6 Docstring

Extrait du programme

Thème : Langages et Programmation

Contenus : Spécification
Capacités attendus : Prototyper une fonction.
Décrire les préconditions sur les arguments.
Décrire des postconditions sur les résultats.
Commentaires : Des assertions peuvent être utilisées pour garantir des préconditions ou des postconditions.

Contenus : Mise au point de programmes
Capacités attendus : Utiliser des jeux de tests.
Commentaires : L'importance de la qualité et du nombre des tests est mise en évidence.
Le succès d'un jeu de tests ne garantit pas la correction d'un programme.

Contenus : Utilisation de bibliothèques
Capacités attendus : Utiliser la documentation d'une bibliothèque.
Commentaires : Aucune connaissance exhaustive d'une bibliothèque particulière n'est exigible.

I. Utilisation de bibliothèques

Python est un langage très populaire car il est accompagné d'un nombre impressionnant de bibliothèques [1].
De base, on peut à peu près tout faire en Python, mais il peut être plus rapide et plus pratique d'utiliser un module contenant déjà la fonction voulue. Par exemple pour calculer le cosinus d'un angle...

a) Importer un module

Il existe plusieurs méthodes pour importer un module :

- La ligne 1 signifie :
  du module `math` *importe tout.*
  On peut alors utiliser la fonction `cos` et la constante `pi` du module.

```
1 from math import *
2 x = cos(pi)
```

  AVANTAGE : on ne cherche pas à comprendre, on importe tout ce que contient le module.
  INCONVÉNIENT : on peut surcharger la mémoire avec des fonctions inutiles. A éviter dans les systèmes embarqués.

- La ligne 1 signifie :
  du module `math` *importe la fonction* `cos` *et la constante* `pi`.
  On peut alors utiliser la fonction `cos` et la constante `pi` du module.

```
1 from math import cos, pi
2 x = cos(pi)
```

  AVANTAGE : on ne surcharge pas la mémoire avec des fonctions inutiles.
  INCONVÉNIENT : il faut rajouter à la main tous les éléments dont on a besoin.

- La ligne 1 signifie :
  importe le module `random`.
  La ligne 2 signifie :
  importe le module `math`.
  On peut alors utiliser la fonction `cos` et la constante `pi` du module `math` mais aussi la fonction `randint` du module `random`.
  On remarque que l'on doit préciser pour chaque élément, de quel module il provient.

```
1 import random
2 import math
3 angle = random.randint(-2,2)*math.pi
4 x = math.cos(angle)
```

  AVANTAGE : quand on a un projet un peu complexe, on sait pour toute fonction extérieure de quel module elle provient. Il peut aussi arriver que des fonctions provenant de modules différents aient le même nom. Ainsi, on peut les différencier [2].
  INCONVÉNIENT : c'est un peu plus long et fastidieux à écrire.

1. En Python, on a plus l'habitude de dire *modules*, mais il s'agit bien de la même chose.
2. Ainsi le module `math` et le module `numpy` ont tous les deux une fonction cosinus `cos`, mais en réalité ce n'est pas exactement la même fonction...

- La ligne 1 signifie :
  importe le module `random` *sous le nom* `rand`.
  La ligne 2 signifie :
  importe le module `math` *sous le nom* `m`
  On peut alors utiliser la fonction `cos` et la constante `pi` du module `math` mais aussi la fonction `randint` du module `random`.
  On remarque que l'on doit préciser pour chaque élément, de quel module il provient, mais sous le nouveau nom qu'on lui a donné.

```
1 import random as rand
2 import math as m
3 angle = rand.randint(-2,2)*m.pi
4 x = m.cos(angle)
```

AVANTAGE : on garde l'avantage de préciser l'origine de l'élément utilisé mais avec une écriture plus courte.
INCONVÉNIENT : on peut oublier sous quel nouveau nom on a appelé le module.

b) S'informer sur un module

- La fonction `dir` permet d'explorer le contenu d'un module.

```
1 >>> import math
2 >>> dir(math)
3 ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh
      ', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', '
      cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
      'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose'
      , 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
       'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh',
       'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

- Nous pouvons ensuite obtenir des informations particulières à l'aide de la fonction `help`.

```
1 >>> help(math.floor)
2 Help on built-in function floor in module math:
3
4 floor(x, /)
5     Return the floor of x as an Integral.
6
7     This is the largest integer <= x.
```

La fonction `floor` prend en paramètre un nombre `x` et renvoie le plus grand entier inférieur ou égal à `x`.

Exercice 1

1. Quelle est la fonctionnalité de `ceil` du module `math` ?
   .....
   .....
   .....
2. Quelle est la fonctionnalité de `hypot` du module `math` ?
   .....
   .....
   .....
3. Quelle est la fonctionnalité de `random` du module `random` ?
   .....
   .....
   .....

Il existe une autre solution pour obtenir la documentation d'un module ... rechercher sur internet à l'aide de son moteur de recherche favori ... Mais faites attention, il existe principalement deux versions de Python : Python 2 et **Python 3**.
Entre Python 2 et Python 3 il existe quelques subtilités qui font que ... quand vous avez un programme écrit en Python 2 qui tourne bien, vous ne pouvez pas le faire tourner en Python 3 mais vous n'avez pas envie de le réécrire en Python 3. On trouve donc encore actuellement de la documentation pour les 2 versions de Python.
Nous travaillons en Python 3, donc essayez de chercher les documentations correspondantes...
`https://docs.python.org/3/library/math.html`
`https://docs.python.org/3/library/random.html`

Tous les dépôts de modules sont officiellement répertoriés sur **PyPI** (*Python Package Index* à l'adresse suivante :
`https://pypi.org/`

II. Documentation

Voici programme en Python qui tourne bien :

```
def ma_fonction(t):
        m = 0
        for i in range(len(t)):
                if t[i] > t[m]:
                        m = i
        return m
```

Mais que fait-il ? ? ? A n'en pas douter, celui qui a conçu ce programme sait très bien de quoi il retourne, quand il l'a écrit, mais qu'en sera-t-il dans plusieurs jours, semaines, mois ? ? ? Si cette fonction est une partie d'un projet beaucoup plus vaste fait à plusieurs, comment les autres programmeurs comprendront ce programme ? ? ? S'il existe un bug dans le projet comment feront-ils pour trouver qu'elle est la partie qui provoque l'erreur ? ? ?
Un bon programmeur n'est pas forcément quelqu'un qui arrive à écrire un code qui fonctionne bien.

Un bon programmeur écrit toujours le plus simplement et le plus clairement *possible* et s'assure que son code soit bien **documenté** pour le rendre le plus compréhensible *possible*.

La première chose à faire est donc d'utiliser des noms de fonctions le plus explicite possible (même si c'est plus long à écrire). Dans notre cas, on pourrait choisir le nom suivant :

```
def maximum(t):
        m = 0
        for i in range(len(t)):
                if t[i] > t[m]:
                        m = i
        return m
```

Exercice 2
Mais que recherche exactement cette fonction ? Le maximum d'une liste ou la position du maximum d'une liste ? ? ?
. . . . . .

a) Spécification

La ***spécification*** d'un algorithme ou d'une fonction définit :
- son **rôle**
- ses **entrées**
- les **pré-conditions** sur ses entrées
- ses **sorties**
- les **post-conditions** sur ses sorties.

Le **rôle** est une description du problème que résout l'algorithme ou la fonction.
Les **entrées** et **sorties** sont ce qui fait le prototype d'une fonction.
Les **préconditions** sont des précisions sur la nature exacte des entrées.
Les **postconditions** sont des précisions sur la nature exacte des sorties.

Par exemple, si on souhaite écrire une fonction qui effectue une division entière :
- son rôle : effectuer une division entière
- ses entrées : deux nombres a et b
- les pré-conditions sur ses entrées : b doit être non nul
- ses sorties : le résultat de la division entière de a par b
- les post-conditions sur ses sorties : ce résultat doit être un entier.

Exercice 3
Donner les spécifications de notre fonction `maximum()` :

```
def maximum(t):
        m = 0
        for i in range(len(t)):
                if t[i] > t[m]:
                        m = i
        return m
```

- son rôle : .....
  .....
- ses entrées : .....
  .....
- les pré-conditions sur ses entrées : .....
  .....
- ses sorties : .....
  .....
- les post-conditions sur ses sorties : .....
  .....

Exercice 4
Donner les spécifications pour la fonction `occurence()` :

```
def occurence(tab,n):
    compteur = 0
    for case in tab:
       if case==n:
           compteur = compteur + 1
    return compteur
```

- son rôle : .....
  .....
- ses entrées : .....
  .....
- les pré-conditions sur ses entrées : .....
  .....
- ses sorties : .....
  .....
- les post-conditions sur ses sorties : .....
  .....

Exercice 5
Donner les spécifications pour la fonction `cesar()` :

```
def cesar(chaine,clef) :
    crypt = ''
    for car in chaine :
        nouveau = ord(car)+clef
        while nouveau > 122 :
            nouveau = nouveau - 26
        crypt = crypt + chr(nouveau)
    return crypt
```

- son rôle : .....
  .....
- ses entrées : .....
  .....
- les pré-conditions sur ses entrées : .....
  .....
- ses sorties : .....
  .....
- les post-conditions sur ses sorties : .....
  .....

b) Docstring

Avec le langage Python, on écrit la spécification dans des **docstring**, une chaîne de caractère destinée à documenter une partie du code, en étant interprétée comme telle.
C'est un texte que l'on écrit, entre 3 guillemets, au début du code de la fonction.

Une **docstring** est composée d'un descriptif du rôle de la fonction, des entrées, des pré-conditions, des sorties, des post-conditions **ET** d'un jeu *le plus complet possible* d'exemples.

Nous reprenons notre exemple de la fonction `division_entiere()` accompagnée par sa ***docstring*** :

```
def division_entiere(a,b):
    """
    Calcul la division entière de a par b

    entrées : deux nombres a et b
    pré-conditions : a et b sont des nombres, b est non nul
    sorties : le résultat de la division entière de a par b
    post-conditions : le résultat est un entier

    Exemples :
    >>>division_entiere(11,5)
    2
    >>>division_entiere(0,5)
    0
    """

    return a//b
```

La documentation peut ensuite être récupérer comme on a l'a vu dans la première partie de ce cours avec la fonction `help`.

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

division_entiere(a, b)
    Calcul la division entière de a par b

    entrées : deux nombres a et b
    pré-conditions : a et b sont des nombres, b est non nul
    sorties : le résultat de la division entière de a par b
    post-conditions : le résultat est un entier

        Exemples :
        >>>division_entiere(11,5)
        2
        >>>division_entiere(0,5)
        0
```

Réaliser une telle chaîne de documentation permet à l'utilisateur de la fonction de savoir à quoi peut servir la fonction, comment il peut l'utiliser et quelles conditions il doit respecter pour l'utiliser.
Il permet aussi au programmeur de la fonction de préciser le nombre et la nature de ses paramètres, la relation entre la valeur renvoyée et celle du ou des paramètres, ses idées avec quelques exemples...
Et même mieux encore ...

Exercice 6
Proposer une ***docstring*** pour notre fonction `maximum`.

c) Le module doctest

Les exemples donnés dans une chaîne de documentation peuvent être testés **automatiquement** à l'aide du module `doctest`.

```
>>> import doctest
>>> doctest.testmod()
```

La fonction `testmod` du module `doctest` est allée chercher, dans les ***docstring*** des fonctions, tous les exemples (reconnaissables à la présence des triples chevrons $>>>$) et a vérifié que la fonction documentée satisfait bien ces exemples. Dans le cas présent il n'y a eu aucun échec et donc pas de message.

En modifiant un des exemples, par exemple :

```
>>> division_entiere(0,5)
12
```

on obtient le message d'erreur suivant :

```
>>>doctest.textmod()
**********************************************************************
File "/home/laurent/NSI/division_entiere.py", line 15, in __main__.division_entiere
Failed example:
    division_entiere(0,5)
Expected:
    12
Got:
    0
**********************************************************************
1 items had failures:
   1 of   2 in __main__.division_entiere
***Test Failed*** 1 failures.
```

Qu'est ce que tout cela révèle ?
Tout d'abord que les tests ont échoué et qu'il y a eu 1 échec (cf dernière ligne) et que cet échec est dû à la fonction `division_entiere` (cf avant dernière ligne).
Ensuite que le test incriminé est celui concernant `division_entiere(0,5)` pour lequel le test a obtenu (`Got`) 0 en exécutant la fonction, alors qu'il attendait (`Expected`) 12 selon l'exemple donné par la documentation.
Lorsqu'il y a de tels échecs, cela invite le programmeur à vérifier son programme, ... ou bien les exemples de sa documentation, comme c'est le cas ici.

Si on souhaite rendre bavard le `doctest` même en cas de succès, il faut le mettre en mode verbeux :

```
>>> doctest.testmod(verbose = True)
```

On peut aussi vouloir rendre automatique le test à chaque exécution du programme en rajoutant ces quelques lignes à la fin du code :

```
1 if __name__ == '__main__':
2     import doctest
3     doctest.testmod()
```

Attention ! ! ! La réussite à ces tests n'est pas une preuve de correction, mais permet de guider la programmation et l'utilisateur.
Edsger Dijkstra disait « *Testing shows the presence, not the absence of bugs* », soit « *Tester un programme démontre la présence de bugs, pas leur absence.* »

d) Programmation défensive

En phase de conception ou plutôt de débogage, on peut vouloir s'assurer que les préconditions sont effectivement remplies. Pour éviter que l'on obtienne un résultat incohérent qui risquerait de déclencher une erreur plus tard, on peut interrompre le programme dès lors qu'une pré-condition n'est pas respectée. Dans ce cas, on utilise des **assertions** : on exprime chacune des préconditions après le mot clef `assert`.

```
1  def division_entiere(a,b):
2      """
3      Calcul la division entière de a par b
4
5      entrées : deux nombres a et b
6      pré-conditions : a et b sont des nombres, b est non nul
7      sorties : le résultat de la division de a par b
8      post-conditions : le résultat est un entier
9
10     Exemples :
11     >>>division_entiere(11,5)
12     2
13     >>>division_entiere(0,5)
14     0
15     """
16
17     assert isinstance(a, int) or isinstance(a, float), "a n'est pas un nombre"
18     assert isinstance(b, int) or isinstance(b, float), "b n'est pas un nombre"
19     assert b!= 0, "division par 0 impossible"
20
21     return a//b
```

Lignes 17 à 19, on s'assure que les variables a et b sont bien des nombres et que b ne prendra pas la valeur 0. Si ce n'est le cas, la fonction arrêtera son exécution avec le message d'erreur donné après la virgule et donc, n'essayera pas d'effectuer la division `a//b`.

```
>>> division_entiere("mot",5)
AssertionError: a n'est pas un nombre
>>> division_entiere(10,"mot")
AssertionError: b n'est pas un nombre
>>> division_entiere(12,0)
AssertionError: division par 0 impossible
```

Une fois que la fonction a été documentée, a passé tous les tests, on peut supprimer les assertions [3] si on est certain que le reste du projet respectera bien ces pré-conditions.
En général, une fois les assertions supprimer, pour éviter les bugs, la fonction renvoie le nombre -1 si les pré-conditions ne sont pas respectées. (sauf si évidemment la fonction effectue un calcul dont le résultat peut être le nombre -1)

Exercice 7
Proposer un ***doctest*** complet muni des ***assertions*** nécessaires pour notre fonction `maximum()`.

Exercice 8
Même chose pour notre fonction `occurence()`.

Exercice 9
Même chose pour notre fonction `cesar()`.

3. En les mettant en commentaires.

III. Création de notre propre module

Nous allons créer un module contenant des fonctions *vraiment très* utiles que nous appellerons `tres_utiles` :

- une fonction `division_entiere(a,b)` qui effectue la division entière de a par b ;
- une fonction `indice_maximum(t)` qui renvoie un tuple contenant l'indice de l'élément le plus grand d'une liste `t` et la valeur de ce maximum ;
- une fonction `plus_grand(a,b)` qui renvoie `True` si le nombre a est plus grand que le nombre b ;
- une fonction `maximum_3(a,b,c)` qui renvoie le maximum des trois nombres a, b et c ;
- une fonction `double(mot)` qui renvoie une chaîne caractères composée des caractères de l'argument `mot` répétés deux fois. Par exemple `double("bon")` renvoie `"bboonn"` ;
- une fonction `look_and_say(a)` qui renvoie un nombre qui lit et dit le nombre a. Par exemple, si $a = 1455666$ elle renvoie le nombre 11142536 car on a lu dans a un 1, puis un 4 puis deux 5 et enfin trois 6.

a) Proposer une spécification puis un jeu de tests puis des assertions pour chacune des nouvelles fonctions

1. une fonction `plus_grand(a,b)`
   Spécification :
   - son rôle : .....
   - ses entrées : .....
   - les pré-conditions sur ses entrées : .....
   - ses sorties : .....
   - les post-conditions sur ses sorties : .....

   Jeu de tests :
   - .....
   - .....
   - .....
   - .....
   - .....

   Assertions :
   - .....
   - .....

2. une fonction `maximum_3(a,b,c)`
   Spécification :
   - son rôle : .....
   - ses entrées : .....
   - les pré-conditions sur ses entrées : .....
   - ses sorties : .....
   - les post-conditions sur ses sorties : .....

   Jeu de tests :
   - .....
   - .....
   - .....
   - .....
   - .....

   Assertions :
   - .....
   - .....

3. une fonction `double(mot)`
   Spécification :
   - son rôle : .....
   - ses entrées : .....
   - les pré-conditions sur ses entrées : .....
   - ses sorties : .....
   - les post-conditions sur ses sorties : .....

   Jeu de tests :
   - .....
   - .....
   - .....
   - .....
   - .....

   Assertions :
   - .....
   - .....

4. une fonction `look_and_say(a)`
   Spécification :
   - son rôle : .....
   - ses entrées : .....
   - les pré-conditions sur ses entrées : .....
   - ses sorties : .....
   - les post-conditions sur ses sorties : .....

   Jeu de tests :
   - .....
   - .....
   - .....
   - .....
   - .....

   Assertions :
   - .....
   - .....

b) Création du module et utilisation du module

Écrire toutes les fonctions correspondantes dans un seul fichier que l'on sauvegardera sous le nom `tres_utiles.py`... Et voilà notre module est terminé ! C'est tout !

Pour pouvoir utiliser dans n'importe quel autre code une de nos fonctions, il suffit d'écrire [4], comme on en a déjà l'habitude :

```
1 from tres_utiles import look_and_say
```

Et voilà !

4. A condition que le fichier `tres_utiles.py` et notre nouveau programme se situe dans le même répertoire !!!