extend(itérable)` ajoute en fin de liste tous les éléments d'un itérable.

```
ma_liste = [1, 2, 3, 4, 5]
ma_liste.extend([6, 7])
print(ma_liste) # [1, 2, 3, 4, 5, 6, 7]
```

La majorité des méthodes **modifie** la liste originale. Seules les méthodes suivantes renvoient une valeur ou **une copie** de la liste :

```
ma_liste.index(x) # renvoie l'indice de l'élément
ma_liste.count(x) # renvoie le nombre d'occurrences
ma_liste.copy()   # renvoie une copie superficielle de la liste
```

Cas particulier

`pop(i)` modifie la liste et renvoie l'élément supprimé. Pour obtenir une version triée sans modifier l'originale, on utilise la fonction globale `sorted(ma_liste)` et non la méthode `.sort()`.

Les méthodes de la classe `dict`

```
mon_dict = {"nom": "Alice", "age": 25, "ville": "Paris"}
```

`get(cle, valeur_par_defaut)`

Renvoie la valeur associée à une clé. Si la clé n'existe pas, renvoie une valeur par défaut (ou `None` si rien n'est précisé).

```
print(mon_dict.get("age"))      # 25
print(mon_dict.get("pays", "FR")) # "FR"
```

`keys()`

Renvoie un objet contenant toutes les clés du dictionnaire.

```
print(mon_dict.keys()) # dict_keys(['nom', 'age', 'ville'])
```

`values()`

Renvoie un objet contenant toutes les valeurs.

```
print(mon_dict.values()) # dict_values(['Alice', 25, 'Paris'])
```

`items()`

Renvoie des couples (`clé`, `valeur`). Très utile pour parcourir le dictionnaire.

```
print(mon_dict.items()) # dict_items([('nom', 'Alice'), ('age', 25), ('ville', 'Paris')])
```

`update(autre_dict)`

Met à jour le dictionnaire avec les clés/valeurs d'un autre dictionnaire.

```
mon_dict.update({"age": 26, "pays": "France"})
print(mon_dict) # {'nom': 'Alice', 'age': 26, 'ville': 'Paris', 'pays': 'France'}
```

`pop(cle)`

Supprime la clé indiquée et renvoie sa valeur.

```
 valeur = mon_dict.pop("ville")
 print(valeur)      # "Paris"
 print(mon_dict)    # {'nom': 'Alice', 'age': 25}
```

popitem()

Supprime et renvoie le dernier couple (clé, valeur) ajouté.

```
 print(mon_dict.popitem()) # par ex. ('ville', 'Paris')
```

clear()

Vide entièrement le dictionnaire.

```
 mon_dict.clear()
 print(mon_dict)  # {}
```

copy()

Renvoie une copie superficielle du dictionnaire.

```
 copie = mon_dict.copy()
```

- Les méthodes comme `get`, `keys`, `values`, `items`, `copy` ne modifient pas le dictionnaire original.
- Les méthodes comme `update`, `pop`, `popitem`, `clear` modifient directement le dictionnaire.

Conclusion

En Python, les types primitifs constituent la base de toute manipulation de données. Ils définissent la nature et le comportement des valeurs que l'on manipule dans un programme. Les plus fondamentaux sont `int`, `float`, `bool` et `str`.

La compréhension de ces types primitifs est indispensable, car ils déterminent la manière dont Python interprète, stocke et opère sur les données. Chaque type possède ses règles de conversion et ses opérations autorisées, ce qui permet d'écrire des programmes robustes et cohérents. En maîtrisant leur utilisation et leurs interactions, on acquiert les fondations nécessaires à la programmation structurée et orientée objet, qui viendront s'appuyer sur ces concepts fondamentaux pour construire des structures de données et des applications plus complexes.