Langage de programmation

Chapitre 2

I- Python

1) Quelques informations

Python est un langage de programmation développé à partir de 1989 par Guido van Rossum.

Ce langage de programmation est :

- entièrement gratuit,
- portable : c'est-à-dire est exécutable sur tout système d'exploitation,
- modulaire : de nombreuses librairies dédiées à des tâches précises ont été développées autour d'un noyau concis,
- populaire : vous pouvez trouver de nombreux forums pour vous aider en cas de difficultés,
- de "haut niveau" : vous pouvez programmer sans tenir compte des spécificités de votre système d'exploitation et des structures de données complexes (qui seront vues et utilisées sur les deux ans) sont disponibles,
- **interprété**. (Détails avec Mme Pelletier)

2) Environnement de travail

Il existe de nombreux environnements de travail pour programmer en Python. Vous rencontrerez deux environnements :

- Un environnement de type IDLE (Integrate DeveLopment Environment = environnement de développement intégré) (qui est utilisé en mathématiques) : EDUPYTHON.
- Un environnement qui utilise votre navigateur web : JUPYTER (il faut télécharger l'environnement qui s'appelle anaconda ou disponible sur l'ENT).

Nous utiliserons Jupyter.(quand ce sera possible !)
Le mode console se repère facilement avec les symboles >>> devant l'instruction.

II- Affectation

Remarque :

Pendant l'exécution d'un programme, les données que ce dernier manipule sont stockées en mémoire centrale. En nommant ces données, on peut les manipuler beaucoup plus facilement.
Les **variables** nous permettent donc de manipuler ces données sans avoir à se préoccuper de l'adresse explicite qu'elles occuperont effectivement en mémoire. Pour cela, il suffit de leur choisir un **nom** ou **identificateur**.

1. Les identificateurs sont des suites de lettres, de chiffres et de tirets bas (alt gr + 8), le premier caractère ne pouvant être un chiffre.
2. Attention à ne pas choisir comme identificateur des mots-clés ou de types comme for, int, ...

L'interpréteur est un programme qui traduit les lignes de code en langage Python en un langage directement compréhensible par l'ordinateur : le langage machine. L'interpréteur s'occupe d'attribuer une adresse à chaque variable et de superviser tous les éventuels futurs changements d'adresse.

Propriété 1 :

En langage Python, l'affectation sera notée à l'aide du symbole égal : **=**. On note ainsi l'affectation de 5 à x : `x = 5`

L'instruction **(en mode console)** de cette affectation s'écrit donc :

```
>>>x = 5
```

Remarques :

→ Attention ! En écrivant ceci, on n'exprime pas une égalité mais l'interpréteur exécute les actions suivantes :

- il définit un **identificateur** x comme une nouvelle **variable**,
- il réserve un **espace mémoire** pour cette variable,
- il associe à cette variable un entier dont la **valeur** est 5, x est dès lors une variable de **type** entier.

Ainsi, en python, on peut lire x=5 comme : "x est lié à un objet de type int dont la valeur est 5".

→ Même si Python est un langage non typé, on peut tout de même préciser le type directement ainsi : `x : int = 5`. De même, l'instruction

```
>>>x = x+1
```

est très fréquente en informatique en revanche, elle est inacceptable en mathématiques.

Propriété 2 :

De plus, à chaque variable correspond un **identifiant**, un numéro qui identifie une case mémoire. En Python, il est possible de connaître :

- L'identifiant d'une variable en utilisant l'instruction `id`
- Le type d'une variable en utilisant l'instruction `type`

Exercice 1 :

1) Saisir le code suivant :

```
a = 5
id(a)
```

2) Est-ce que chacun d'entre vous à le même identifiant à l'issue de l'exécution de ce code ?
3) Saisir le code suivant :

```
b = a
id(b)
```

Que remarquez-vous sur l'identifiant de b ?
4) Déterminer, grâce à l'instruction type le type de la variable b.
5) Saisir le code suivant :

```
b = "mot"
```

6) Déterminer l'identifiant et le type de la variable b. Qu'en déduisez-vous ?

A retenir :

- L'affectation se fait en Python à l'aide du symbole =.
- En Python, on copie en fait des adresses mémoire, pas des valeurs.
- En Python, on peut réaffecter une valeur d'un autre type à une variable.

Remarques :

- Sous Python, on peut affecter une valeur à plusieurs variables simultanément :

```
>>>a=b=2
>>>a
2
>>>b
2
>>>
```

- Sous Python, on peut affecter plusieurs valeurs à plusieurs variables simultanément :

```
>>>a,b = 3,0.07
>>>a
3
>>>b
0.07
>>>
```

- Dans certains langages, comme le C ou Java, il est possible de réaffecter une valeur à une variable, mais cette nouvelle valeur doit être de même type que celle initiale,
- Dans certains langages, comme le Rust (développé initialement pour Firefox), il est impossible de réaffecter une valeur à une variable.

III- Affichage et saisie

1) Affichage

Propriété 3 :

La fonction print permet d'afficher les éléments mis entre parenthèses.

Exemple 1 : Tester l'instruction suivante :

```
print("Salut tout le monde !")
```

Propriété 4 :

La fonction print peut prendre plusieurs arguments séparés par une virgule « , » cela permet ainsi d'afficher un mélange de texte et de contenu de variables.

Exemple 2 : Tester l'instruction suivante :

```
a=25
print("Cette année, Noël sera le",a,"décembre 2020 !")
```

Propriété 5 :

Transformer les espaces

Vous avez dû remarquer avec l'exemple précédent qu'à l'exécution, un espace ' ' ou " " est automatiquement ajouté entre chaque argument de la fonction print. Ce comportement peut être modifié par l'ajout d'un argument identifié par le nom sep, de type chaîne de caractères (str en Python).

Exemple 3 : Tester l'instruction suivante :

```
val=3.14159
print("Mots",'et','valeur',val,"à afficher.",sep="_")
```

Propriété 6 :

Aller à la ligne dans un print

Le chaîne de caractères \n sert à passer à la ligne lors de l'affichage.

Exemple 4 : Testez l'instruction suivante :

```
print("c'est ma première ligne \ncelle la la deuxième \net la troisième.")
```

Exercice 2 :

Réaliser l'affichage suivant en n'utilisant qu'un seul \n :

```
Mots
et
valeur
3.14159
à afficher.
```

À retenir :

- La fonction print("chaînes de caractères",'à','afficher',sep=' ') sert à afficher un ensemble de chaînes de caractères avec comme séparateur par défaut un espace.
- \n, pour newline, sert à passer à la ligne lors de l'affichage

2) Saisie

Propriété 7 :

input

La fonction input permet d'obtenir une **saisie** depuis le clavier.

Exercice 3 :

1) Tester le code suivant :
```
annee=input("Quelle année sommes-nous ?")
```
2) Tester le code suivant :
```
type(annee)
```
3) Tester le code suivant :
```
print("L'an 2050 est dans",2050-annee,"ans.")
```
Pourquoi cet affichage ?

Remarque :

Attention ! La fonction input renvoie toujours une chaîne de caractères (son type est bien str) même si un nombre (entier ou réel) a été saisi. Pour pouvoir utiliser le renvoi dans un calcul, il est parfois nécessaire de changer son typage à l'aide des fonctions int ou float qui permettent de transformer respectivement une chaîne de caractères correspondant à un nombre en nombre entier ou nombre réel (un flottant pour être précis).

Exercice 4 :

1) Tester le code suivant :

```
annee=int(input("Quelle année sommes-nous ?"))
```

2) Tester le code suivant :

```
type(annee)
```

3) Tester le code suivant :

```
print("L'an 2050 est dans",2050-annee,"ans.")
```

Pourquoi cet affichage ?

À retenir :

- input("texte") permet de saisir une information, sous forme de chaîne de caractères (=texte), pour un programme.
- Il faudra éventuellement convertir ce texte dans le type voulu avec les instructions int() ou float().

IV- Les fonctions

1) Notion

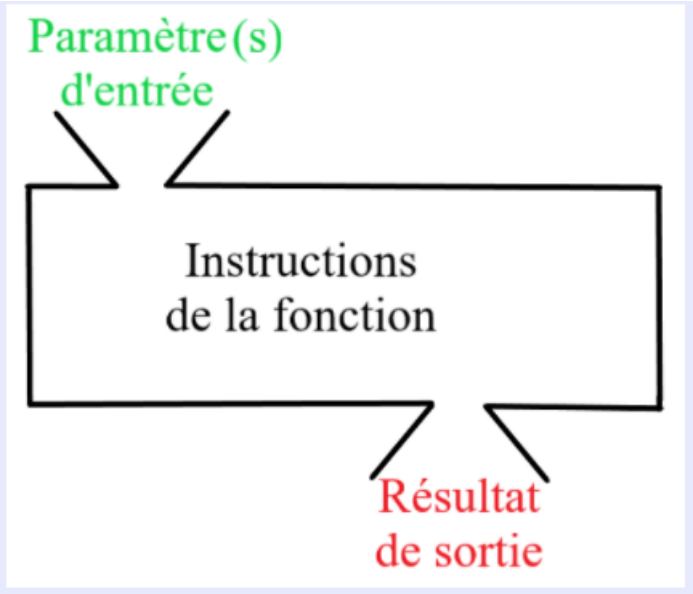
Définition 1 :

Fonctions en informatique

En informatiques, les fonctions servent à mieux structurer votre code. Par exemple, elles permettent d'éviter de répéter plusieurs fois des portions de codes identiques. Ainsi, une fonction peut être vu comme un «petit» programme :

- à qui on donne des paramètres en entrée,
- puis qui effectue alors un traitement sur ces paramètres,
- qui renvoie enfin un résultat en sortie.

Une fonction qui modifie des variables mais sans renvoyer de résultat est appelée une procédure. Le langage Python ne fait pas de différence dans la syntaxe entre fonction et procédure.



2) En python

Définition 2 :

Fonctions en Python

En Python, une fonction est définie (= être créée) en suivant toujours le même formalisme :

1. Commencer par le mot clé def,
2. Poursuivre sur la même ligne par l'entête constituée des 3 éléments successifs suivants :
    a. le nom de la fonction
    b. entre parenthèses, de 0 à N paramètres avec pour chacun un nom
    c. Terminer obligatoirement la première ligne par deux points :
3. En dessous, écrire le bloc des instructions. Attention il faut indenter (=décaler) ce bloc !
4. Finir en dernière ligne par le mot clé return suivi de ce que renvoie la fonction (ou None si la fonction ne retourne rien). Attention, cette ligne est indentée également et marque la fin de la fonction.

Voici visuellement la structure d'une fonction en Python :

```
def nomFonction(liste des arguments):
    blocs des instructions
    return résultat
```

Exemple 5 : Voilà la fonction carrée :

```
def carree(a) :
        return a**2      # renvoie l'image de a par la fonction carree
```

Remarques :

- En Python, l'exponentiation (=puissance) se note avec **. Exemple : 5**2 correspond à 5^2.
- Commentaires : le symbole # apparaîtra à maintes reprises. Il marque le début d'un commentaire que la fin de la ligne termine. Autrement dit, un commentaire est une information aidant à la compréhension du programme mais n'en faisant pas partie.
- La notion de fonction en informatique relève du même concept qu'une fonction mathématique, c'est-à-dire qu'on définit une fonction puis on l'applique à différentes valeurs.
- Vous remarquerez le symbole : très important en Python qui marque le début d'un bloc en dessous.
- C'est l'indentation qui délimite le bloc d'instructions.
- La fonction se termine avec une instruction return. Ce qui suit le return est l'image des entrées par la fonction. Dès que la fonction rencontre un return, elle renvoie ce qui suit le return et stoppe son exécution.

Propriété 8 :

Appel de fonction

Une fonction est utilisée comme une instruction quelconque. Un **appel** de fonction est constitué du nom de la fonction suivi entre parenthèses des valeurs des paramètres d'entrée. Cet appel peut être fait :

- Soit par un appel dans la console :

```
>>>carree(3)
9
```

- Soit dans le corps d'un programme :

```
def carree(a):
    return a**2          # renvoie l'image de a par la fonction carree

a = carree(3)            # a stocke la valeur 9
b = a - carree(2)        # b stocke la valeur 5
```

Définition 3 :

- Lorsqu'on définit la fonction carree(), a est appelé un **paramètre** de la fonction.
- Lorsqu'on appelle la fonction avec une valeur explicite pour a, comme dans carree(3), on dira plutôt que 3 est un **argument** de la fonction. Ainsi, en appelant la fonction carree() d'argument 3, on obtient 9.

Exercice 5 :

On veut pouvoir convertir une durée en minutes en une spécifiant le nombre d'heure et le nombre de minutes. Pour cela, il suffit de créer une fonction.

1) Définir une fonction convertir qui prend comme paramètre un entier duree_minute en renvoie un tuple (=couple ici) formé des deux valeurs nb_heures et nb_minutes.

Remarque :

Il peut vous être utile d'utiliser les deux opérateurs // et % définis ainsi :

- a // b renvoie le quotient de la division euclidienne de a par b.

Exemple 6.

`13 // 5` renvoie 2 car $13 = 5 \times 2 + 3$.

- a % b renvoie le reste de la division euclidienne de a par b.

Exemple 7.

`13 % 5` renvoie 3 car $13 = 5 \times 2 + 3$.

2) Appeler cette fonction afin de convertir en nombres d'heures 1000 minutes. Vérifier que vous obtenez comme affichage :

```
convertir(1000)
(16, 40)
```

3) Bonnes pratiques de programmation

- **Préciser le typage de chacun des paramètres**

Le langage Python est plus aisé pour démarrer la programmation pour la concision des codes écrits et pour la gestion automatique du typage par l'interpréteur. Cependant, le but est que vous puissiez à terme être capable de faire basculer vos compétences acquises en NSI sur Python vers d'autres langages de programmation.

Comme la plupart des langages de programmation nécessitent la spécification du typage des variables, à terme, on vous demandera d'écrire en Python, une fonction en précisant le type de chaque entrée et sortie en suivant le même formalisme que ci-dessous :

```
def nomFonction(liste des arguments: type) -> typeRetour:
        blocs des instructions
        return résultat
```

La convention PEP8 donne l'habitude de nommer les fonctions (comme les variables) avec des **lettres minuscules** des **soulignés bas** (touche du "8") _. Pour clarifier la fonction, il est conseillé d'utiliser un **verbe** dans son nom (obtenir, donner, get, set, ...).

- **Documenter ses fonctions**

Il est important de **documenter** vos fonctions, c'est-à-dire de décrire en quelques phrases le rôle de la fonction, de donner des informations sur la fonction, le lien entre les entrées et la sortie.

Pour cela, juste en dessous de la première ligne définissant la fonction, il suffit de mettre ses informations entre """ et """ ; c'est ce que l'on appelle en franglais le **docstring** de la fonction). En reprenant l'exemple précédent (sans le typage), on peut écrire :

```
def carree(a):
    """
    Fonction permettant de renvoyer le carré du nombre a qui est en paramètre
    """
    return a**2          # renvoie l'image de a par la fonction carree
```

L'intérêt de l'auto-documentation d'une fonction par un texte est double :

→ Pour vous : le faire vous oblige à réfléchir au contenu de votre fonction avant de la programmer ; c'est un gain d'efficacité,

→ Pour les utilisateurs de votre code (ou pour vous longtemps après avoir programmé la fonction) : Quand on saisit dans la console, après l'exécution de la fonction, l'instruction help(nom de la fonction), Python affiche le docstring de la fonction ce qui nous permet d'avoir des informations sur la fonction en cas d'oubli.

Exemple 9.

```
>>> help(carree)
Help on function carree in module __main__:

carree(a: float) -> float
    Fonction permettant de renvoyer le carré du nombre a qui est en paramètre
>>>
```

Exercice 6 :

Voici ci-dessous une fonction qui donne le prix soldé d'un article initialement à prix euros après une remise de t%.

```
def solder(prix,t):
    prix_solde = prix * (1 - t/100)
    return prix_solde
```

Améliorer le code précédent en préciser le typage de chaque paramètre et en documentant cette fonction.

4) Utilisation des modules

Nous avons déjà vues les fonctions print et input. Ce sont deux fonctions prédéfinies dans le langage Python, On appelle ce type de fonction des **fonctions natives** au langage Python , elles sont directement utilisables.

D'autres fonctions ont été développées et testées par différents programmeurs mais ne sont pas directement utilisables. Ces fonctions sont regroupées dans des **modules**, appelés aussi **bibliothèques**. En terminale, vous apprendrez même à en créer par vous-même !

Exemple 10.

- le module `math` contient de nombreuses fonctions mathématiques usuelles comme la racine carrée (`sqrt`), ...
- le module `random` contient de nombreuses fonctions modélisant le hasard comme l'obtention d'un nombre entier aléatoire compris entre deux nombres entiers a et b (`randint(a,b)`), ...
- le module `matplotlib.pyplot` permet de réaliser des graphiques, des tracés de courbes, ...

Pour pouvoir utiliser ces fonctions, il faut d'abord **importer** ces modules. Pour cela, il y a plusieurs méthodes :

- Soit importer tout le module en ajoutant **en début de programme** :

```
from nom_module import *
```

On peut alors utiliser n'importe quelle fonction du module sans prendre en utilisant seulement son nom.

Exemple 11.

```
from math import *      # import de toutes les fonctions du module math
a = (1 + sqrt(5))/2
print("valeur approchée du nombre d'or :",a)
```

- Pour éviter des confusions entre deux fonctions portant éventuellement le même nom dans deux modules différents, il peut être utile d'importer un module en lui fixant un nom d'utilisation en ajoutant **en début de programme** :

```
import nom_module as nom_raccourci
```

Exemple 12 :

```
import matplotlib.pyplot as plt    # import de toutes les fonctions du module matplotlib.pyplot mentionné plt dans toute la suite du programme
plt.grid()      # utilisation de la fonction grid (qui permet d'obtenir un quadrillage), fonction issue du module plt (=matplotlib.pyplot)
plt.show()      # utilisation de la fonction show (qui permet de réaliser le tracé du code précédent), fonction issue du module plt (=matplotlib.pyplot)
```

- Il peut suffire d'importer seulement une fonction d'un module en ajoutant **en début de programme** :

```
from nom_module import nom_fonction
```

Exemple 13.

```
from random import randint    # import de la seule fonction randint du module random
resultat_de = randint(1,6)
```

V- La portée des variables

Il faut faire la différence entre les variables utilisées dans le programme (variables globales) et les variables utilisées dans une fonction (variables locales).

Vous allez comprendre cela à l'aide des exemples de l'exercice suivant :

Exercice 7 :

Ecrire dans un jupyter les trois fonctions suivantes puis faire des appels avec différentes valeurs de x pour observer les différences entre ces codes :

Ici, ni les types ni la documentation n'ont été écrites afin de faciliter la vision de la différence entre variable locale et variable globale.

```
x=-101
def carre(x):
    x=x**2
    return x

x=-101
def carre_2():
    x=x**2
    return x

x=-101
def carre_3():
    global x
    x=x**2
    return x
```

Normalement vous avez dû détecter un problème : il y a une fonction qui ne peut pas être interprétée !

Il faut privilégier les variables locales. La première écriture, celle de carre est la définition à privilégier.

Vous pouvez utiliser des variables globales, comme dans carre3, qu'il faudra bien définir en amont.

Le danger des variables globales est qu'elles peuvent être modifiées à différents endroits d'un programme ce qui rend plus difficile la prévision du comportement du programme, dès que celui-ci devient assez conséquent.

Voici un schéma pour illustrer la différence entre variable locale et variable globale :

En résumé, pour l'instant, pas de variables globales.

Exercice 8 :

1) Voici une fonction augmenter_score qui prend comme paramètre d'entrée les points à rajouter et renvoie le nouveau score obtenu.

```python
def nouveau_score(points: int) -> int:
    """
    fonction qui renvoie le score du nombre de points donnés comme entrée
    paramètres :
        points : nombre de points à rajouter à la variable globale score (nombre entier)
    retour :
        score : nombre de points du score (nombre entier)
    """
    global score
    score = score + points
    return score
```

Combien de variables apparaissent dans la fonction nouveau_score ? De quel type ?

2) Exécuter le script suivant :

```python
score = 100
def nouveau_score(points: int) -> int:
    """
    fonction qui renvoie le score du nombre de points donnés comme entrée
    paramètres :
        points : nombre de points à rajouter à la variable globale score (nombre entier)
        retour :
        score : nombre de points du score (nombre entier)
    """
    global score
    score = score + points
    return score

for parties in range(1,4):              # 3 répétitions de l'affichage suivant
    print("J'ai gagné 10 points ! nouveau_score(10) affiche :",nouveau_score(10))
```

Est-ce que les appels nouveau_score(10) renvoie toujours la même valeur ?

3) Comme il est préférable d'éviter les variables globales, proposer une modification de la fonction nouveau_score afin de supprimer la ligne global score.

4) Tester le script complet précédent avec la fonction nouveau_score modifiée.

A Retenir :

- La structure générale d'une fonction :

```python
def nom_fonction(liste des arguments: type) -> typeRetour:
    blocs des instructions
    return résultat
```

- L'indentation permet de définir ce qui fait partie de la fonction de ce qui en est exclu.
- Une fois une fonction définie, pensez à l'appeler pour l'utiliser.
- Toujours terminer une fonction par un return qui doit être pour le mieux unique.
- Essayer de préciser les types des paramètres et du retour dans l'en-tête d'une fonction.
- Essayer de documenter ses fonctions : texte explicatif entre triples guillemets ou apostrophes.
- Essayer d'éviter l'utilisation de variables globales.

VI- Précondition et postconditions en Python

1) Précondition

Voici le code en Python d'une fonction nommée get_unite qui prend comme paramètre un nombre entier et qui renvoie son chiffre des unités.

```python
def get_unite(n: int) -> int:
    """
    renvoie le chiffre des unités d'un entier n
    """
    while n>=10 :       # répétition tant que n est supérieur ou égal à 10
        n = n-10
    return n
```

Une documentation a été donnée afin d'expliciter le bon usage de la fonction. Mais on ne peut pas être certain qu'un utilisateur de la fonction respectera les contraintes implicites ou explicites de la documentation et du typage. Voici quelques exemples d'appel de la fonction :

```
>>> get_unite(4567)
7
>>> get_unite(45.67)
5.670000000000002
>>> get_unite(-6)
-6
```

On voit que l'appel conduit à une réponse à chaque fois, mais que celle-ci ne correspond pas toujours à ce qui est attendu. Pour rendre "robuste" la fonction précédente, on doit vérifier au début de celle-ci certaines contraintes de bon usage que l'on appelle **précondition**.

Dans l'exemple précédent, ces préconditions sont :

- "Précondition 1" : n est de type entier (par exemple, le programme buggera si une chaîne de caractères est saisie come argument),
- "Précondition 2" : n est positif (sinon le résultat renvoyé est la valeur négative saisie).

Pour cela, le langage Python possède l'instruction assert qui permet un mécanisme d'**assertion**.
Les deux préconditions précédentes s'ajoutent à la fonction précédente ainsi :

```python
def get_unite(n: int) -> int:
    """
    renvoie le chiffre des unités d'un entier n
    """
    assert type(n)==int, "vous devez entrer un nombre entier."          # "Précondition 1"
    assert n>=0, "le nombre étudié doit être positif ou nul."           # "Précondition 2"
    while n>=10 :       # répétition tant que n est supérieur ou égal à 10
        n = n-10
    return n
```

Remarque :

Quelques explications :

- a==b renvoie True si l'égalité "a=b" est vraie (même type et même contenu) et False sinon,
- int est le type "entier". Il existe les types float pour les flottants, str pour les chaînes de caractères, ...

- Une telle instruction assert est suivie :
  - d'une condition (une expression booléenne qui vaut True ou False),
  - éventuellement suivie d'une virgule , et d'une phrase en langue naturelle, sous forme d'une chaine de caractères.
- L'instruction assert teste si sa condition. Deux cas possibles :
  - si la condition est satisfaite, elle ne fait rien (l'interpréteur passe à la ligne suivante)
  - sinon elle arrête immédiatement l'exécution du programme en affichant la phrase qui lui est éventuellement associée. Ainsi, l'interpréteur arrête l'exécution de la fonction plutôt que de faire planter le programme et affiche un message clair pour corriger l'erreur !

Exercice 9 :

Voici une fonction nommée diviser réalisation la division du premier argument par le second.

```
def diviser(a: float,b: float) ->float:
    """
    renvoie le résultat décimal de la division de a par b.
    """
    return a/b
```

Quelle précondition, écrite en langage Python, doit-on rajouter afin d'assurer le bon fonctionnement de la fonction ?

2) Postcondition

Souvent les fonctions sont appelées au cours de programme ; le type et la qualité du résultat renvoyé est important pour ne pas conduire à un plantage. Des contraintes sur la variable renvoyée sont souvent nécessaires : on les appelle les postconditions.

Reprenons l'exemple précédent :

```
def get_unite(n: int) -> int:
"""
renvoie le chiffre des unités d'un entier n
"""
while n>=10 :       # répétition tant que n est supérieur ou égal à 10
    n = n-10
return n
```

Voici le résultat de quelques appels effectués :

```
>>> get_unite(4567)
7
>>> get_unite(45.67)
5.670000000000002
>>> get_unite(-6)
-6
```

Comme le résultat renvoyé doit être un nombre entier compris entre 0 et 9, on va rajouter les postconditions suivantes :

1. "postcondition 1" : n est un entier naturel.
2. "postcondition 2" : n est positif.
3. "postcondition 3" : n est strictement inférieur à 10

On utilise encore l'instruction assert, juste avant le return pour écrire ces postconditions comme montré ci-dessous :

```
def get_unite(n: int) -> int:
    """
    renvoie le chiffre des unités d'un entier n
    """
    while n>=10 :       # répétition tant que n est supérieur ou égal à 10
        n = n-10
    assert type(n)==int, "Le nombre renvoyé devrait être un entier."        # "Postcondition 1"
    assert n>=0, "le nombre renvoyé doit être positif ou nul."              # "Postcondition 2"
    assert n<10, "le nombre renvoyé doit être inférieur à 10."              # "Postcondition 3"
    return n
```

Remarques :

- Ce mécanisme d'assertion est une **aide au développeur** qui permet de repérer des erreurs dans le code.
- En supprimant toutes les assertions, le programme doit à la fin toujours fonctionner.
- Normalement, Lors de la finalisation du programme les différentes assertions doivent être ôtées.

VII- Exercices

Exercice 10 :(Fonction)

L'Indice de Masse Corporelle est un nombre réel utilisé en médecine. Sa formule est pour une masse en kilos et une taille en mètres : $IMC = \frac{masse}{taille^2}$.

1. Écrire en langage Python dans Jupyter un programme qui :
    1. Demande la masse de l'utilisateur en kg (nombre réel)
    2. Demande la taille de l'utilisateur en cm (nombre entier).
    3. Contient une "fonction" de paramètres masse et taille
    4. "fonction" qui affiche l'IMC de l'utilisateur.
2. La "fonction" intégrée est-elle une fonction ou une procédure ? Pourquoi ?
3. Vérifier qu'une personne de 80 kg mesurant 1 mètre 80 a un IMC proche de 24.691358024691358.

Exercice 11 :(Fonction)

1. Écrire une fonction saisir_nom qui ne prend pas de paramètre mais renvoie le nom saisi par l'utilisateur comme chaîne de caractères.
2. Est-ce une fonction ou une procédure ? Pourquoi ?
3. Écrire une fonction nommée dire_bonjour ayant comme paramètre un mot qui affiche bonjour suivi du mot.
   Exemple : dire_bonjour("Paul") affiche "Bonjour Paul !".
4. Est-ce une fonction ou une procédure ? Pourquoi ?
5. Proposer un programme qui :
    - intègre d'abord les deux fonctions précédentes,
    - qui utilise ces deux fonctions dans le corps principal du programme

Ce programme :

- Demande à l'utilisateur son nom.
- Affiche Bonjour suivi du nom de la personne.

Exercice 12 : (Précondition)

Reprise de l'algorithme obtenu à l'exercice 4 du chapitre 1.

1. Écrire une fonction nommée get_moyenne qui renvoie la moyenne de trois nombres quelconques passés en arguments.
2. Proposer une précondition portant sur chacun des paramètres d'entrée pour assurer que la possibilité de calculer la somme.

Exercice 13 : (Fonction)

Reprise de l'algorithme obtenu à l'exercice 11 du chapitre 1.
Écrire une fonction appelée echanger qui échange les valeurs de deux arguments a et b.
Par exemple, echanger("mot",12) renvoie (12,"mot").

Remarque :

La possibilité de stocker un entier dans une variable stockant initialement une chaîne de caractères est possible en Python car c'est un langage **non typé** : l'interpréteur gère automatiquement le typage. Attention ! Dans beaucoup d'autres langages de programmation, cela n'est pas possible.

Exercice 14 : (Précondition)

1) Documenter la fonction suivante en ajoutant un docstring.

```
def examen(x,y,z,t):
    m=(2*x+2*y+z+t)/6
    if m>=10:
        print("Le candidat est reçu")
    else :
        print("le candidat est refusé")
    return None
```

2) Proposer des préconditions écrites sur les variables x, y, z et t en utilisant l'instruction assert qui assurent le bon usage de cette fonction examen.

Exercice 15 : (Fonction)

Voici une fonction cherche_lettre qui affiche si le caractère lettre est présent ou non dans le nom saisi nom, tous deux entrés comme paramètre de la fonction :

```
def cherche_lettre(nom,lettre):
    if lettre in nom:
        print(lettre," est dans le nom ",nom)
    else:
        print(lettre,"n' est pas dans le nom ",nom)
    return None
```

1. Préciser le type de chacun des paramètres d'entrée.
2. Est-ce une fonction ou une procédure ?
3. Proposer des préconditions écrites sur les variables nom et lettre en utilisant l'instruction assert qui assurent le bon usage de cette fonction cherche_lettre.

Exercice 16 :

Reprise de l'algorithme obtenu à l'exercice 10 du chapitre 1.
Écrire une fonction get_max qui prend en paramètre une liste et renvoie le max des éléments de celle-ci.
Indications :

- Une liste en python se note entre crochet [ ] chaque élément étant séparée d'une virgule ,.
- la longueur d'une liste nommée liste en Python s'obtient avec la commande len(liste).

Exercice 17 :

Réécriture de l'algorithme obtenu à l'exercice 12 du chapitre 1 sous forme d'une fonction.

1. Écrire une fonction div_euclidienne qui prend en paramètre deux nombres entiers a et b, qui effectue la division euclidienne de a par b et qui renvoie le couple (a,i), respectivement le reste et le quotient de cette division euclidienne. (Un tel couple est appelé **tuple**.)
2. Proposer des préconditions sur les paramètres a et b afin d'assurer le bon usage de la fonction div_euclidienne.
3. Proposer des postconditions sur les deux valeurs renvoyées (a et i) afin d'assurer le bon usage de la fonction div_euclidienne.

Exercice 18 :

Réécriture de l'algorithme obtenu à l'exercice 16 du chapitre 1 sous forme d'une fonction.

1. Écrire une fonction get_max_position qui prend en paramètre une liste et qui renvoie le couple (le tuple) (PG,IG), respectivement la plus grande valeur de la liste et la position de cette valeur maximale dans la liste.

Indications :

- Une liste en python se note entre crochet [ ] chaque élément étant séparée d'une virgule.
- la longueur d'une liste nommée liste en Python s'obtient avec la commande len(liste)
- Chaque élément d'une liste est positionné à l'aide d'un **index** compris entre 0 et len(liste)-1.
- L'élément d'une liste nommée liste positionné à l'index i est obtenu avec : liste[i].

2. Proposer des postconditions sur la valeur renvoyée IG afin d'assurer le bon usage de la fonction get_max_position.

VIII- Exercices de renforcements

Exercice 19. Renforcement

On veut pouvoir convertir une durée en heures en une spéficiant le nombre de jours et le nombre d'heures correspondant. Pour cela, il suffit de créer une fonction.

1. Définir une fonction `convertir_jour` qui prend comme paramètre un entier `duree_heure` en renvoie un tuple (=couple ici) formé des deux valeurs `nb_jours` et `nb_heures.`
2. Appeler cette fonction afin de convertir en nombres de jours 260 heures.

Exercice 20. Renforcement

Voici ci-dessous une fonction qui donne le prix TTC connaissant le prix Hors Taxe pour une TVA à `t` % :

```
def TTC(HT,t):
    TTC = HT * (1 + t/100)
    return TTC
```

Améliorer le code précédent en préciser le typage de chaque paramètre et en documentant cette fonction.

Exercice 21. Renforcement

Parmi les scripts suivants, le(s)quel(s) peuvent correspondre à l'utilisation de la fonction inverse pour calculer l'image de 4 :

```
# script 1 :
x = 4
def fct1(x):
    x = 1/x
    return x

# script 2 :
x = 4
def fct2():
    x = 1/x
    return x

# script 3 :
def fct3(x):
    return 1/x

# script 4 :
x = 4
def fct4():
    global x
    x = 1/x
    return x
```

Exercice 22. Renforcement

Un moyen d'estimer la profondeur d'un puits est de lâcher une pierre au dessus du puits et de compter le nombre de secondes avant d'entendre le bruit du "plouf" de son entrée dans l'eau. On note t la durée d'attente du "plouf" (en seconde) et p le profondeur du puits (en mètre).

On admet que les paramètres t et p sont liés par la relation physique suivante : $t = \sqrt{\frac{p}{4.9}} + \frac{p}{330}$.

1. **Première partie : une fois la notion précondition vue**

   Proposer une fonction `temps` qui prend en paramètre la profondeur p du puits et renvoie le temps t d'attente du "plouf".

   Remarque(s) :

   Penser à importer la fonction `sqrt` du module `math`.

2. Vérifier que l'on obtient l'affichage suivant pour l'exécution du script suivant :

   ```
   >>>temps(30)
   2.5652673874360583
   ```

3. Rajouter à votre programme deux préconditions sur p.

4. **Seconde partie : une fois la notion postcondition vue**

   Rajouter à votre programme une postcondition sur le valeur renvoyée.

5. Rajouter une documentation à la fonction `temps`.

Exercice 23. Renforcement

L'énergie cinétique d'un objet de masse m (en kg) qui se déplace à la vitesse v (en $m.s^{-1}$) est donnée par la formule : $E_c = \frac{1}{2} \times m \times v^2$.

1. Écrire une fonction nommée `get_energie_cinetique` ayant comme paramètres la masse `m` et la vitesse `v` qui renvoie l'énergie cinétique.
2. Est-ce une fonction ou une procédure ? Pourquoi ?
3. Quelle est l'énergie (en Joule) d'un objet de 3 kg se déplaçant à 1.56 $m.s^{-1}$?

Exercice 24. Renforcement

1. Écrire une fonction `saisir_num_tel` qui ne prend pas de paramètre mais renvoie le numéro de téléphone saisi par l'utilisateur comme chaîne de caractères.
2. Est-ce une fonction ou une procédure ? Pourquoi ?
3. Écrire une fonction nommée `prevenir_message` ayant comme paramètre un numéro de téléphone qui affiche "Vous allez recevoir un message de confimation au ....." où apparaît le numéro de téléphone saisi.
   Exemple : `prevenir_message(0665432100)` affiche `"Vous allez recevoir un message de confimation au 0665432100."`.
4. Est-ce une fonction ou une procédure ? Pourquoi ?
5. Intégrer ces deux fonctions pour faire apparaître un programme qui :
   - Demande à l'utilisateur son numéro de téléphone.
   - Affiche un message selon lequel un message de confirmation va être envoyé au numéro de la personne.

Exercice 25. Renforcement

Reprise de l'algorithme obtenu à l'exercice 18 du chapitre

1. Écrire une fonction nommée `get_moyenne5` qui renvoie la moyenne de cinq nombres quelconques passés en arguments.
2. Proposer une précondition portant sur chacun des paramètres d'entrée pour assurer que la possibilité de calculer la somme.

Exercice 26. Renforcement

Reprise de l'algorithme obtenu à l'exercice 6 du chapitre

Voici une écriture possible en langage Python de l'algorithme de cet exercice 6 sous forme de fonction :

```
def exo6(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s
```

1. Documenter la fonction précédente en ajoutant un docstring et préciser le typage des paramètres.
2. Proposer des préconditions écrites sur la variable `n` en utilisant l'instruction `assert` qui assurent le bon usage de cette fonction `exo6`.
3. Proposer un postcondition écrite sur la variable renvoyée.

Exercice 27. Renforcement

Reprise de l'algorithme obtenu à l'exercice 15 du chapitre

Voici une écriture possible en langage Python de l'algorithme de cet exercice 15 sous forme de fonction :

```
def exo15(phrase,lettre):
    fin = ""
    for elt in phrase:
        if elt != lettre:
            fin = fin + elt
    return fin
```

1. Documenter la fonction précédente en ajoutant un docstring et préciser le typage de chaque paramètre.
2. Proposer des préconditions écrites sur les variables `phrase` et `lettre` en utilisant l'instruction `assert` qui assurent le bon usage de cette fonction `exo15`.
3. Proposer un postcondition écrite sur la variable renvoyée.