Python : les sous-programmes

CNAM UTC503 – Paradigmes de programmation

Xavier Crégut <prenom.nom@enseeiht.fr>

Sommaire

Motivation

Objectif

Le but des sous-programmes est de permettre au programmeur de définir ses propres **instructions (procédures)** ou ses propres **expressions (fonctions)** sous le forme de **sous-programmes**.

Procédure

Une **procédure** est un sous-programme qui n'a pas de résultat.
Exemple : print

Fonction

Une **fonction** est un sous-programme qui retourne un résultat.
Exemple : abs, randint, etc.

Remarque

En Python **tout est fonction** : Une procédure est une fonction qui renvoie **None**.

Exemple de définition de fonction : la fonction pgcd

pgcd.py

```python
'''Exemple de sous-programme (fonction). '''

def pgcd(a, b):
    '''Le pgcd, plus grand commun diviseur, de a et b, entiers naturels.

    Les entiers a et b doivent être strictement positifs.
    Cette version est naïve et donc peu efficace.

    :param a, b: deux entiers strictement positifs.
    :returns: le pgcd de a et b.
    :raises: ValueError si a ou b négatif ou nul.

    >>> pgcd(21, 15)
    3
    '''
    if a <= 0 or b <= 0:
        raise ValueError(f'a et b doivent être > 0 : a={a} et b={b}')
    while a != b:
        # soustraire au plus grand le plus petit
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Explications

La **spécification** d'une fonction est composée de :

- une **signature** : la ligne qui contient `def`, le nom de la fonction, ses paramètres formels
- une **sémantique** : une chaîne de caractères (docstring) qui décrit l'objectif de la fonction
- signature : suffisante pour l'interpréteur Python ; sémantique : nécessaire pour l'utilisateur

La **signature** respecte la syntaxe suivante : `def nom_fonction(paramètre1, ...):`

- où apparaissent le nom de la fonction et de ses paramètres formels
- les « deux-points » ouvrent sur un bloc (les instructions de la fonction)
- attention à indenter les lignes qui suivent ces « deux-points » !
- la première ligne est la chaîne de documentation (docstring) pour la sémantique

La **sémantique** (docstring) est normalisée (et accessible via help() de Python) :

- voir PEP 257 et différents styles
- une phrase pour décrire l'objectif, suivie d'une ligne blanche
- une description plus détaillée (conditions d'utilisation, effets)
- une description des paramètres, du résultat et des éventuelles exceptions

L'**implantation** ou **corps** qui suit la chaîne de documentation (docstring)

- ce sont les instructions déjà vues !
- elles sont exécutées quand la fonction est appelée
- `return` arrête l'exécution de la fonction et indique la valeur retournée
- `raise` signale que la fonction ne peut pas réaliser le traitement attendu (voir exceptions)

En général, on définit plusieurs sous-programmes dans le même fichier (voir Modules)

Utilisation de la fonction pgcd

exemple_pgcd.py

```
1  '''Exemple utilisant le pgcd'''
2
3  from pgcd import pgcd
4
5  def main():
6    '''Afficher le pgcd de 2 entiers
7    lus au clavier. Non robuste.'''
8    # demander deux entiers a et b
9    a = int(input('a = '))
10   b = int(input('b = '))
11
12   # calculer le pgcd de a et b
13   p = pgcd(a, b)
14
15   # afficher le pgcd
16   print('pgcd =', p)
17
18 main()
```

Cas nominal

```
$ python exemple_pgcd.py
a = 15
b = 20
pgcd = 5
```

Cas d'erreur :

```
$ python exemple_pgcd.py
a = 15
b = 0
Traceback (most recent call last):
  File "exemple_pgcd.py", line 18, in <module>
    main()
  File "exemple_pgcd.py", line 13, in main
    print('pgcd =', pgcd(a, b))
  File "pgcd.py", line 19, in pgcd
    raise ValueError(f'a et b doivent être > 0 : a={a} et b={b}')
ValueError: a et b doivent être > 0 : a=15 et b=0
```

Explications

- utilisation de `import` car pgcd est défini dans un autre fichier (voir Modules)
- cas nominal : `return` termine la fonction et l'exécution continue à ligne 13
- cas d'erreur : `raise` termine la fonction et provoque l'arrêt du programme (voir exceptions)

Variété des paramètres

Considérons les appels suivants :

```
len("bonjour")      # 1 paramètre positionnel
print(421)          # 1 paramètre positionnel
print('n =', n)     # mais il peut y en avoir un nombre quelconque
print(a, b, sep=' ; ', end='.\n')   # 2 paramètes positionnels
            # et deux paramètres nommés (end et sep)
```

Question : Plusieurs sous-programmes nommés `print` ?
Non, pas de surchage en Python. Il n'y qu'un seul `print` !

Vocabulaire :

- paramètre formel (parameter) : lors de la définition du sous-programme
  - exemple : def len(obj): ==> obj est un paramètre formel
- paramètre effectif ou réel (argument) : lors de l'appel du sous-programme
  - exemple : len("bonjour") ==> obj (formel) référence "bonjour" (effectif)
- paramètre positionnel : l'association entre effectif et formel se fait grâce à la position
- paramètre nommé : l'association entre effectif et formel se fait par le nom du paramètre formel
- nombre variable de paramètres effectifs : non connu lors de la spécification du SP
  - exemple : print, max, min, all, any, etc.

Remarque : un paramètre peut être à la fois positionnel et nommé.

Paramètres positionnels

Paramètres positionnels

Un paramètre positionnel est identifié par sa position.
Le ième **paramètre effectif** correspond au ieme **paramètre formel**.

```
def f(a, b):
    print(a, b)
f(1, 2)      # a est lié à 1 et b à 2 (affiche : 1 2)
```

Appel en nommant les paramètres

Lors de l'appel, on peut nommer les paramètres.

```
f(a=1, b=2)    # Affiche : 1 2
f(b=2, a=1)    # Affiche : 1 2
f(1, b=2)      # Affiche : 1 2
f(b=2)         # TypeError: f() missing 1 required positional argument: 'a'
f(a=1, 2)      # SyntaxError: positional argument follows keyword argument
f(1, 2, a=1)   # TypeError: f() got multiple values for argument 'a'
```

Règles

- Tous les paramètres formels doivent recevoir une et une seule valeur
- Quand on commence à nommer un paramètre, on ne peut plus utiliser les positionnels

Valeur par défaut d'un paramètre

Règle : valeur par défaut

- On peut donner une valeur par défaut à un paramètre formel positionnel.
- Il faut donner une valeur par défaut à tous les paramètres positionnels suivants.
- Lors de l'appel, si on omet un paramètre effectif, sa valeur par défaut sera utilisée.

Exemple

```
def f(a, b=2, c=3):
    print(a, b, c)
f(1)              # 1 2 3
f(1, 0, 'x')      # 1 0 x
f(1, 0)           # 1 0 3
f(1, c='x')       # 1 2 x
```

Danger : les valeurs par défaut sont évaluées lors de la définition de la fonction

```
def g(x, l = []):
    l.append(x)
    return l
g(1)    # [1]
g(2)    # [1, 2]
```

Comment faire pour avoir une nouvelle liste à chaque fois ?

Nombre variable de paramètres

Principe

`*args` et `**kargs` : récupérer respectivement les arguments positionnels et nommés en surplus.
Remarque : on peut changer les noms 'args' et 'kargs' ?

Exemple

```
def f(a, b=2, c=3, *p, **k):
    print('a={}, b={}, c={}, p={}, k={}'.format(a, b, c, p, k))
f(1)                    # a=1, b=2, c=3, p=(), k={}
f(1, 4)                 # a=1, b=4, c=3, p=(), k={}
f(1, c=5)               # a=1, b=2, c=5, p=(), k={}
f(9, 8, 7, 6, 5, d=4, e=3)  # a=9, b=8, c=7, p=(6, 5), k={'d': 4, 'e': 3}
f(z=1, y=2, a=5)          # a=5, b=2, c=3, p=(), k={'y': 2, 'z': 1}
```

Paramètres seulement nommés (keywork-only argument)

Principe

C'est un paramètre formel qui ne peut pas être initialisé avec un paramètre formel positionnel mais seulement un paramètre effectif nommé.
Exemple : 'end' et 'sep' de `print`

Syntaxe

```
def f(a, *p, b):
    print("a = {}, p = {}, b = {}".format(a, p, b))

f(1, 2, 3)  # TypeError: f() missing 1 required keyword-only argument: 'b'
f(1, 2, 3, b=4) # a = 1, p = (2, 3), b = 4
```

Ne pas mettre `p` si on ne veut pas autoriser de paramètres positionnels supplémentaires.

Exercices

1. On considère la fonction `index` qui calcule l'indice de la première occurrence d'un élément dans une séquence à partir de l'indice début inclu jusqu'à indice fin exclu. Si l'indice fin est omis, on cherche jusqu'au dernier élément de la séquence. Si l'indice de début est omis, la recherche commence au premier élément (indice 0). Donner la signature de cette fonction.
2. Donner la signature de la fonction range. Dans sa forme générale, elle prend en paramètre l'entier de début, l'entier de fin et un pas. Si le pas est omis, il vaut 1. Si on ne donne qu'un seul paramètre effectif, il correspond à l'entier de fin et l'entier de début vaut 0.
3. L'appel `range(stop=5)` provoque `TypeError: range() does not take keyword arguments`. Est-ce que ceci remet en cause la signature proposée ?
4. Donner la signature d'une fonction max qui calcule le plus grand de plusieurs éléments.
5. Donner la signature de print.
6. Que fait la fonction suivante ?

```
def printf(format, *args, **argv):
    print(format.format(*args), **argv)
```

Mode de passage des paramètres

Question

Que peut faire un sous-programme sur les paramètres et que verra le sous-programme appelant ?

Exemple

```
def f1(s):
    s[0] = '5'

def f2(s):
    s = 'résultat ?'

liste = [0, 1] ; f1(liste)
liste                  # ???
chaine = "oui" ; f1(chaine)
chaine                 # ???
f2(liste)
liste                  # ???
f2(chaine)
chaine                 # ???
```

Espace de noms

Définition

Un **espace de nom** permet d'associer des objets à des noms.
Les espaces de noms sont hiérarchisés :

- espace de noms **prédéfini** (builtins) : objets prédéfinis.
- espace de noms **global** : noms définis au premier niveau (interprèteur)
- espace de noms d'une fonction : les noms définis dans cette fonction (à l'exécution)

Opérations

- `dir()` affiche les noms de l'espace de noms courant
- `vars()` affiche les noms et les objets associés de l'espace de noms courant
- `globals()` : l'espace de noms global
- `locals()` : l'espace de noms local

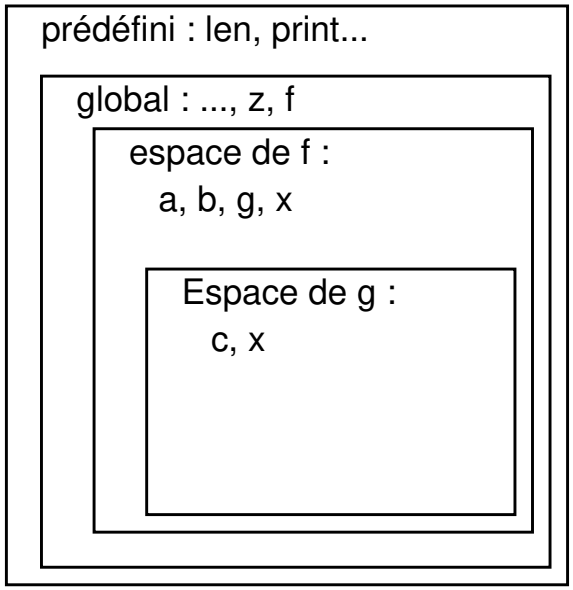
Masquage

Un nom déclaré dans un espace de noms plus interne masque les noms des espaces supérieurs.

Exemple : dans l'exemple qui suit, x défini dans g masque x défini dans f.

Exemple

```
1  z = 0    # nom 'z' dans l'espace de noms global
2
3  def f(a, b):     # nom 'f' dans l'espace de noms global
4                   # a et b dans l'espace de noms de f
5      def g(c):    # g dans l'espace de noms de f
6          x = 3
7          print('g :', dir())
8          print('g :', vars())
9
10     x = 5
11     print('f :', dir())
12     print('f :', vars())
13     g(a+b)
14
15 print(dir())
16 print(vars())
17 f(10, 20)
```



affiche

```
['...', 'f', 'z']
{'...', 'z': 0, 'f': <function f at 0x7fe6e8ceae18>}
f : ['a', 'b', 'g', 'x']
f : {'x': 5, 'g': <function f.<locals>.g at 0x7fe6e74c2510>, 'b': 20, 'a': 10}
g : ['c', 'x']
g : {'x': 3, 'c': 30}
```

Paramètres et variables locales

Exemple

```
def f(a, b):     # deux paramètres a et b
    x = 1     # variable locale de f (x)
    x = z     # utilisation de z, nom global
    print(vars())     # {}
    pass     # fin de la portée de a, b et x
z = 0   # nom de niveau global
```

Principe

- Initialiser un nom dans une fonction consiste à déclarer un nom local (variable locale).
- Une fonction définit un espace de noms dans lequel apparaîssent :
  - les paramètres
  - les variables locales
- Ces noms n'existent que pendant l'exécution de la fonction et disparaissent après.
- Ces noms masquent les noms déclarés dans un espace de noms englobant.
- Consultation possible des noms des espaces de noms englobants si non masqués.

Conseil

1. Ne jamais définir des variables au niveau globale (risque d'erreur).
2. Toujours définir ses instructions dans une fonction (puis appeler la fonction)
   - pour éviter de polluer l'espace de noms global avec les noms créés par les instructions.

Nom local, nom global et nom non local

Exemple

```
a = 10       # variable globale
def f():
    a = 5
    print('dans f(), a = ', a)

print(a) ; f() ; print(a)
```

- Que donne cette exécution ?
- Que se passe-t-il si on supprime `a = 5` ?
- Que se passe-t-il si on fait `print(a)` avant `a = 5` ?
- Que se passe-t-il si on ajoute `global a` en début de f() ?

global

Donner accès à un nom global

nonlocal

Donner accès à un nom d'un contexte englobant (et non global).

La récursivité

Définition

Un sous-programme récursif est un sous-programme dont l'implantation contient un appel à lui même.

Exemple : factorielle

```
def fact(n):
    if n <= 1:  # cas terminal
        return 1
    else:       # cas général
        return n * fact(n - 1)
```

Terminaison

- Prévoir un (ou plusieurs) cas de base (terminal) sans appel récursif.
- Dans le cas général, mettre en évidence un entier positif (taille du problème) qui décroit strictement à chaque appel récursif.

Exercice

Résoudre le problème des tours de Hanoï.

Intérêt des sous-programmes

1. **Structuration de l'algorithme :**
   - les sous-programmes correspondent aux étapes du raffinage
   - les étapes de l'algorithme apparaissent donc clairement
2. **Compréhensibilité :**
   - découpage d'un algorithme en « morceaux »
   - lecture à deux niveaux : 1) la spécification et 2) l'implantation
   - la spécification est suffisante pour comprendre l'objectif d'un sous-programme (et savoir l'utiliser)
3. **Factorisation et réutilisation**
   - un sous-programme évite de dupliquer du code
   - il peut être réutilisé dans ce programme et dans d'autres (modules)
4. **Mise au point facilitée**
   - tester individuellement chaque sous-programme avant le programme complet
   - erreurs détectées plus tôt, plus près de leur origine, plus faciles à localiser et corriger
5. **Amélioration de la maintenance :**
   - car le programme est plus facile à comprendre
   - l'évolution devrait rester localisée à un petit nombre de sous-programmes

Sommaire

Modules

Objectifs

- Organiser les fonctions, classes et autres éléments.
- Éviter les conflits de nom : un module définit un espace de noms

Moyen

- Un module est un fichier Python (extension .py) qui contient des définitions et des instructions.
- Le nom du module est le nom du fichier.
- Les définitions et instructions sont exécutées une seule fois au chargement du module (`import`)
- Les instructions ont pour but d'initialiser le module.
- Par convention, les noms qui commencent par un souligné (_) sont réputés locaux au module
  - ne devraient pas être utilisés depuis d'autres modules/programmes
  - ne sont pas importés par : `from mon_module import *`

Remarque

- Le nom `__name__` est intialisé avec `__main__` si le fichier est exécuté comme un script

module.py
```
print('dans module :', __name__)
```

main.py
```
import module
print('dans main :', __name__)
```

```
$ python module.py
dans module : __main__
$ python main.py
dans module : module
dans main : __main__
```

Sommaire

Les fonctions comme données

Les fonctions sont de objets

```
def f1():
    print("C'est f1 !")

type(f1)     # <class 'function'>
f2 = f1      # Un autre nom sur la fonction attachée à f1
f2()         # affiche "C'est f1 !"
f2.__name__ # 'f1'

def g(f):
    print('début de g')
    f()      # f doit être « appelable » (callable)
    print('fin de g')

g(f1)        # "début de g" puis "C'est f1 !" puis "fin de g"
```

Les lambdas : fonctions courtes, anonymes, avec une seule expression

```
cube = lambda x : x ** 3     # à éviter

def cube(x):                 # Mieux !  Plus clair !
    return x ** 3
```

Calcul d'un zéro d'une fonction continue

```
def zero(f, a, b, *, precision=10e-5):  # par dichotomie
    if f(a) * f(b) > 0:
        raise ValueError()
    if a > b:
        a, b = b, a
    while b - a > precision:
        milieu = (a + b) / 2
        if f(a) * f(milieu) > 0:
            a = milieu
        else:
            b = milieu
    return (a + b) / 2

def equation(x):
    return x ** 2 - 2 * x - 15

assert abs(5 - zero(equation, 0, 15)) <= 10e-5
```