FONCTIONS

B. Landelle

Table des matières

I Introduction

1 Définitions

Définition 1. *Une* fonction *est un programme qui accepte en entrée des* arguments *(ou* paramètres*) et qui peut renvoyer un résultat.*

Vocabulaire : Un *appel* d'une fonction signifie l'utilisation de celle-ci.

Exemples : Fonctions natives sous python ou présentes dans les modules courants (`numpy`, `scipy`, `matplotlib`,...)

```
type, int, float, str, len, range, abs, np.sqrt, np.cos, np.sin, ...
```

On peut écrire des fonctions qui ne prennent pas d'arguments en entrée ou des fonctions sans résultat de sortie. Par exemple, une fonction qui renvoie la date ne nécessite pas d'arguments d'entrée. Une fonction peut afficher un message à l'écran sans renvoyer de résultat. Les fonctions informatiques diffèrent donc des fonctions mathématiques car elles peuvent interagir avec l'environnement extérieur et des appels successifs avec les mêmes arguments en entrée peuvent fournir des résultats différents.

Définition 2. *Une fonction est dite à* effet de bord *(ou* effet secondaire*) si elle modifie un état en dehors de son environnement local.*

Exemples : 1. La fonction `print(argument)` affiche la valeur de l'argument mais ne renvoie pas de résultat. Une affectation sur le résultat d'une telle fonction ne provoque pas d'erreur mais renvoie un type `None`.

```
>>> a=print("Bonjour")
Bonjour
>>> type(a)
<class 'NoneType'>
```

La fonction agit sur l'environnement extérieur (affichage dans la console) et ne renvoie rien.

2. Après l'importation `import numpy.random as rd`, on peut utiliser un générateur de nombres pseudo-aléatoire dans l'intervalle $[0\,;1]$ avec l'instruction `rd.rand()` :

```
>>> rd.rand()
0.9607538467232002
>>> rd.rand()
0.5323582955809545
```

Des appels successifs renvoient des résultats différents (c'est ce qu'on le souhaite !). La génération de nombres qui imitent le hasard est d'une importance majeure pour toute une gamme de techniques de *Monte-Carlo* qui visent à estimer certaines quantités d'intérêt en probabilités et statistiques (très utilisées en finance, assurance, ...)

Même s'il est possible d'écrire des fonctions à effet de bord, on s'efforcera, sauf dans certains cas précis (génération de pseudo-hasard, mesures de temps, fonctions qui agissent *en place*, ...), d'écrire des fonctions sans interaction avec l'environnement extérieur.

Définition 3. *Une fonction est dite* pure *si elle est sans effet de bord et pour un même argument renvoie un même résultat.*

Remarque : Cette notion est conforme à celle vue en mathématique, à savoir une fonction « boite noire » qui prend une entrée et renvoie un résultat sans interférer avec le monde extérieur.

Une fonction très utile : La fonction `help` renseigne sur le fonctionnement d'une autre fonction.

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

2 Syntaxe

Il existe essentiellement deux syntaxes pour créer une fonction en python :

Syntaxe standard

```
def nom_fonction(arguments):
    Instructions
    return résultat
```

On notera la présence de l'indentation pour le *corps* de la fonction. Cette indentation est indispensable. Le retour à l'indentation de `def` détermine la fin du corps de la fonction. L'instruction `return` provoque la sortie de la fonction et renvoie le résultat.

```
def␣nom_fonction(arguments):
␣␣␣␣Instructions
␣␣␣␣return␣résultat
```

Syntaxe courte

```
nom_fonction=lambda arguments : résultat
```

Remarque : Dans la majorité des cas, il est d'usage de saisir les fonctions que l'on code dans l'éditeur, dans un fichier `*.py` que l'on exécutera *a posteriori*. Il n'est pas impossible de saisir des fonctions directement dans la console mais dès que le codage est un peu élaboré, l'éditeur s'impose. On présente un cas rudimentaire de construction d'une même fonction selon trois procédés différents directement dans la console :

```
>>> f=abs
>>> f=lambda x:abs(x)
>>> def f(x):
        return abs(x)
```

Ces trois instructions « construisent » la fonction `f` qui réalise la valeur absolue.

⚠ Exceptées des situations très basiques comme celle qui précède, les fonctions sont saisies dans l'éditeur, dans un script python. L'exécution du script rend les fonctions accessibles dans la console.

Exemples : 1. Fonction déterminant la parité d'un entier.

```
def pair(n):
    return not bool(n%2)
```

2. Fonction calculant $\sum_{k=1}^{n} k$ pour n entier.

```
somme=lambda n : n*(n+1)//2
```

Exercice : Écrire une fonction `comp5(texte)` qui complète éventuellement la chaîne `texte` par des espaces et en renvoie les 5 premiers caractères. Le résultat devra systématiquement être un texte de 5 caractères.

Corrigé :

```
def␣comp5(texte):
␣␣␣␣return␣(texte+"␣␣␣␣␣")[:5]
```

Exercice : Écrire une fonction `sec(time)` qui convertit la chaîne `time` contenant une heure au format `"HH:MM:SS"` en nombres de secondes. Le résultat attendu est un entier.

Corrigé :

```
def sec(time):
    h=int(time[:2])
    m=int(time[3:5])
    s=int(time[6:])
    return s+60*(m+60*h)
```

3 Documentation

Lors de la définition d'une fonction, on peut, et il est souhaitable de le faire, documenter la fonction à l'aide d'une *docstring*. Il s'agit de fournir un descriptif du comportement de la fonction, descriptif qui s'affiche lors d'un appel de la fonction dans la console. Ceci est particulièrement utile lorsqu'on utilise une fonction conçue il y a longtemps ou alors conçue par un autre programmeur.

Syntaxe

```
def nom_fonction(arguments):
    """Description des arguments
    Description du résultat"""
    Instructions
    return résultat
```

Exemple :

```
def palindrome(n):
    """palindrome(n:int)->bool
    Teste si l'entier n est palindrome"""
    a=str(n)
    return a==a[::-1]
```

```
>>> help(palindrome)
Help on function palindrome in module __main__:

palindrome(n)
    palindrome(n:int)->bool
    Teste si l'entier n est palindrome
```

```
>>> palindrome(
(n)
palindrome(n:int)->bool
Teste si l'entier n est palindrome
```

FIGURE 1 – Documentation de la fonction `palindrome`

Définition 4. *L'ensemble des arguments avec leurs types en entrée et en sortie d'une fonction constitue la* signature *de la fonction.*

Exemple : Dans la documentation de la fonction `palindrome`, la ligne `palindrome(n:int)->bool` décrit sa signature.

Un fichier script peut et même doit être commenté. Les commentaires sont placés après le caractère # et ne sont pas interprétés comme du code python.

Exemple : On pourrait imaginer le fichier script suivant pour écrire puis tester la fonction fonction palindrome :

```
##################################################
#    Test du caractère palindrome d'un nombre
##################################################

def palindrome(n):
    """palindrome(n:int)->bool
    Teste si l'entier n est palindrome"""
    a=str(n)             # conversion de l'entier n en chaîne a
    return a==a[::-1]    # renvoie test d'égalité entre a et a retournée

# Scénario de test n°1

print("n=",2024)
print("test palindrome=",palindrome(2024))
print()


# Scénario de test n°2

x=123454321
print("n=",x)
print("test palindrome=",palindrome(x))
```

II Variables et arguments d'une fonction

1 Variable locale et globale

Définition 5. *Une variable* globale *est une variable créée dans un script (fichier* `*.py`*) à l'extérieur du corps d'une fonction.*

Proposition 1. *La fonction* `globals` *renvoie la liste des variables globales.*

Définition 6. *Une variable* locale *est une variable créée dans le corps d'une fonction et qui n'existe que dans le corps de cette fonction.*

Exemple :

```
a=5                          # a : variable globale

def sym(n):                  # n : argument
    """sym(n:int)->int
    Renvoie le nombre renversé"""
    a=str(n)                 # a : variable locale
    return int(a[::-1])
```

Une variable `a` globale est créée dans le script. Dans la fonction `sym`, qui renvoie le nombre dont l'écriture décimale est renversée par rapport à celle de l'argument, une variable `a` locale est créée dont l'existence se limite au corps indenté de la fonction.

Remarques : (1) Une variable locale disparaît avec la fin de l'indentation de la fonction. On peut faire l'analogie avec les lettres muettes en mathématique dans une somme, un produit, une intégrale ...
(2) Une variable locale n'est pas en conflit avec une variable globale du même nom : la fonction ne « voit » que la variable locale.

Exemple :

```
def sec(time):
    """sec(time:str)->int
    Convertit l'heure au format "HH:MM:SS" en secondes"""
    h=int(time[:2])
    m=int(time[3:5])
    s=int(time[6:])
    return s+60*(m+60*h)
```

Dans la console :

```
>>> h=12
>>> sec("10:30:23")
37823
>>> h
12
>>> m
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    m
NameError: name 'm' is not defined
```

On crée préalablement une variable globale `h`. L'appel de la fonction crée une variable `h` complètement indépendante de la variable globale de même nom. Les variables locales `h`, `m` et `s` n'existent plus hors du corps de la fonction. L'appel à `h` est donc sans ambiguïté un appel à la variable globale qui n'a en aucune manière été modifiée par la fonction.

Proposition 2. *Les arguments d'une fonction sont des variables locales avec une valeur initiale qui est celle fournie en paramètre.*

Proposition 3. *Une fonction peut être variable locale d'une autre fonction.*

Syntaxe

```
def fonction1(arguments):
    instructions
    ...
    def fonction2(arguments):
        instructions
        ...
    # fin de la fonction locale fonction2
    instructions...
```

Exemple : La fonction `maxtime` prend en arguments deux heures au format `"HH:MM:SS"` et renvoie la plus tardive. La fonction convertit les deux chaînes en secondes, détermine le maximum en utilisant la fonction `max` et convertit le maximum en secondes en une chaîne de caractères avec la fonction locale `conv` qui elle-même utilise une fonction locale `st2`. La fonction `sec` a été définie précédemment et est utilisable par la fonction `maxtime`.

```
def maxtime(time1,time2):
    """maxtime(time1:str,time2:str)->str
    Renvoie l'heure la plus tardive entre deux heures au format "HH:MM:SS"
    """
    m=max(sec(time1),sec(time2))
    def conv(s):
        """conv(s:int)->str
        Convertit une heure en seconde au format "HH:MM:SS"""
        def st2(x):
            """st2(x:str)->str
            Extrait les deux derniers caractères
            avec complétion éventuelle par des "0" à gauche"""
            a="0"+str(x)
            return a[-2:]
        res=st2(s//3600)+":" +st2((s//60)%60)+":"+st2(s%60)
        return res
    return conv(m)
```

On a créé au sein de la fonction `maxtime` une fonction locale `sec(time)` pour effectuer la conversion en seconde.

Proposition 4. *La fonction* `locals` *renvoie la liste des variables locales présentes dans la fonction en cours d'appel.*

Exemple :

```
def sec(time):
    h=int(time[:2])
    m=int(time[3:5])
    s=int(time[6:])
    print(locals())
    return s+60*(m+60*h)
```

Dans la console :

```
>>> sec("10:30:23")
{'time': '10:30:23', 'm': 30, 'h': 10, 's': 23}
37823
```

2 Effets de bord

Dans certaines circonstances, on peut être amené à écrire des fonctions avec effets de bord : fonctions qui utilisent des constantes physiques définies en début de script, fonctions qui produisent des graphiques, ...

Même si ce n'est pas souhaitable, une fonction peut donc utiliser une variable globale.

Exemple :

```
a=2
def f(x):
    x+=a
    return x
```

Dans la console :

```
>>> f(4)
6
```

La création de situations ambiguës d'une fonction utilisant une variable globale et cherchant à la modifier engendre un message d'erreur.

```
a=2
def test():
    a+=1
```

Dans la console :

```
>>> test()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    test()
  File "CH03_FONCTIONS.py", line 51, in test
    a+=1
UnboundLocalError: local variable 'a' referenced before assignment
```

Dans la première situation, la variable locale `x` est créée puis modifiée en utilisant `a`. Dans la deuxième situation, on essaie de modifier `a` qui n'existe pas localement mais globalement d'où l'échec.

Il est possible, bien que peu souhaitable en pratique, de permettre à une fonction de modifier une variable globale.

Syntaxe

```
def nom_fonction(arguments):
    global var_globale
    instructions
    ...
```

Exemple :

```
a=1
def test():
    global a
    a+=1
```

Dans la console :

```
>>> test()
>>> a
2
```

On est à l'opposé de la bonne hygiène de programmation ...

3 Arguments d'une fonction

D'après ce qui a été vu précédemment, les arguments d'une fonction sont des variables locales. Ainsi, une action sur les arguments devraient *a priori* se faire sans effet de bord, autrement dit sans modification sur l'environnement extérieur.

```
a=1
def incr(x):
    x+=1
    return x
```

Dans la console :

```
>>> incr(a)
2
>>> a
1
```

Toutefois, quand l'argument est une liste, les modifications opérées par python peuvent sembler déroutantes.

```
def change(L):
    L[0]=0
```

Dans la console :

```
>>> a=[5,6,7]
>>> change(a)
>>> a
[0, 6, 7]
```

La variable `L` est bien locale à la fonction et pourtant, l'affectation `L[0]=0` modifie la variable globale `a` vivant à l'extérieur de `change`. Ceci s'explique par le fait que le passage des arguments se fait par valeur de référence. La variable `L` est une étiquette locale sur la liste étiquetée par la variable `a`. Or, comme une liste est un objet mutable, la modification est faite directement sur l'objet, sans création d'un autre objet avec un nouvel étiquetage.

On peut remédier à cet effet de bord en forçant python à dupliquer l'objet `L` dans la fonction avec l'indexation globale `L[:]` ou avec la création d'un type `list(L)`. On présente deux versions du programme précédent mais sans effet de bord.

```
def change(L):
    a=L[:]
    a[0]=1
    return a

def change(L):
    a=list(L)
    a[0]=1
    return a
```

Dans la console :

```
>>> a=[5,6,7]
>>> change(a)
[1, 6, 7]
>>> a
[5, 6, 7]
```

Pour une liste contenant des sous-listes, ce qui précède ne suffit pas : il faut réaliser une copie en profondeur.

Proposition 5. *Une fonction peut être argument d'une autre fonction.*

Exemple : La partie positive d'une fonction f est définie par $\max(f, 0)$.

```
def pospart(f,x):
    """pospart(f:func,x:float)->float
    Renvoie max(f(x),0)"""
    return max(f(x),0)
```

Exercice : On considère la fonction `f` codée par :

```
def f(n):
    def g(x):
        return x**n
    return g(2)
```

1. Quelles sont les variables locales de la fonction `f` ? de la fonction `g` ?
2. Que produit la modification suivante :

```
def f(n):
    def g(x):
        n-=1
        return x**n
    return g(2)
```

Corrigé : 1. En ajoutant l'affichage

```
def f(n):
    def g(x):
        return x**n
    print(locals())
    return g(2)
```

on observe :

```
>>> f(5)
{'n': 5, 'g': <function f.<locals>.g at 0x0000028127E5B6A8>}
32
```

L'arugment `n` et la fonction `g` sont variables locales de `f`. En ajoutant l'affichage

```
def f(n):
    def g(x):
        print(locals())
        return x**n
    return g(2)
```

on observe :

```
>>> f(5)
{'n': 5, 'x': 2}
32
```

La variable `n` est considérée comme une variable locale au sein de la fonction `g` ce qui n'a rien d'évident.

2. Pourtant, la tentative de modification de l'argument `n` se solde par un échec.

```
>>> f(5)
Traceback (most recent call last):
  ....
    n-=1
UnboundLocalError: local variable 'n' referenced before assignment
```

III Quelques règles

1 Structure d'un fichier python

En général, un script python (fichier `*.py`) est composé de une ou plusieurs fonctions. Une fonction peut faire appel à une autre fonction et c'est même recommandé : on écrit plusieurs fonctions dont chacune résout un sous-problème d'un problème plus vaste. On obtient ainsi une organisation hiérarchisée de fonctions dont les plus « primitives » résolvant un problème simple sont appelées par des fonctions plus élaborées et ainsi de suite ...

Cette approche hiérarchisée permet de séparer les difficultés, d'organiser son code, d'avoir un script facile à lire, facile à corriger et plus évolutif. La phase de test y gagne aussi en souplesse.

Une bonne habitude consiste à commenter son code : pour soi, pour avoir les idées claires et surtout pour un relecteur futur.

On peut, selon les situations, incorporer des scénarios d'utilisation des fonctions avec affichage des sorties ou affichage graphique.

2 En cours de réalisation

Tant que le programme final n'est pas terminé et que les différentes fonctions n'ont pas été validées par des jeux de tests, on peut insérer à loisir des affichages de résultats intermédiaires pour contrôler le bon fonctionnement des programmes. On rend les fonctions aussi « bavardes » que nécessaire pour tester et surtout corriger son code.

```
def narciss(n):
    """narciss(n:int)->bool
    Teste si n entier à 3 chiffres est narcissique"""
    a=n%10
    b=(n//10)%10
    c=n//100
    print(c,b,a)
    return n==a**3+b**3+c**3
```

Une fois que le programme a été testé, les affichages intermédiaires sont enlevés pour garder une fonction qui fait ce qui est demandé et pas plus :

```
def narciss(n):
    """narciss(n:int)->bool
    Teste si n entier à 3 chiffres est narcissique"""
    a=n%10
    b=(n//10)%10
    c=n//100
    return n==a**3+b**3+c**3
```

3 Interpréter, Tester

Les scripts pythons à écrire dans le cadre du programme de CPGE seront en général constitués d'une ou plusieurs fonctions. Il est indispensable d'exécuter régulièrement le script que l'on écrit afin de le nettoyer de toute erreur de syntaxe (très souvent, oubli de parenthèse) et de le tester. Pour la phase de test, selon la complexité du programme, on peut tester directement dans la console les fonctions codées sur des exemples pertinents ou saisir dans l'éditeur un ou plusieurs scénarios de tests et faire afficher les résultats fournis.

Exemple : On saisit dans l'éditeur :

```
def pair(n):
    """pair(n:int)->bool
    Renvoie le test de parité de n"""
    return not bool(n%2)
```

puis on teste dans la console :

```
>>> pair(3)
False
>>> pair(4)
True
```

On peut aussi saisir le programme et le jeu de test dans l'éditeur :

```
def pair(n):
    """pair(n:int)->bool
    Renvoie le test de parité de n"""
    return not bool(n%2)

print("pair(3)=",pair(3))
print("pair(4)=",pair(4))
```

L'exécution affiche :

```
>>>
pair(3)= False
pair(4)= True
```

On peut aussi utiliser le module `pytest` qui permet, via l'invite de commande de lancer toutes les fonctions intitulées `test_xxx` où `xxx` désigne le nom d'une fonction à tester. La fonction `test_xxx` devra contenir un jeu de tests aussi exhaustifs que possible pour accorder un bon indice de confiance en la fonction testée.

On considère le fichier `exemple.py` contenant le code suivant :

```
def pair(n):
    """pair(n:int)->bool
    Renvoie le test de parité de n"""
    return not bool(n%2)

def test_pair():
    assert pair(3)==False
    assert pair(4)==True

def uni(L):
    """uni(L:list)->bool
    Renvoie True si L contient deux éléments distincts, False sinon"""
    return L[0]!=L[-1]
```

```
def test_uni():
    assert uni([1]*3)==False
    assert uni([1,1,2])==True
    assert uni([1,2,2])==True
    assert uni([1,2,1])==True
```

Dans l'invite de commande, on lance :

```
(...chemin...)>pytest exemple.py
```

On obtient :

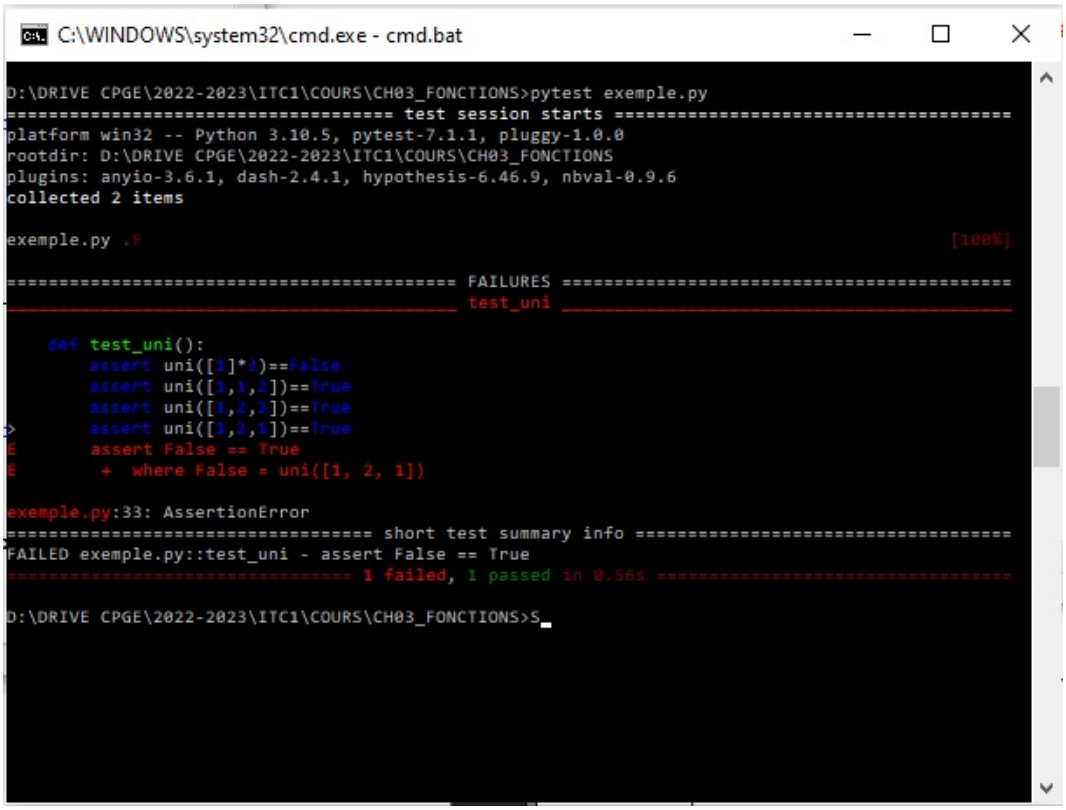

FIGURE 2 – Jeu de tests

Exercice : Dans l'exemple précédent, identifier le test ayant échoué et expliquer cet échec.

Corrigé :

Enfin, on recommande :

- de ne pas modifier les arguments d'une fonction (sauf dans le cas d'une fonction conçue *en place*, voir le chapitre sur les tris) ;
- d'écrire des fonctions sans effet de bord : éviter le recours à des variables globales, éviter les affichages de textes ou de graphiques depuis une fonction.