Rappels sur les fonctions en Python

Python

1 Introduction

Les fonctions sont un moyen efficace de structurer un programme et de l'organiser en modules. Le choix du nom d'une fonction est important pour la lisibilité du programme. En Python, une fonction est définie à l'aide du mot-clé `def`.

```
def ma_fonction():
    print("Les fonctions en Python sont faciles à apprendre")

ma_fonction()
```

À noter que l'appel s'effectue par `ma_fonction()` au lieu de `print(ma_fonction())` car `ma_fonction` affiche son message avec la fonction `print` au lieu de renvoyer une valeur avec l'instruction `return`.

2 Portée locale des variables

Lorsqu'une nouvelle variable est affectée à l'intérieur du corps de la fonction, elle n'est définie qu'à l'intérieur de la fonction. La variable n'est pas visible en dehors de la fonction.

```
x = 10  # x est une variable globale
y = 10  # y aussi

def une_fonction():
    x = 5       # x est une variable locale
    global y    # permet de modifier la variable globale y
    y = 5
```

```
print(x)            # 10
print(y)            # 10
une_fonction()
print(x)            # 10 (inchangé)
print(y)            # 5  (modifié par global)
```

La variable `x` ne change pas en dehors de la fonction puisque la portée de `x` définie à l'intérieur de la fonction est limitée à la fonction elle-même. Seul le mot-clé `global` permet de réaffecter une variable globale (`y = 5`). En revanche, une liste ou un dictionnaire global peut être modifié en place sans `global`, par exemple avec `L.append(3)` ou `d[cle] = valeur`.

③ Type de paramètres

Les paramètres sont utilisés pour transmettre des informations dans une fonction.

3.1 Paramètres positionnels

Dans l'appel `fraction(10, 4)`, les valeurs sont attribuées dans l'ordre de la définition : `10` correspond à `numerateur` et `4` à `denominateur`.

```
def fraction(numerateur, denominateur):
    print(f"{numerateur}/{denominateur}")

fraction(10, 4)         # 10/4
```

3.2 Paramètres par mots-clé

On peut nommer explicitement les paramètres lors de l'appel, ce qui permet de les passer dans n'importe quel ordre :

```
fraction(numerateur=10, denominateur=4)    # 10/4
fraction(denominateur=4, numerateur=10)    # 10/4
```

3.3 Paramètres par défaut

Il est possible de préciser des valeurs par défaut pour certains paramètres. Ces valeurs sont utilisées si aucun argument n'est passé à leur place.

```
def presentation(prenom, age=15, ville="Lyon"):
    print(f"Je m'appelle {prenom}, j'ai {age} ans, "
          f"j'habite à {ville}.")

presentation("Gaudric")                    # age=15, ville=Lyon
presentation("Faustin", 19)                # ville=Lyon
presentation("Justine", 21, "Paris")       # tout explicite
presentation("Audrey", ville="Toulouse")   # age=15
```

④ Nombre variable de paramètres

4.1 La syntaxe `*args`

Complément hors programme : les syntaxes `*args` et `**kwargs` ne sont pas exigibles au baccalauréat, mais on les rencontre dans la documentation des bibliothèques.

La syntaxe `*args` permet d'accepter un nombre variable de paramètres positionnels, regroupés dans un tuple :

```
def aire_rectangle(*args):
    if len(args) == 2:
        return args[0] * args[1]
    else:
```

```
        print("Merci de stipuler deux paramètres")

print(aire_rectangle(3, 8))  # 24
```

4.2 La syntaxe `**kwargs`

La syntaxe `**kwargs` permet de recevoir un nombre variable de paramètres nommés, regroupés dans un dictionnaire :

```
def aire_rectangle2(**kwargs):
    if len(kwargs) == 2:
        result = 1
        for key, value in kwargs.items():
            result *= value
        return result
    else:
        print("Merci de stipuler deux paramètres")

print(aire_rectangle2(cote1=4, cote2=8))  # 32
```

4.3 Décomposition des arguments

On peut utiliser `*` et `**` pour décomposer un tuple, une liste ou un dictionnaire en arguments d'une fonction :

```
def somme(a, b, c):
    print(a + b + c)

ma_liste = [1, 2, 3]
somme(*ma_liste)        # 6

mon_dico = {'a': 1, 'b': 2, 'c': 3}
somme(**mon_dico)       # 6

fraction(*(10, 4))      # 10/4 (décomposition d'un tuple)
fraction(*[10, 4])      # 10/4 (décomposition d'une liste)
fraction(**{'numerateur': 10, 'denominateur': 4})  # 10/4
```

5 Fonctions comme paramètres

On peut passer une fonction comme paramètre d'une autre fonction :

```
phrase = ['Les', 'fonctions', 'Python', 'sont', 'formidables']

def appliquer_fonction(prop):
    return [prop(mot) for mot in phrase]

print(appliquer_fonction(len))
# [3, 9, 6, 4, 11]
```

On peut aussi passer une fonction définie par l'utilisateur :

```python
def voyelle_presente(mot):
    for x in mot:
        if x in 'aeiouy':
            return True
    return False

print(appliquer_fonction(voyelle_presente))
# [True, True, True, True, True]
```

⑥ La valeur `None` et le test `is None`

Une fonction qui se termine sans `return` renvoie la valeur spéciale `None`, qui signifie « aucune valeur ». On utilise aussi `None` pour indiquer qu'une variable ne désigne encore rien : par exemple, une variable `meilleur` qui ne désigne encore aucun élève avant le parcours d'une liste d'élèves.

Pour tester si une valeur vaut `None`, on écrit `x is None`, ou `x is not None`, plutôt que `x == None` : `is` vérifie qu'il s'agit exactement de l'objet `None`.

```python
def chercher(liste, valeur):
    """Renvoie l'indice de valeur dans liste, ou None si elle n'y est pas."""
    for i in range(len(liste)):
        if liste[i] == valeur:
            return i
    return None

resultat = chercher([4, 8, 15], 8)
if resultat is not None:
    print(f"Trouvé à l'indice {resultat}")   # Trouvé à l'indice 1

print(chercher([4, 8, 15], 3) is None)       # True
```

On réserve `is` à ce test. Pour comparer deux valeurs, on utilise toujours `==`.

⑦ Vérification des paramètres et documentation

7.1 Assertions

Python n'oblige pas à spécifier le type des paramètres d'une fonction. La fonction `isinstance(v, T)` renvoie `True` si la valeur `v` est de type `T`, `False` sinon (par exemple, `isinstance("a", str)` vaut `True` et `isinstance(3, str)` vaut `False`). On utilise l'instruction `assert` pour vérifier les préconditions :

```python
def est_une_voyelle(lettre):
    assert isinstance(lettre, str), "doit être une chaîne"
    assert len(lettre) == 1, "doit être un seul caractère"
    if lettre in 'aeiouy':
        return True
    return False

print(est_une_voyelle("f"))      # False
print(est_une_voyelle("poire"))  # AssertionError
```

Si l'instruction `assert` échoue, elle produit une erreur, affiche le message correspondant et la fonction arrête l'exécution.

7.2 Docstrings

Il est recommandé de documenter les fonctions avec une *docstring* :

```
def est_une_voyelle(lettre):
    """Prend en paramètre un caractère et renvoie True
    si c'est une voyelle, False sinon."""
    assert isinstance(lettre, str), "doit être une chaîne"
    assert len(lettre) == 1, "doit être un seul caractère"
    if lettre in 'aeiouy':
        return True
    return False
```

La docstring est accessible avec `help(est_une_voyelle)` ou `est_une_voyelle.__doc__`.