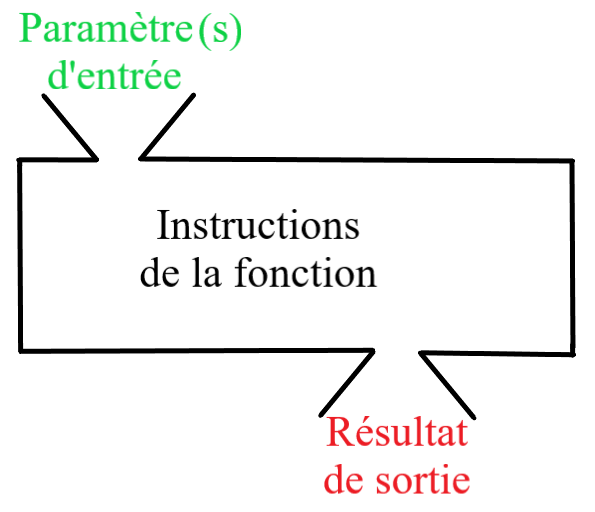
|  |
| --- |
| Capacités attendues |
| * Prototyper une fonction. * Décrire les préconditions sur les arguments. * Décrire des postconditions sur les résultats. * Utiliser des jeux de test. |

# Introduction

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 vue 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.



# Les fonctions en python

En Python, une fonction peut s'écrire 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 :
   1. le nom de la fonction
   2. entre parenthèses, de 0 à N paramètres avec pour chacun un nom
3. Terminer obligatoirement la première ligne par deux points :
4. En dessous, écrire le bloc des instructions. Attention il faut **indenter** (=décaler) ce bloc.
5. 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

Voici la fonction carrée :

def carree(a: float) -> float:

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 5²
* 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.
* 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

* 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.

###### Vocabulaire :

* Lorsqu'on définit la fonction carree(), a est appelé **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.

# 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 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 et des tirets bas (celui 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.

>>> 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

# 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 suivants :

Ecrire dans un fichier python 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.

def carre(x):

x=x\*\*2

return x

def carre\_2():

x=x\*\*2

return x

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,dans des cas qu'il faudra bien définir en amont.

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

# Précondition et postconditions en Python

## 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.

def get\_unite(n: int) -> int:

"""

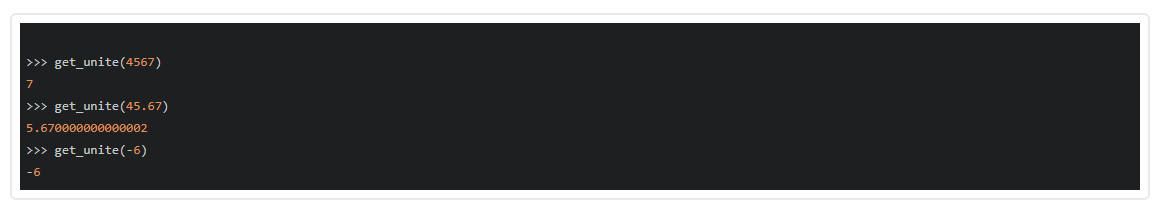
renvoie le chiffre des unités d'un entier n

"""

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 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 :

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 2" : 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 :

def get\_unite(n: int) -> int:

"""

renvoie le chiffre des unités d'un entier n

"""

# "Précondition 1"

assert type(n)==int, "vous devez entrer un nombre entier."

# "Précondition 2"

assert n>=0, "le nombre étudié doit être positif ou nul."

while n>=10 : # répétition tant que n est supérieur ou égal à 10

n = n-10

return n

Remarques :

* 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 :
  1. d'une condition (une expression booléenne qui vaut True ou False),
  2. 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 :
  1. si la condition est satisfaite, elle ne fait rien (l'interpréteur passe à la ligne suivante)
  2. 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 !

## 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 :



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

# "Postcondition 1"

assert type(n)==int, "Le nombre renvoyé devrait être un entier."

# "Postcondition 2"

assert n>=0, "le nombre renvoyé doit être positif ou nul."

# "Postcondition 3"

assert n<10, "le nombre renvoyé doit être inférieur à 10."

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.

# Jeux de tests

Une fois une fonction créée, il faut impérativement vérifier sa validité. C’est-à-dire qu’elle effectue bien ce pour quoi elle a été conçue.

Pour assurer cette validation, mettre en œuvre un jeu de tests sous la forme d’une suite d’appels de la fonction définie. Ces tests doivent envisager suffisamment de cas afin que le concepteur ait confiance dans la validité de sa fonction. L’écriture du jeu de tests se compose d’une suite d’assertions.

def ajoute\_1(x)

return x + 1

#Jeu de tests

assert ajoute\_1(2) == 3

assert ajoute\_1(-2) == - 1

L’exécution de ces lignes ne renvoie rien car les tests effectués sont vrais. En revanche si on écrit un test qui est faux ( comme assert ajoute\_1(2) == -2), on obtient :

assert ajoute\_1(2) == -2

AssertionError