

Python 3 –TDs-Fipa1



Saison 1 - épisode 5

1	Découvrons et testons le langage Python	1
1.1	Les modules	1
1.2	Utilisation des fonctions dans un script	3
2	Mise en pratique	7

1 Découvrons et testons le langage Python

1.1 Les modules

Un module est un fichier script Python permettant de définir des éléments de programme réutilisables. Il est ainsi possible d'élaborer des bibliothèques de fonctions, de construire et organiser son application informatique (architecture logicielle).

Avantages : réutilisation, documentation, partage, partition de l'espace de nom du système.

La déclaration de l'instruction « **import <nom module>** » en tête de script donne accès aux ressources du module.

L'instruction « **from<nom module> import n1, n2** » donne accès à une sélection de fonctions n.

Le module et ses définitions existent dans leur espace de mémoire propre. Les modules sont ceux de bibliothèques de l'API Python ou propres à l'application. Dans l'API Python, il y a les modules standards installés de bases et les modules tiers à installer pour étendre les possibilités de développement.

Il est conseillé d'importer dans l'ordre :

- Les modules de la bibliothèque standard ;
- Les modules des bibliothèques tierces ;
- Les modules personnels ;

Testons !

```
>>> import math
>>> val=math.pi
>>> print(val)
3.141592653589793
```

Et aussi

```
>>> from math import sin,pi
>>> print("valeur de pi",pi,"sinus de(pi/4) : ",sin(pi/4))
valeur de pi 3.141592653589793 sinus de(pi/4) :  0.7071067811865475
```

Il est tentant d'utiliser systématiquement "[from](#) math [import](#) *", mais il s'agit généralement d'une mauvaise idée, en particulier lorsqu'une [fonction](#) existe dans plusieurs bibliothèques. Le choix de la [fonction](#) exécutée dépend alors de l'ordre d'importation des fonctions.

Les bibliothèques *math* et [numpy](#) ont toutes les deux des fonctions trigonométriques. Cependant, les fonctions de [numpy](#) peuvent s'appliquer à des tableaux, et pas celles de *math*. Exemples :

```
In [1]: import math
```

```
In [2]: import numpy as np
```

```
In [3]: a = np.linspace(0, np.pi/2, 3)
```

```
In [4]: np.sin(a)
```

```
Out[4]: array([ 0.          ,  0.70710678,  1.          ])
```

```
In [5]: math.sin(a)
```

```
-----
-
TypeError                                Traceback (most recent call
last)
<ipython-input-5-c8ade0db21c5> in <module>()
----> 1 math.sin(a)
```

TypeError: only length-1 arrays can be converted to Python scalars

```
In [6]: from numpy import *
```

```
In [7]: from math import *
```

```
In [8]: sin(a)
```

```
-----
-
TypeError                                Traceback (most recent call
last)
<ipython-input-8-66bf5e82d1e2> in <module>()
----> 1 sin(a)
```

TypeError: only length-1 arrays can be converted to Python scalars

```
In [9]: from math import *
```

```
In [10]: from numpy import *
```

```
In [11]: sin(a)
Out[11]: array([ 0.          ,  0.70710678,  1.          ])
```

1.2 Utilisation des fonctions dans un script

L'usage des fonctions dans un script rend le programme plus clair, plus lisible, plus efficace.

La définition des fonctions doit précéder leur utilisation.

L'ordre des déclarations des bibliothèques est important. Si une fonction se trouve dans deux bibliothèques, la première sera retenue (math, numpy ...).

Petite mise au point !

Le corps principal du programme est réservé à l'exécution séquentielle des fonctions.

Il est reconnu dans le fonctionnement interne de l'interpréteur sous le nom réservé « `__main__` ».

L'exécution d'un script commence toujours par la première instruction de cette entité où qu'elle puisse se trouver dans le programme.

Il, s'agit ici de comparer l'écriture du code pour optimiser le point d'entrée d'une application multi modules.

Observons !

Soit un module <code>testoutil1 .py</code>	Soit un module <code>testoutil .py</code>
<pre># -*- coding: utf-8 -*- """ Created on Fri Dec 11 10:34:54 2015 @author: gautropa """ # script python testoutil1.py def maville(): global code, ville ville = 'Brest' code = '29200' print(ville, code) maville()</pre>	<pre># -*- coding: utf-8 -*- """ Created on Fri Dec 11 10:34:54 2015 @author: gautropa """ # script python testoutil.py def maville(): global code, ville ville='Brest' code='29200' print(ville,code) if __name__ == '__main__': print('\n ce test doit afficher ce message par default') maville() else: print('module ',__name__, ' chargé')</pre>

1. Comparez leur code respectif !
2. Exécutez-les et comparez les résultats

Résultats

Du côté de testoutil1.py - exécution de testoutil1

```
C:\APPLIS\WinPython\python-3.6.1.amd64\python.exe "C:\Program Files\JetBrains\PyCharm 2018.1
\helpers\pydev\pydev_run_in_console.py" 64781 64782 C:/Users/gautropa/PycharmProjects/TdFipaS1Ep5/testoutil1.py
Running C:/Users/gautropa/PycharmProjects/TdFipaS1Ep5/testoutil1.py
Brest 29200

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\gautropa\\PycharmProjects\\TdFipaS1Ep5', 'C:/Users/gautropa/PycharmProjects/

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
> -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
PyDev console: using IPython 5.3.0

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
In[2]:
In[2]:
In[2]: testoutil1.maville()
Traceback (most recent call last):
  File "C:\APPLIS\WinPython\python-3.6.1.amd64\lib\site-packages\IPython\core\interactiveshell.py", line 2881, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-2-5fbb3b52c997>", line 1, in <module>
    testoutil1.maville()
NameError: name 'testoutil1' is not defined
In[3]:
In[3]:
In[3]: import testoutil1
Brest 29200

In[4]:
```

1

2

3

1. Le résultat de l'exécution directe du module est réalisé sans erreur par l'exécution de maville()
2. Le résultat de l'exécution directe testoutil1.maville() est en erreur sans import du module.
3. Si il y a import du module, la fonction maville() est exécutée directement

Du côté de *testoutil.py* - exécution de testoutil

```
C:\APPLIS\WinPython\python-3.6.1.amd64\python.exe "C:\Program Files\JetBrains\PyCharm 2018.1
.4\helpers\pydev\pydev_run_in_console.py" 64529 64530 C:/Users/gautropa/PycharmProjects/TdFipaS1Ep5/testoutil.py
Running C:/Users/gautropa/PycharmProjects/TdFipaS1Ep5/testoutil.py

ce test doit afficher ce message par défaut
Brest 29200

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\gautropa\\PycharmProjects\\TdFipaS1Ep5', 'C:/Users/gautropa/PycharmProjects/TdFipaS1Ep5'])

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)]
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
PyDev console: using IPython 5.3.0

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
In[2]:
In[2]:
In[2]:
In[2]: testoutil.maville()
Traceback (most recent call last):
  File "C:\APPLIS\WinPython\python-3.6.1.amd64\lib\site-packages\IPython\core\interactiveshell.py", line 2881, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-2-b93caaa2998f>", line 1, in <module>
    testoutil.maville()
NameError: name 'testoutil' is not defined

In[3]:
In[3]:
In[3]:
In[3]: import testoutil
module testoutil chargé
In[4]: testoutil.maville()
Brest 29200

In[5]:
```

1

2

3

1. Le résultat de l'exécution directe du module est redirigé vers l'exécution de la fonction `__main__`, sans erreur.
2. Le résultat de l'exécution de `testoutil.maville()` directement est impossible.
3. l'exécution de `testoutil.maville()` devient possible en effectuant au préalable un `import` qui charge le module.

Un point

En informatique une application sera organisée en plusieurs fichiers qui s'utilisent les uns les autres. Les ressources sont dispersées et accessibles selon une certaine logique. Un fichier de démarrage ou point d'entrée de l'application est dédié à l'exécution de l'application.

L'**algorithme** est la partition qui met en musique la logique du traitement. En Python le **module** est la base de l'organisation logicielle d'un programme réparti en plusieurs fichiers. Le module rassemble des **fonctions**.

Les ressources utiles seront dans des modules développés pour le projet ou accessibles dans des bibliothèques standards ou spécifiques des ressources du langage Python ou API (*Application Programmable Interface*).

La commande **import** permet l'accessibilité aux ressources d'un module à l'autre.

Un résumé

Selon G.Swinen « Apprendre à programmer avec Python3 »

Résumé : structure d'un programme Python type

```
# -*- coding:Utf8 -*-
#####
# Programme Python type #
# auteur : G.Swinen, Liège, 2009 #
# licence : GPL #
#####

#####
# Importation de fonctions externes :

from math import sqrt

#####
# Définition locale de fonctions :

def occurrences(car, ch):
    "Cette fonction renvoie le \
    nombre de caractères <car> \
    contenus dans la chaîne <ch>"

    nc = 0
    i = 0
    while i < len(ch):
        if ch[i] == car:
            nc = nc + 1
        i = i + 1
    return nc

#####
# Corps principal du programme :

print("Veuillez entrer un nombre :")
nbr = eval(input())

print("Veuillez entrer une phrase :")
phr = input()
print("Entrez le caractère à compter :")
cch = input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print("La racine carrée du cube", end=' ')
print("du nombre fourni vaut", end=' ')
print(rc)

print("La phrase contient", end=' ')
print(no, "caractères", cch)
```

Un programme Python contient en général les blocs suivants, dans l'ordre :

- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales).
- Les définitions locales de fonctions et/ou de classes.
- Le corps principal du programme.

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes. Vous pouvez vous-même définir de tels modules.

La définition d'une fonction comporte souvent une liste de PARAMÈTRES. Ce sont toujours des VARIABLES, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit toujours inclure au moins quatre éléments :

- l'initialisation d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrément du compteur.

La fonction "renvoie" toujours une valeur bien déterminée au programme appelant. Si l'instruction 'return' n'est pas utilisée, ou si elle est utilisée sans argument, la fonction renvoie un objet vide : 'None'.

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'ARGUMENTS, lesquels peuvent être des valeurs, des variables, ou même des expressions.

2 Mise en pratique

Nous allons réaliser un pseudo **Jeu de pendu**. Le principe de jeu est présenté comme suit :

Notre programme choisit un mot au hasard dans une liste de mots dont chaque mot est composé de huit lettres maximum. L'objectif du joueur est de tenter de trouver les lettres composant un mot.

A chaque coup, il saisit une lettre. Si la lettre figure dans le mot, l'ordinateur affiche le mot avec les lettres déjà trouvées. Celles qui ne le sont pas encore sont remplacées par des étoiles (*). Le joueur a 8 chances. Au-delà, il a perdu.

Rappelez-vous que le joueur ne doit donner qu'une seule lettre à la fois et que le programme doit bien vérifier que c'est le cas avant de continuer.

Nous voulons décomposer la solution en **trois modules** :

- **Le module *donnees.py*** contiendra les variables nécessaires à notre application : « le nombre de coups, la liste des mots.
- **Le module *fonction.py*** contiendra les fonctions utiles à l'application :
 - Importer votre module *donnee.py*
 - Complétez la fonction ***recup_lettre()*** :
Cette fonction récupère une lettre saisie par l'utilisateur. Si la chaîne récupérée n'est pas une lettre, on rappelle la fonction jusqu'à obtenir une lettre. (utilisez ***lower()***, ***isalpha()***)
 - Compléter la fonction ***choisir_mot()*** :
Cette fonction renvoie le mot choisi dans la liste des mots ***liste_mots***.
On utilise la fonction ***choice()*** du module *random* pour tirer un mot au hasard dans la liste ***liste_mots***.
 - Complétez la fonction ***recup_mot_masque(mot_complet, lettres_trouvees)***
Cette fonction renvoie un mot masqué tout ou en partie, en fonction du mot d'origine (type *str*) et des lettres déjà trouvées (type *list*).
On renvoie le mot d'origine avec des * remplaçant les lettres que l'on n'a pas encore trouvées.

▪ Le module *pendu.py* pour le jeu :

- Importer les modules utiles
- Algorithme :

Tant que l'utilisateur veut jouer

Choisissez un *mot à trouver* par tirage aléatoire dans la liste des mots : *choisir_mot()*

Créez une liste vide *lettres_trouvees*

Récupérez le mot masqué à partir du *mot à trouver* :

recup_mot_masque(mot_a_trouver, lettres_trouvees)

Le nombre de chance = nombre de coup

Tant que le mot n'est pas trouvé et le nombre de chance >0

Récupérez une lettre (*recup_lettre()*)

Si La lettre a déjà été choisie

Affichez "Vous avez déjà choisi cette lettre."

Sinon Si La lettre est dans le mot à trouver

Ajoutez la à la liste des lettres trouvées, *lettres_trouvees*, et affichez "Bien joué."

Sinon:

Le nombre de chance est décrémenté de 1

Affichez "... non, cette lettre ne se trouve pas dans le mot..."

Récupérez le mot masqué (*recup_mot_masque(mot_a_trouver, lettres_trouvees)*)

Si le mot à trouver est le mot trouvé:

Affichez "Félicitations ! Vous avez trouvé le mot » et Affichez le mot trouvé

Sinon

Affichez « PENDU !!! Vous avez perdu."

Demander si on continue la partie !