Fonctions et procédures

Skander Zannad et Judicaël Courant

2013-09-21

1 Introduction

On se donne n et k et on veut calculer $b = \binom{k}{n}$.
On utilise l'égalité

$$\binom{k}{n} = \frac{n!}{k!(n-k)!}$$

On sait déjà calculer $n!$:

```
fact_n = 1
for i in range(1, n):
  fact_n *= i
```

Donc on peut faire un copier-coller et faire ça pour k et $n-k$ et ça donne le programme suivant :

```
fact_n = 1
for i in range(1, n):
  fact_n *= i
fact_k = 1
for i in range(1, k):
  fact_n *= i
fact_n_moins_k = 1
for i in range(1, n-k):
  factn_moins_k *= i
b = fact_n / (fact_k * fact_n_moins_k)
```

Facile mais cinq bugs à corriger quand même ! (lesquels ?)
Dont quatre liés au copier-coller...
Les copier-coller :

- Dupliquent les bugs.
- En créent de nouveaux.

Bref, les copier-coller **saymal**.
DRY ! (again)

2 Fonctions

Et si on expliquait à python ce qu'est la fonction factorielle ?
Ensuite, il suffirait d'écrire :

```
b = fact(n) / (fact(k) * fact(n-k))
```

Définir une fonction :

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

3 Variables locales

En mathématiques : notion de variable liée.
En informatique : notion de variable locale à une fonction (invisible de l'extérieur de la fonction).
Principe en Python : dans une fonction, les variables sont *locales*.
Ici :

```
>>> fact(10)
3628800
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
>>> f
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
```

Les variables locales n'interfèrent pas avec les variables globales, ce sont des variables différentes, même si elles portent le même nom.

```
>>> n = 25
>>> fact(10)
3628800
>>> n
25
```

4 Variables globales

Première exception au principe de localité des variables :

```
pi = 3.14159265
def aire_disque(r):
  return pi * r**2

def perimetre_cercle(r):
  return 2 * pi * r
```

Variable `pi` *globale*.

Une variable qui n'est pas affectée dans une fonction est (implicitement) considérée comme globale.

Deuxième exception : *Une variable peut être explicitement déclarée comme globale.* Cela permet alors de la modifier. Mais en général saymal.

Quizz : que se passe t-il lors de l'exécution du programme suivant ?

```
pi = 3.14159265
def aire_disque(r):
  a = pi * r**2
  pi = 3
  return a
aire_disque(10)
```

5 Drôles de fonctions

(en Python3)

```
>>> x = print('hello') # que calcule print ?
hello
>>> # un affichage après une affectation ?
... # mais qu'y a t-il dans x ?
... print(x)
None
```

Contrairement à une fonction mathématique

- **print** ne retourne pas de résultat (plus exactement, elle retourne un objet bidon appelé `None`).
- **print** a un *effet de bord*.

Effet de bord :

- Traduction littérale de *side effects* (effet secondaire)
- Définition : *modification de l'état ou interaction observable avec le monde extérieur, autre que le fait de retourner une valeur.*
- Exemples :
  - Modification d'une variable extérieure à la fonction.
  - Écriture sur le disque dur, envoi d'un paquet réseau.

  - Lecture du disque dur, réception d'un paquet réseau.

Une fonction sans effet de bord sera dite *pure*.

6 Procédures et fonctions

Les effets de bords sont généralement :

- Gênants lorsqu'on veut utiliser une fonction comme une fonction mathématique (pure).
- Pratiques pour réaliser une procédure (suite d'actions à réaliser sur le monde extérieur).

Pour programmer proprement, les fonctions qu'on écrira seront :

- Soit des fonctions pures (sans effet de bord et calculant un résultat).
- Soit des procédures (fonctions à effet de bord ne retournant aucun résultat).

7 Procédure ou fonction ?

Besoin : «calculer la somme des entiers de 0 à 1000 inclus»
Vaut-il mieux écrire

- Une procédure affichant la somme ?
- Une fonction pure retournant la somme ?

Version procédure :

```
def affiche_somme(liste):
    s = 0
    for x in liste:
      s += x
    print(s)
affiche_somme(range(1001))
```

NB : pas de return a le même effet que « return None »

Version fonction pure :

```
def somme(liste):
    s = 0
    for x in liste:
      s += x
    return(s)
print(somme(range(1001)))
```

Les deux solutions conviennent.
On demande maintenant d'afficher aussi le carré de la somme des entiers de 0 à 100.
Version fonction pure :

```
print(somme(range(1001)))
print(somme(range(101))**2)
```

Version procédure : ???
Intérêt des fonctions pures : on peut en réutiliser le résultat.