Chapitre 1 : Écriture d'un programme

Si écrire un programme informatique offre une certaine liberté, il n'en demeure pas moins qu'il existe certaines règles à suivre. Nous allons les détailler dans ce chapitre.

1. Instructions

1.1.Définitions

D'une manière générale, un programme est constitué d'instructions et de commentaires. Une instruction est un morceau de code minimal qui produit un effet et qui est exécuté par la machine. Un commentaire est une ligne de texte non utilisée par l'interpréteur python et donc sans effet.

1.2.Expressions et affectations

En python, les expressions et affectations sont deux instructions particulières qu'il est nécessaire de bien distinguer.

Une expression est utilisée dans le but de calculer une valeur ou pour appeler une procédure comme un test ou un changement de format. Une expression peut constituer une partie d'une ligne de code ou être saisie directement dans le shell afin d'obtenir le résultat. Une expression peut être constituée de nombres, de listes, chaînes de caractères...
Exemples :

- `n-5`
- `2+3==4`
- `int('9')`
- `[5]+[3,6]`

Une affectation permet de lier une valeur à un nom et donc de créer en mémoire une variable du même nom, ou de modifier un objet mutable déjà existant (nombre, liste, chaîne de caractère...). Une affectation comportera nécessairement un signe égal =.
Exemples :

- `res=x+3*x**2-5`
- `L=L+[6,7] ou L=L.append(6,7)`
- `phrase='Bonjour'`

1.3.Instructions simples et instructions composées

On parle d'instruction simple lorsque celle-ci s'écrit sur une seule ligne.
Exemples :

- `i=i+1`
- `print('Bonjour')`

Remarque : il est possible d'écrire plusieurs instructions simples sur une même ligne en les séparant par des points-virgules.

Mises à part les expressions et affectations qui sont des instructions simples, on peut également citer :

- Renvoyer avec `return` : renvoie le résultat d'une fonction
- Arrêter avec `break` : permet d'arrêter une boucle
- Importer avec `import` : permet l'importation de tout ou partie d'un module
- Affirmer avec `assert` : vu plus loin dans ce chapitre

On parle d'instruction composée lorsque celle-ci s'écrit sur une ligne terminée par deux points suivie d'une ou plusieurs instructions simples indentées.
Exemples :

- ```
  If x<0:
     valeur_absolue=-x
  ```
- ```
  For i in range(100):
     print(i**2)
  ```

1.4.Effet de bord d'une fonction

On dit d'une fonction qu'elle a un effet de bord si son exécution modifie quelque chose en dehors de ce qui est défini dans le corps de cette fonction (par exemple un de ses paramètres ou une variable globale définie dans le corps principal du programme).

Par exemple, la première de ces deux fonctions présente un effet de bord mais pas la seconde.

```
def ajoute1(liste,x):
     liste.append(x)

def ajoute2(liste,x):
     liste=liste+x
     return(liste)
```

En effet, dans la première fonction, la liste passée en paramètre est directement modifiée par la procédure `append`. Dans la seconde, l'affectation crée une nouvelle liste qui est une variable locale et qui laisse donc inchangée la liste passée en paramètre.
Remarque : il est à noter, comme illustré dans la seconde fonction, que deux variables, l'une locale l'autre globale, peuvent porter le même nom sans que cela pose problème.

2. Spécifications et annotations

2.1.Spécification d'une fonction

La spécification d'une fonction (ou *docstring* en anglais), notée entre triple guillemets, permet :

- d'informer sur la tâche effectuée par la fonction et le type de résultat qui peut être attendu
- de préciser les contraintes éventuelles sur les paramètres

D'une manière général, on aura ce type de syntaxe :

```
def nom_de_la_fonction(arguments):
     """informations sur la fonction non obligatoires mais
recommandées"""
     corps de la fonction
```

Cette spécification n'est pas interprétée par Python ; elle est destinée à l'utilisateur qui peut y avoir accès via la commande `help`.

Exemple : si on souhaite obtenir des informations sur la fonction nommée divmod

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

Ceci nous permet de savoir que :

- La fonction renvoie un couple
- Les deux éléments de ce couple sont `x//y` et `x%y`

- Si on note `div` et `mod` ces deux quantités, la propriété `div*y+mod==x` est invariante

2.2.Annotations et commentaires

Afin de permettre une relecture plus aisée par l'auteur d'un programme ou une personne qui découvre ce programme, il est important d'annoter certaines lignes de code ou des blocs d'instructions pour préciser leur rôle.

Pour cela, on utilise un commentaire qui est une ligne (ou une fin de ligne) précédée du symbole #. Tout comme les spécifications de fonctions, un commentaire n'est pas utilisé par l'interpréteur python.

On pourra utiliser un commentaire dans les cas suivants :
- Expliquer un choix de structure ou de méthode
- Justifier un choix (un dictionnaire plutôt qu'une liste, une fonction récursive plutôt qu'itérative,etc)

```
def mafonction(…):
    …

    # utilisation d'un dictionnaire pour stocker les données
    …

    # on teste si la clé est dans le dictionnaire
    if cle in dico:
        …

    # si la clé n'appartient pas au dictionnaire, on la crée
    else:
        dico[cle]=val
    …
```

- Dans le cas d'un programme composé de plusieurs parties, préciser l'utilité de chacune d'elles.

```
# importation de modules et de fonctions
…

# définitions de constantes et de variables
…

# définitions de fonctions annexes
…

# définition de la fonction principale
…

# exemples d'utilisation et tests
…
```

3. Assertions

Les assertions sont des instructions qui mettront fin au programme en cas de mauvaise utilisation. Une assertion est l'affirmation qu'une propriété est vraie. Dans le cas contraire, le programme s'arrête.

Une assertion se compose du mot `assert` suivi d'une propriété (test) qui correspond à un booléen. Dans le cas où le test retourne `True`, il ne se passe rien. Si le test retourne `False`, alors le programme s'arrête et le message d'erreur `AssertionError` s'affiche.

Premier exemple :
On considère la fonction inverse renvoyant l'inverse d'un nombre.

```
def inverse(x):
    """x est un nombre non nul de type int ou float
       renvoie l'inverse de x"""
    assert x!=0
    return 1/x
```

On peut alors remarquer les résultats suivants :

```
>>> inverse(-3)
-0.3333333333333333

>>> inverse(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/arnaud/Documents/MPSI/ITC/Cours/Chapitre 11/Ch1.py", line
11, in inverse
    assert x!=0
AssertionError
```

Deuxième exemple : on considère la fonction recherchant un maximum dans une liste de nombres, celle-ci devant être non vide.

```
def cherche_max(liste):
    """liste est une liste de nombres non vide
       renvoie le maximum de la liste"""
    assert liste!=[]
    n=len(liste)
    max=liste[0]
    for i in range(1,n):
        if liste[i]>max:
            max=liste[i]
    return max
```