

Deuxième partie

Fonction

1 Introduction : programmes et fonctions

Deux points de vue pour définir un programme :

- c'est une suite d'instructions ;
- c'est une fonction (mathématique).

Les deux sont utiles et intéressants et conduisent à des styles de programmation différents.

Le premier donne une programmation **impérative**, le second une programmation **fonctionnelle**.

Pour Python, c'est le premier point de vue.

Comment écrire un programme dans ce cas ?

Création dans l'éditeur d'un programme calculant la surface d'un triangle à partir de sa base et de sa hauteur (exemple1.py)

```
b = 11
h = 5
surface = b*h/2
print (surface)
```

Lancer l'exécution avec **Run** puis avec **python3 exemple1**

En programmation impérative, un programme est donc une suite d'instructions (souvent des affectations) exécutées séquentiellement.

Cette suite pouvant être longue, pour en faciliter la conception et la maintenance, il est fondamental de pouvoir décomposer un problème en sous-problèmes.

Ceux-ci (qui pourront aussi être décomposés !), seront plus faciles à écrire, tester, maintenir et pourront aussi être réutilisés pour d'autres problèmes.

2 Écriture de fonctions avec Python

2.1 Fonction sans paramètre

2.1.1 Spécification de fonction

C'est le contrat entre le concepteur et l'utilisateur : il est **fondamental** de l'écrire

Elle donne cinq informations :

- le nom de la fonction ;
- le type des données reçues (les paramètres) ;
- le type du résultat produit ;
- le nom des paramètres ;
- la description de l'action réalisée.

Pour les types, nous utiliserons (dans un premier temps) :

vide : absence de valeur ;

booléen : vrai ou faux

naturel : entier ≥ 0 ;

entier : entier relatif

réel : flottant

nombre : réel \cup entier ;

chaîne : chaîne de caractères ;

indifférent : n'importe quel élément.

Soit :

```
aireTriangle5_11 : vide → vide
                  → affiche l'aire d'un triangle de base 11 cm
                     et de hauteur 5 cm
```

2.1.2 Définition de fonction en Python

On utilise une instruction composée avec le mot-clé `def`.

Instruction composée :

```
~~~~entête :

~~~~<indentation> instruction 1

~~~~....

~~~~<indentation> instruction n
```

Dans le cas d'une fonction, cela donne :

```
~~~def nomfonction (liste paramètres) :

~~~~~(entête de la fonction)

~~~~~ instructions de la fonction

~~~~~(corps de la fonction)
```

Dans le corps de la fonction vous pouvez trouver toutes les instructions que vous autorise Python.

Portable : saisie (sans commentaire!) de la fonction `aireTriangle5_11` dans l'éditeur (faisable dans shell, mais sans doute pas judicieux en L1).

Mais, comme cela, ce n'est pas exploitable : il faut des commentaires.

Toute programmation doit être accompagnée de commentaires

— avant la fonction, reprendre sa spécification ;

— la description peut (et **doit**) être donnée dans une chaîne de caractères sur la première ligne

Cela vous sera demandé en TP.

Définition de la même fonction dans un fichier (exemple2).

Bien insisté sur la présentation proposée, à faire systématiquement.

2.1.3 Appel de fonction

On donne le nom de la fonction suivi de `()`.

```
aireTriangle11_5(). Portable dans le shell
```

2.1.4 Variable locale / variable globale

Dans la fonction créée, il y a trois variables : **b**, **h** et **surface**.

Elles sont définies dans la fonction et sont inaccessibles (inexistantes) en dehors : ce sont des variables **locales**.

S'il existe une variable globale de même nom qu'une variable locale, elle ne sera pas modifiée par l'appel de la fonction.

Attention : l'utilisation de variable globale dans une fonction rend plus difficile la mise au point : **à éviter!**

Faire des tests avec une var globale `b` dans le shell, en montrant qu'elle n'est pas modifiée par un appel de la fonction (en fonction du temps ...)

2.2 Fonction avec paramètres

Pourquoi ne calculer que la surface d'un triangle de base 11 et de hauteur 5 ?

En considérant **b** et **h** comme des paramètres formels, et avec la même formule, nous aurons une fonction capable d'afficher la surface de n'importe quel triangle (dont on donnera la base et la hauteur).

Spécification :

aireTriangle : *nombre* \times *nombre* \rightarrow *vide*
 \rightarrow affiche l'aire d'un triangle de base **b** cm
et de hauteur **h** cm

Mise en œuvre et utilisation. Sur le portable, exemple3 plus appel **airetriangle(6,3)**

2.3 Paramètres formels / paramètres effectifs

Dans la définition de **aireTriangle**, **b** et **h** sont appelés des paramètres formels.

Leur signification est intuitivement la suivante : soient **b** et **h** deux nombres auxquels on donnera une valeur plus tard, alors, la fonction **aireTriangle** que nous venons de définir rendra le résultat fourni par l'évaluation du corps de cette fonction, où **b** et **h** auront d'abord été remplacés par les valeurs données.

Dans l'appel, **aireTriangle(6,3)**, 6 et 3 sont appelés les paramètres effectifs de l'appel de fonction.

3 Définition d'une « vraie » fonction

Une fois compilée la fonction est utilisable comme n'importe quelle fonction offerte par le logiciel.

Cependant, ce que nous venons d'écrire ne produit pas une « vraie » fonction, mais plus exactement une **procédure**, car elle ne rend pas de résultat (type *vide* en résultat).

Par exemple, elle ne peut être utilisée par une autre fonction pour un calcul plus complexe.

Pour rendre un résultat, il faut utiliser le mot clé **return**

Spécification 1.

surfaceTriangle : *nombre* \times *nombre* \rightarrow *réel*
b, h \rightarrow *surface du triangle de base b*
et de hauteur h

Programmation 1.

```
def surfaceTriangle(b,h):  
    "Rend la surface du triangle de base b et de hauteur h"  
    return b * h / 2.
```

Faire la définition sur le portable (exemple4).

Nous pouvons maintenant utiliser cette fonction pour réaliser un autre calcul ou une autre fonction.

Par exemple, pour calculer l'aire d'un polygone régulier à **n** cotés, chaque côté étant de taille **c**

Propriété à utiliser : un polygone à **n** pans de côté **c** cm peut se découper en **n** triangles isocèles de base **c** et de hauteur $\frac{c}{2 \times \tan(\frac{\pi}{n})}$

Spécification 2.

surfacePolygone : *naturel* \times *nombre* \rightarrow *réel*
n, c \rightarrow *surface d'un polygone à n cotés (n>2)*
chaque côté faisant c cm

Programmation 2.

```
def surfacePolygone(n,c):  
    "Rend la surface d'un polygone à n cotés (n>2) chaque coté faisant c cm"  
    return n*surfaceTriangle(c,c/(2*tan(pi/n)))  
Mais tan et pi sont définis dans une bibliothèque !
```

Utilisation d'une bibliothèque (ou module) :

```
import math
```

Portable : exemple5

4 Test d'une fonction

La phase de test est un point important de la programmation :
vous devez vérifier que le code produit répond bien à la **spécification**, d'où l'importance de la donner avant la programmation !

Les tests doivent assurer une bonne couverture des cas possibles. Une bonne stratégie de test est définie avant la programmation.

5 Typage des paramètres

Le type d'un paramètre est celui du paramètre effectif donné au moment de l'appel.

Si le paramètre effectif est d'un type incorrect (ne correspondant pas à la spécification), le résultat est non garanti !

La spécification donne le contrat d'utilisation.

Faire un exemple avec une chaîne en paramètre

Si nous devons sécuriser l'appel, nous ne pouvons que faire un contrôle à posteriori (non utilisé dans le cadre de ce cours) !