

FONCTIONS

I. Notion de fonction

1) Introduction

Le programme suivant calcule $\sum_{k=0}^n \binom{n}{k}^2$, avec n un entier donné.

```
somme = 0
for k in range(n+1):
    #Calcul de n!
    facto_n = 1
    for i in range(1,n+1):
        facto_n = facto_n * i
    #Calcul de k!
    facto_k = 1
    for i in range(1,k+1):
        facto_k = facto_k * i
    #Calcul de (n-k)!
    facto_n_k = 1
    for i in range(1,n-k+1):
        facto_n_k = facto_n_k * i
    somme = somme + (facto_n/(facto_k*facto_n_k))**2
```

Ce type de programmation présente des inconvénients :

- On réécrit plusieurs fois des portions de code très semblables : ici, on calcule trois factorielles. Le programme est long, difficile à modifier (de nombreuses modifications à apporter si on veut modifier la manière de calculer les factorielles... ou corriger une erreur!).
- Le programme est peu lisible (commentaires nécessaires).
- Le programme utilise des variables auxiliaires qui étaient peut-être déjà affectées.

Pour éviter cela, on va utiliser des fonctions, chaque fonction exécutant une tâche « élémentaire ».

Une fonction :

- est écrite une seule fois et exécutée autant de fois qu'on le veut (on évite des redondances),
- peut prendre des arguments (entrées/paramètres),
- exécute une suite d'instructions,
- retourne éventuellement un résultat,
- peut faire appel à d'autres fonctions.

Les fonctions vont nous permettre de construire des programmes complexes, en les décomposant en problèmes de plus en plus simples (on parle d'*analyse descendante*), chaque problème élémentaire sera résolu par une fonction.

2) Syntaxe et exemple

En Python, la syntaxe pour définir une fonction est la suivante :

```
def <nom de la fonction>(<paramètre 1>, <paramètre 2>,...):
    <corps de la fonction>
```

Le corps de la fonction peut contenir des instructions **return**; une instruction **return** interrompt immédiatement la fonction en renvoyant éventuellement une valeur de retour.

C'est l'indentation qui définit le corps de la fonction; le retour au niveau d'indentation du mot-clé **def** marque la fin de la fonction.

On prendra soin de choisir des noms de fonctions explicites.

Exemple

Écrivons un autre programme calculant $\sum_{k=0}^n \binom{n}{k}^2$, avec cette fois-ci une fonction de calcul de factorielle et une fonction de calcul de coefficient binomial :

```
def facto(n):
    res = 1
    for i in range(1,n+1):
        res = res * i
    return res

def binom(n,p):
    return facto(n) // (facto(p)*facto(n-p))

somme = 0
for k in range(n+1):
    somme = somme + binom(n,k)**2
```

Ce nouveau programme :

- est plus clair (le rôle de chaque fonction est facilement identifiable),
- est plus facilement modifiable (pas de redondances),
- utilise peu de variables.

3) print vs return

Une erreur fréquente consiste à utiliser `print` au lieu de `return`.

- L'instruction `return` sert à définir la valeur renvoyée par une fonction lorsque celle-ci sera appelée par le programme; cette valeur pourra ensuite être utilisée dans le programme.
- `print` sert à afficher des informations sous forme de texte dans la fenêtre d'exécution, pour que celles-ci soient lues par un utilisateur humain. Autrement dit :

« `print` is for people »

Il est néanmoins parfois utile d'écrire des fonctions d'affichage, qui ne renverront pas de résultat mais qui produiront un affichage à destination de l'utilisateur.

Exemple

```
#importation du module de gestion des dates
from datetime import *
def affiche_date():
    print("Nous sommes le : ",date.today())

#Exemple d'appel:
affiche_date()
```

II. Portée des variables

1) Variables locales

Une fonction a souvent besoin de variables pour s'exécuter. Un appel à la fonction ne doit pas modifier des variables extérieures à celle-ci (en tout cas sans que cela soit le but de la fonction...). Par défaut, les variables utilisées par une fonction sont donc *locales*, ce qui signifie qu'on peut changer leur valeur à l'intérieur de la fonction, sans changer la valeur d'une variable extérieure homonyme.

Exemple

```
def f(x):
    y = 3
    print("dans la fonction, x =", x, " et y =", y)

x = 5
y = 4
print("avant l'appel de la fonction, x =", x, " et y =", y)
f(17)
print("apres l'appel de la fonction, x =", x, " et y =", y)
```

Ce programme va afficher :

avant l'appel de la fonction, $x = 5$ et $y = 4$
 dans la fonction, $x = 17$ et $y = 3$
 après l'appel de la fonction, $x = 5$ et $y = 4$

Une variable locale n'est pas vue de l'extérieur de la fonction.

Par exemple, le programme suivant :

```
def f(x) :
    y = 5
    return x + y

print(f(3))
print(y)
```

va produire l'erreur

NameError: name 'y' is not defined

2) Variables globales

Si une fonction utilise une variable sans jamais l'affecter, alors la variable a la même valeur dans la fonction qu'à l'extérieur de la fonction :

Exemple

```
def f(x) :
    return x + y

y = 1789
print(f(3))

y = 1802
print(f(3))
```

affichera

1792
1805

⚠ ATTENTION ! Comme le montre l'exemple précédent, lorsqu'une variable globale est utilisée dans une fonction, c'est la valeur de la variable au moment de l'appel qui est utilisée, et non celle au moment de la définition de la fonction (on parle de *liaison dynamique*).

Tout est fait pour qu'une fonction ne puisse pas modifier une variable globale, ce qui est la conduite prudente. Il est néanmoins possible de modifier une variable globale à l'intérieur d'une fonction, à l'aide de l'instruction `global`, mais cela est extrêmement déconseillé.

Exemple

```
def f(x) :
    global y
    z = 100
    y = x * y
    print("A la fin de la fonction : x =", x, "; y =", y, "; z = ", z)

x, y, z = 1, 2, 3
print("Avant l'appel : x =", x, "; y =", y, "; z = ", z)
f(5)
print("Après l'appel : x =", x, "; y =", y, "; z = ", z)
```

affiche :

Avant l'appel : $x = 1$; $y = 2$; $z = 3$
 A la fin de la fonction : $x = 5$; $y = 10$; $z = 100$
 Après l'appel : $x = 1$; $y = 10$; $z = 3$

III. Pour aller plus loin...

Les notions abordées dans cette partie ne sont pas au programme, mais peuvent se révéler utiles.

1) Manipuler des fonctions

En Python, une fonction est une valeur comme une autre : elle peut être stockée dans une variable, passée en argument, ou encore renvoyée comme résultat. Dans l'exemple suivant, la fonction `somme_fonction` prend en

argument une fonction f et un entier n et renvoie $\sum_{k=0}^n f(k)$:

```
def somme_fonction(f,n):
    s = 0
    for k in range(n+1):
        s = s + f(k)
    return s

def carre(x):
    return x**2

print(somme_fonction(carre,10))
```

2) Fonctions anonymes

On peut définir des fonctions sans nom : elle doivent être suffisamment simples dans le sens où elles doivent rendre leur résultat directement, sans calcul intermédiaire et sans utiliser de variables auxiliaires. On utilise le mot clé `lambda`

```
lambda x : x*x
# la fonction carre

f = lambda x : x**3
#equivalent a def f(x) : return x**3

(lambda x : 2*x+3)(10)
#pour appliquer une fonction anonyme
```

3) Fonction comme valeur de retour

La valeur de retour d'une fonction peut être une fonction :

```
def compose(f,g):
    h = lambda x : f(g(x))
    return h

def carre(x):
    return x**2

p4 = compose(carre,carre)
```

La variable `p4` désigne alors une fonction.

Exercices

EXERCICE 1

Écrire une fonction prenant en argument un entier n et renvoyant l'entier 2^n .

On n'utilisera pas l'opérateur `**`.

EXERCICE 2

Écrire une fonction prenant en argument trois entiers a , b , c et renvoyant le plus grand des trois entiers; on n'utilisera pas la fonction `max`.

EXERCICE 3

La fonction `random.random()` renvoie un flottant entre 0 inclus et 1 exclu. Écrire une fonction qui renvoie la valeur booléenne `True` ou `False` de manière équiprobable.

EXERCICE 4

1. Écrire une fonction qui prend en entrée un entier `a` et qui renvoie `True` s'il correspond à une année bissextile, `False` sinon.
2. Écrire une fonction qui prend en entrée trois entiers `j,m,a` et qui renvoie `True` si la date représentée par le triplet (j, m, a) (jour, mois, année) est valide, `False` sinon.