Python : les sous-programmes

FULLSTACK : Algorithmique et programmation

Xavier Crégut <prenom.nom@enseeiht.fr>

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 (fichier mon_index.py)

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

def index(sequence, element):
    """Retourner l'indice de la première occurrence de 'element' dans 'sequence'

    :param sequence: la séquence
    :param element: l'élément cherché
    :returns: l'indice de la première occurrence de 'element'
    :raises ValueError: l'élément n'est pas dans la séquence
    """
    for indice, elt in enumerate(sequence):
        if elt == element:   # On l'a trouvé !
            return indice
    else:   # jamais trouvé
        raise ValueError('élement non trouvé')

if __name__ == "__main__":  # Le fichier est exécuté comme programme principal
    indice = index([1, 4, 2, 6], 4)
    print('OK si affiche 1 :', indice)

    assert index([1, 4, 2, 6], 1) == 0
    assert index([1, 4, 2, 6], 4) == 1
    assert index([1, 4, 2, 6], 2) == 2
    assert index([1, 4, 2, 6], 6) == 3
    assert index('Bonjour', 'j') == 3
```

Explications

1. Généralement, on définit un sous-programme dans un fichier (extension `.py`).
2. En Python, tout sous-programme est fonction (renvoie None si pas de résultat)
3. La définition d'une fonction commence par le mot-clé **def**.
4. La signature (première ligne) inclut le nom de la fonction suivie de ses paramètres entre parenthèses.
5. Le bloc qui suit (noter les deux-points et l'indentation !) inclut :
   - un commentaire de documentation
   - les instructions exécutées quand la fonction est appelée
6. La documentation est essentielle pour savoir utiliser la fonction : cf help() de Python
7. La documentation est normalisée : 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
8. Les instructions sont les instructions déjà vues :
   - Principe : la spécification décrit l'objectif (R0), les instructions le comment (voir raffinages)
   - L'instruction **return** arrête l'exécution de la fonction et est suivie de la valeur retournée par la fonction
9. On peut bien sûr définir plusieurs sous-programmes dans un même fichier
10. Quand est fichier est exécuté par l'interpréteur Python, une variable **name** est initialisée :
    - `__name__` vaut `'__main__'` si le fichier est exécuté comme programme principal
    - le nom du fichier sans .py s'il est importé (**import**)
    - ici, on l'utilise pour tester la fonction. On verra d'autres manières de le faire. <!–
11. Les fonctions dont le nom commence par un souligné (_) sont réputées locales au module (fichier). –>

doctest : mettre les tests dans la spécification comme des exemples

```python
def statistiques(donnees):
    """ Calculer la moyenne, le min, le max et et le nombre de max de donnees.

    Args:
        donnees: les données à traiter
    Returns:
        (moyenne, min, max, nb_max): quelques statistiques sur donnees.
    Raises:
        ValueError: une erreur...

    >>> statistiques([1, 3, 2])
    (2.0, 1, 3, 1)

    >>> statistiques([1, 2, 3, 3, 1, 1, 3, 3, 2, 3])
    (2.2, 1, 3, 5)

    >>> statistiques([])
    Traceback (most recent call last):
        ...
    ValueError: pas de valeurs
    """
    pass

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

- Exécuter ce programme. Donner ensuite son implantation... et la tester.

Avec pytest

Constat : On ne peut pas mettre tous les tests dans la spécification des sous-programmes !

Principe de pytest :

- écrire les tests dans des fichiers `*_test.py` ou `test_*.py`
- les fonctions qui commencent par 'test' sont considérées comme des programmes de test
- **assert** est utilisé pour savoir si le test réussit ou échoue

test_statistiques.py

```
import pytest
from statistiques import statistiques

def test_statistiques():
    assert statistiques([1, 3, 2]) == (2.0, 1, 3, 1)
    assert statistiques([15]) == (15.0, 15, 15, 1)
    assert statistiques([5, 5, 5]) == (5.0, 5, 5, 3)
    assert statistiques((5, 5, 1, 2, 3, 5)) == (3.5, 1, 5, 3)
    assert statistiques(range(1, 4)) == (2.0, 1, 3, 1)
    with pytest.raises(ValueError, match='pas de valeurs'):
        statistiques([])
```

Lancer les tests : pytest

- lance tous les tests trouvés
- l'option `--doctest-modules` permet de tester aussi les tests de type docstring

Conclusion : Préférer pytest (compatible avec unittest) !

Quelques éléments supplémentaires de pytest

- `@pytest.fixture` : définir le contexte du test
- `pytest.raises(ExpectedException)` : vérifie que le test lève bien l'exception attendue

test_list.py

```python
import pytest

@pytest.fixture # pour initialiser les paramètre liste1 des fonctions de test
def liste1():
    return [1, 5, 3, -8, 12]

def test_len(liste1):   # liste1 sera initialisé avec le résultat de la fonction liste1()
    assert 5 == len(liste1)

def test_contains(liste1):
    assert 1 in liste1
    assert -8 in liste1
    assert 10 not in liste1

def test_index(liste1):
    assert 0 == liste1.index(1)
    assert 4 == liste1.index(12)
    with pytest.raises(ValueError):
        liste1.index(2)

def test_avec_erreur(liste1):
    liste1.index(2)
```

Le résultat de l'exécution :

```
(python3.6) cregut@hpxc:~/ens/python/solutions/divers/tests$ pytest
============================= test session starts ==============================
platform linux -- Python 3.6.1, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /home/cregut/ens/python/solutions/divers/tests, inifile:
collected 5 items

test_list.py ...F
test_statistiques.py .

=================================== FAILURES ===================================
_____________________________ test_avec_erreur _____________________________

liste1 = [1, 5, 3, -8, 12]

    def test_avec_erreur(liste1):
>       liste1.index(2)
E       ValueError: 2 is not in list

test_list.py:24: ValueError
====================== 1 failed, 3 passed in 0.03 seconds ======================
```

Couverture des tests : coverage.py

- Essentiel en Python : l'interpréteur ne vérifie que la syntaxe. Il faut exécuter le code pour trouver les autres erreurs !
- Conséquence : S'assurer qu'il y a une couverture de 100 % de chaque instruction
- Coverage.py fait des calculs de couverture : instruction et branche
- Installation :
  pip install coverage
- Faire un calcul de couverture des tests unitaires
  coverage run -m py.test
- ou pour un programme particulier
  coverage run statistiques.py
- L'option `--branch` ajoute le calcul des branches
  coverage run –branch statistiques.py
- Produire un rapport (`-m` pour indiquer les lignes non exécutées : ou `--show-missing`) :

```
> coverage report -m
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
statistiques.py        20      1    95%   39, 38->39, 45->exit
```

- ou en HTML :
  coverage html

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 ok
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 paramètre positionnels supplémetnaires.

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                # ???
```

Nom local et nom global

Variable locale

Quand on initialise un *nom* dans une fonction, on définit un nom local qui n'existe que tant que l'exécution de la fonction n'est pas terminée.
Conseil : Ne jamais définir une variable au niveau fichier/module !

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() ?

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

Étapes pour définir un sous-programme

1. **Définir la spécification du sous-programme**
   a. Définir l'objectif du SP (équivalent R0)
   b. Identifier les paramètres du sous-programme : rôle puis nom, valeur par défaut ?
   c. Choisir un nom significatif pour le sous-programme
   d. Identifier les conditions d'applications
   e. Expliquer l'effet du sous-programme
   f. Rédiger la spécification du sous-programme à partir de ces informations (signature + docstring)
2. **Écrire des programmes de test (test unitaire)**
3. **Définir l'implantation du sous-programme**
   - Appliquer la méthode des raffinages : la spécification du SP est le R0
4. **Tester l'implantation du sous-programme**
   - corriger le sous-programme (ou les tests) en cas d'échec et rejouer tous les tests

Exemple : Écrire un sous-programme qui calcule le pgcd de deux nombres.

Conseils sur la définition de sous-programmes

- Dans un même SP, **ne pas mélanger traitement** (calcul) **et interactions avec l'utilisateur** (affichage ou saisie) :
  - Les traitements sont plus stables que l'IHM.
  - L'interface utilisateur peut changer, il peut y en avoir plusieurs
- Un SP doit **être une boîte noire** :
  - on devrait pouvoir le copier/coller [1] dans un autre contexte
  - ne pas dépendre de variables globales
- Un SP ne doit **pas avoir trop de paramètres**
  - soit mauvais découpage,
  - soit regrouper les paramètres avec un nouveau type (liste, tuple, classe...)
- Un sous-programme doit être découpé en sous-programmes si :
  - il est trop long (20 lignes, arbitraire !)
  - il est trop complexe (trop de structures de contrôle imbriquées)
- On doit **être capable d'exprimer clairement l'objectif du SP** (commentaire)...
  ...sinon c'est qu'il est mal compris !

1. C'est une image ! Il ne faut jamais faire de copier/coller !

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 instructions sont exécutées une seule fois au chargement du module (`import`)
- Elles ont pour but d'initialiser le module.
- Par convention, les noms qui commencent par un souligné (_) 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

Paquetage

Objectif

- Organiser les modules

Définition

- Un paquetage est un dossier (le nom du paquetage est le nom du dossier)
  - c'est la présence du fichier `__init__.py` qui fait que le dossier est considéré comme un paquetage
- Un paquetage contient des paquetages (sous-dossiers) et des modules (fichiers .py).
- Un paquetage peut aussi contenir les mêmes éléments qu'un module
  - ils sont définis (ou importés) dans le fichier `__init__.py`

Utilisation

```
xc/
    __init__.py     # qui définit sp1()
    m1.py           # qui définit sp()

>>> import xc   # on peut faire xc.sp1() mais pas xc.m1 (voir __all__)
>>> xc.m1       # provoquera : AttributeError: module 'xc' has no attribute 'm1'
>>> import xc.m1 ; m1.sp()
>>> from xc import m1 ; m1.sp()
>>> from xc.m1 import sp ; sp()
>>> from xc.m1 import sp as rsp ; rsp()
```

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

```python
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
```

Décorateur

Définition : Un décorateur est une fonction qui prend en paramètre un élément pour retourner un élément du même type décoré (avec des fonctionalités supplémentaires).

Exemple :

```
def fn_bavard(f):
    def f_interne(*p, **k):
        print('debut de f_interne()')
        f(*p, **k)
        print('fin de f_interne()')
    print('dans fn_bavard')
    return f_interne

@fn_bavard
def exemple(x, y='ok'):
    print('exemple:', y, x)

print('Appel à exemple')
exemple('?')
print(exemple.__qualname__)
```

Explications :

- La fonction fn_bavard est un décorateur car elle prend une fonction et retourne une fonction.
- `@fn_bavard` devant la fonction applique `fn_bavard` comme décorateur sur `exemple`
- C'est l'équivalent de : `exemple = fn_bavard(exemple)`

Résultat de l'exécution :

```
dans fn_bavard
Appel à exemple
debut de f_interne()
exemple: ok ?
fin de f_interne()
fn_bavard.<locals>.f_interne
```

Problème : `exemple.__qualname__` retourne... `f_interne` (et pas `exemple`) !

`help(exemple)` affiche la documentation de... `f_interne` (et pas `exemple`) !

Il faut utiliser la méthode `update_wrappers` de `functools` ou mieux `@functools.wraps(f)` devant la définition de `f_interne`.

Décorateur : tracer les appels des fonctions

Le module trace (fichier trace.py) :

```
import functools

_level = 0      # nombre d'appels imbriqués
_tabsize = 4    # nombre d'espaces pour indenter par appel de fonction

def trace(f):
    @functools.wraps(f)        # Copier les informations f dans local_trace
    def local_trace(*p, **k):
        global _level
        indentation = ' ' * _level * _tabsize
        _level += 1
        positionnels = [repr(pp) for pp in p]
        nommes = ['{}={}'.format(c, repr(v)) for c, v in k.items()]
        print(indentation + '[[ call {name}({args})'.format(name=f.__qualname__,
                args=', '.join(positionnels + nommes)))
        r = f(*p, **k)
        print(indentation + ']] returns ' + repr(r))
        _level -= 1
        return r
    return local_trace
```

Utilisation du module trace :

```
@trace
def f(x, *, a):
    print('f: x vaut {} et a {}'
        .format(x, a))

@trace
def f2(p, q):
    print('f2: p vaut {} et q {}'
        .format(p, q))
    f(p, a=q)

@trace
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)

f2(1, 'un')
print(fact(6))
print(fact.__qualname__)
```

Résultats :

```
[[ call f2(1, 'un')
f2: p vaut 1 et q un
    [[ call f(1, a='un')
f: x vaut 1 et a un
    ]] returns None
]] returns None
[[ call fact(6)
    [[ call fact(5)
        [[ call fact(4)
            [[ call fact(3)
                [[ call fact(2)
                    [[ call fact(1)
                    ]] returns 1
                ]] returns 2
            ]] returns 6
        ]] returns 24
    ]] returns 120
]] returns 720
720
fact
```