Chapitre (S1) 6 Modules

1 Modules ....
2 Exemples de modules ....

Objectifs

- Connaître les différentes manières d'importer et d'utiliser un module.
- Connaître et savoir utiliser quelques modules et fonctions associées.

1. MODULES

1.1. Présentation générale

1.1.1. Qu'est-ce qu'un module ?

Des fonctions Python enregistrées dans un fichier peuvent être mises à disposition d'autres programmes Python. Cette programmation modulaire offre une grande souplesse et évite de programmer plusieurs fois les mêmes fonctions.

Un *module* est un fichier qui contient des objets (déclarations de variables, définitions de fonctions et de classes [1] susceptibles d'être *importées*), étendant les fonctionnalités initiales de Python. Enregistré sous un nom `module_filename.py`, le script, une fois appelé, permet l'utilisation des objets du module.

Les modules regroupent des variables, fonctions ou classes qui tournent autour d'une même thématique (nous verrons ultérieurement quelques modules incontournables), comme le calcul numérique (associé à la manipulation de tableaux), la statistique, ou encore le tracé de fonctions. Utiliser un module permet d'étendre les possibilités de python sans avoir à reprogrammer les fonctions usuelles du domaine utilisé. L'ensemble des modules disponibles est très vaste et en constante évolution. On peut dire que si on se pose un problème un peu général, il existe probablement déjà un module qui résout ce problème.

1. Utilisées en programmation orientée objet.

1.1.2. Les différents modes d'importation

L'utilisation d'un module se fait à l'aide de l'instruction `import`. On peut l'utiliser soit directement dans la ligne de commande, soit dans un script python.

MÉTHODES D'IMPORTATION POSSIBLES

- Une première façon de faire est d'importer uniquement les objets (constantes, fonctions ou classes) qui nous intéressent, par une instruction du type

  ```
  from module_filename import objet
  ```

  Par exemple, une utilisation directe de `cos` et `pi` sans importation préalable

  ```
  >>> cos(pi)
  Traceback (most recent call last):
    File "<input>", line 1, in <module>
  NameError: name 'cos' is not defined. Did you mean: 'os'?
  ```

  produit une erreur (car `cos` et `pi` en sont pas connus nativement par `Python`), alors que les lignes :

  ```
  >>> from math import cos,pi
  >>> cos(pi)
  -1.0
  ```

  fournissent le résultat attendu.
  La ligne de code `from math import cos,pi` réalise l'importation et permet l'utilisation de `cos` et `pi`.
  Cependant, ce type d'importation n'est pas très utilisé car il nécessite de connaitre dès l'importation l'ensemble des objets que l'on souhaite utiliser.
- Une deuxième manière de procéder est d'utiliser une instruction du type

  ```
  from module_filename import *
  ```

  qui permet d'importer tous les objets d'un module, qui sont alors directement utilisables. On a par exemple :

```
>>> from math import *
>>> sin(pi/2)
1.0
```

Ici encore, ce type d'importation n'est pas très utilisé car il est assez gourmand en mémoire et peut poser des problèmes de conflits entre objets (cas de deux objets différents appartenant à deux modules distincts ayant le même nom).
Par exemple, on peut comparer les deux blocs de commandes suivants :

```
>>> from math import *
>>> from turtle import *
>>> radians(90)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: radians() takes 0 positional arguments but 1 was \
↪ given
```

qui renvoie une erreur, et :

```
>>> from turtle import *
>>> from math import *
>>> radians(90)
1.5707963267948966
```

qui renvoie la valeur en radians de l'angle 90°.
Ce comportement vient du fait que la fonction `radian` existe à la fois dans le module `turtle` (dans lequel elle ne prend aucun argument et fixe le système d'unité angulaire en radians), et dans le module `math` (dans lequel elle prend en argument un angle en degrés et le convertit en radians). L'ordre d'appel entraine l'écrasement des objets du premier module par ceux du deuxième qui possèdent le même nom.

Méthode d'importation recommandée

Les deux méthodes précédentes présentent des inconvénients, et on privilégie donc une troisième méthode, qui consiste à importer le module (et non pas les objets qu'il renferme), le plus souvent en le renommant à l'aide d'un *alias*[2]) à l'aide du mot clé `as`, avec une instruction du type :

```
import module_filename as nom_alias
```

Les objets du module sont alors accessibles par la syntaxe générale `nom_alias.objet`.

Par exemple, on peut écrire :

```
>>> import math as m
>>> import turtle as t
>>> t.radians()
>>> m.radians(90)
>>> m.sin(m.pi/2)
```

Ici les fonctions `radians` des modules `turtle` et `math` peuvent coexister, et on peut utiliser n'importe quel objet (constante, fonction ou autre) de chaque module.

Aide relative aux modules

Des informations sur le contenu d'un module peuvent être obtenues à l'aide des instructions suivantes.

- `dir(module_filename)` renvoie l'ensemble des objets présents dans un module,
- `help(module_filename)` renvoie des informations sur les fonctions du module,
- `help(module_filename.objet)` ou `help(nom_alias.objet)` (dans le cas où on a importé et renommé le module) renvoie l'aide relative à cet objet.

On peut ainsi mieux comprendre le problème lié à l'imprortation globale :

```
>>> import math as m
>>> import turtle as t
>>> help(t.radians)
>>> help(m.radians)
```

Remarque 1 Pour tout savoir, rien ne vaut le site officiel `https://docs.python.org/fr`!

2. Exemples de modules

2.1. Quelques modules parmi d'autres

Comme signalé précédemment, le nombre de modules existants est très important et on ne peut être exhaustif en la matière. On peut cependant signaler quelques modules particuliers :

- `math` – module de fonctions mathématiques usuelles,
- `cmath` – module qui reprend la plupart des fonctions du module `math` pour les nombres complexe `(a+bj)`,
- `random` – module de fonctions de hasard et de probabilité,

2. Un alias peut être vu comme une abréviation d'une commande ou la définition d'une commande complétée par des options souvent utilisées.

- `numpy` – Module principal de calcul scientifique. Il permet la définition et la manipulation efficace de **tableaux**. Il fournit des outils pour l'algèbre linéaire, pour l'analyse de Fourier, pour la manipulation de nombres pseudo-aléatoires, …
- `matplotlib` et le sous module `matplotlib.pyplot` – Module qui permet de créer des graphiques de type courbes, histogrammes, spectres, barres d'erreur, …
- `scipy` - Module de calcul scientifique qui complète `numpy` et `matplotlib`. Il fournit des outils d'optimisation, d'algèbre linéaire, de statistiques, de traitement du signal, du traitement d'images, de résolution d'équations différentielles.

2.2. Module NumPy

2.2.1. Généralités

Le module NumPy apporte des fonctions de calculs numériques autour des tableaux multidimensionnels ainsi que des fonctions de calculs mathématiques de haut niveau (algèbre linéaire, statistique, ajustement polynomial d'une courbe …). L'importation se fait usuellement à l'aide de l'alias suivant :

```
import numpy as np
```

```
>>> import numpy as np
>>> np.sin(np.pi/2)
1.0
>>> np.sqrt(2)
1.4142135623730951
```

Le module Numpy reprenant de nombreuses fonctionnalités des modules internes `math` et `random`, on peut ici encore rencontrer des différences entre fonctions issues de modules (et sous-modules) différents. Par exemple, la fonction `randint(a,b)`, où `a` et `b` sont deux entiers, renvoie un entier :

- compris dans $[a, b]$ pour la fonction du module `random`,
- compris dans $[a, b[$ pour la fonction du module `np.random`.

2.2.2. Tableaux

L'un des atouts du module `numpy` est la possibilité de manipuler des **tableaux** multidimensionnels. Ces derniers sont des objets permettant le stockage d'éléments repérés par des indices (de manière similaire aux listes), avec la condition que tous leurs éléments sont du même type. Le module propose des outils de manipulation et de calcul sur ces objets qui sont particulièrement efficaces. Ce qui rend ce module très performant dans le cadre du calcul numérique.

La création d'un tableau utilise la fonction `array`.

```
>>> import numpy as np
>>> T1 = np.array([1,2,3,4])
>>> T1
array([1, 2, 3, 4])
>>> type(T1)
<class 'numpy.ndarray'>
>>> T2 = np.array([[1,2,3],[4,5,6]])
>>> T2
array([[1, 2, 3],
       [4, 5, 6]])
```

Le module `numpy` permet de manipuler ces tableaux ou d'en extraire certaines informations. Ces fonctions sont disponibles dans l'aide python. Nous en présentons quelques-unes ci-dessous.

- `T = numpy.array([[1,0], [-1,1]])` : définition du tableau à 2 lignes et 2 colonnes
- `T[i,j]` : renvoie l'élément d'indice (i, j) du tableau, situé à la ligne $(i+1)$ et à la colonne $(j+1)$
- `T.shape` ou `np.shape(T)` : renvoie taille du tableau sous forme d'un tuple
  Exemple :
  ```
  >>> import numpy as np
  >>> T2 = np.array([[1,2,3],[4,5,6]])
  >>> T2[1,0]
  4
  >>> T2.shape
  (2, 3)
  >>> np.shape(T2)
  (2, 3)
  ```
- `T[i,:]` : renvoie les éléments de la ligne $(i+1)$ (sous forme d'un tableau)
- `T[:,j]` : renvoie les éléments de la colonne $(j+1)$ (sous forme d'un tableau)
  Exemple :
  ```
  >>> import numpy as np
  >>> T2 = np.array([[1,2,3],[4,5,6]])
  >>> T2[1,:]
  array([4, 5, 6])
  >>> T2[:,1]
  array([2, 5])
  ```

- `numpy.zeros([a,b,c,...])` : renvoie un tableau rempli de zéros de dimensions a, b, c, …
- `numpy.ones([a,b,c,...])` : renvoie un tableau remplis de 1 de dimensions a, b, c, …
- `np.linspace(a,b,n)` : création d'un tableau unidimensionnel de n valeurs flottantes comprises entre a et b (tous deux inclus)
- `np.arange(a,b,c)` : création d'un tableau unidimensionnel de valeurs flottantes comprises entre a (inclu) et b (exclu) par pas de c

Par exemple, on a deux manières presque équivalentes de créer le tableau unidimensionnel des nombres entiers entre 0 et 10 :

```
>>> np.linspace(0,10,11)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
>>> np.arange(0,11,1)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Les opérations algébriques sur les tableaux sont dites vectorielles ou élément par élément (« element wise » en anglais). Cette vectorisation évite le recours à des boucles **`for`**, ce qui contribue à l'efficacité de ce module. Cette propriété différentie fortement les listes des tableaux.

- somme de deux tableaux de même taille :
```
>>> A = np.array([1,2,3])
>>> B = np.array([4,5,6])
>>> A+B
array([5, 7, 9])
```
alors que pour deux listes, on obtient :
```
>>> A = [1,2,3]
>>> B = [4,5,6]
>>> A+B
[1, 2, 3, 4, 5, 6]
```
- multiplication par un nombre
```
>>> A = np.array([1,2,3])
>>> 2*A
array([2, 4, 6])
```
alors que pour une liste on a :
```
>>> A = [1,2,3]
>>> 2*A
[1, 2, 3, 1, 2, 3]
```
- élévation d'un tableau à une puissance
```
>>> A = np.array([1,2,3])
>>> A**3
array([ 1,  8, 27])
```

Ces opérations peuvent être appliquées à des tableaux de dimension supérieure à 1, comme ci-dessous avec des tableaux de dimension 2 :

```
>>> A = np.array([[1,2],[3,4],[5,6]])
>>> A
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> 2*A # produit de chaque coefficient par un réel
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> A**2 # chaque coefficient au carré
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> B = np.array([[0,1],[-1,2],[0,2]])
>>> A+B # somme des tableaux
array([[1, 3],
       [2, 6],
       [5, 8]])
>>> A*B # produit coefficient par coefficient
array([[ 0,  2],
       [-3,  8],
       [ 0, 12]])
```

Les fonctions mathématiques de numpy fonctionnent également de manière **vectorielles**[3] car on peut les appliquer sur des tableaux de nombres :

```
>>> import numpy as np
>>> x = np.linspace(0,2,11) # tableau de 11 valeurs de 0 à 2
>>> x
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ])
>>> np.exp(x) # on applique l'exponentielle à chaque valeur dans \
↪ x
array([1.        , 1.22140276, 1.4918247 , 1.8221188 , \
↪ 2.22554093,
       2.71828183, 3.32011692, 4.05519997, 4.95303242, \
       ↪ 6.04964746,
```

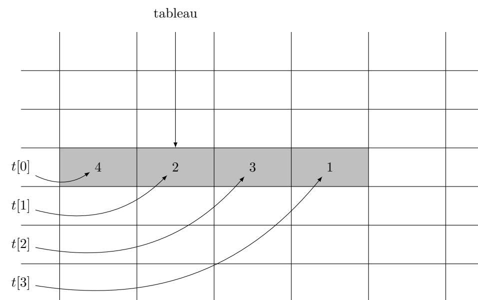
3. Ce qui n'est pas le cas des fonctions du module `math`.

ITC 2024-2025 – Lycée Michel MONTAIGNE – Bordeaux

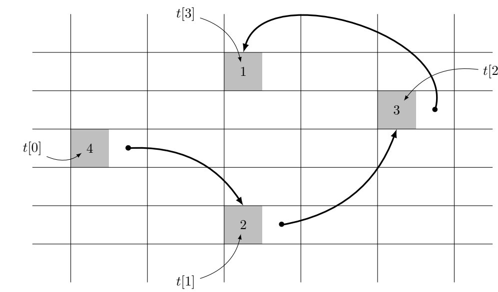
```
       7.3890561 ])
```

2.2.3. Listes ou tableaux ?

En informatique, les tableaux et les listes constituent deux structures de données linéaires. Bien que présentant des similitudes syntaxiques, elles sont pourtant différentes d'un point de vue usage. Dans un **tableau**, les données sont stockées dans des emplacements consécutifs de la mémoire. En outre, les données, de même type, occupent toutes un même nombre c de cases mémoires. L'accès à chacune de ces données se fait à coût constant, la connaissance de l'adresse mémoire m du premier élément permettant la détermination de celle d'un élément d'indice i : $m + ic$. Une contrepartie à cette propriété est que la taille d'un tableau est fixée une fois pour toute au moment de sa création. Ce qui entraîne également que sa taille ne peut pas évoluer.



Dans une **liste**, les données sont stockées dans des emplacements quelconques de la mémoire. L'accès à ces données nécessite de conserver à la fois la donnée mais également son emplacement dans la mémoire. C'est pourquoi les listes sont appelées des **listes chaînées**. À chaque donnée est associé un pointeur qui précise la localisation dans la mémoire de la donnée suivante (sauf pour la dernière donnée qui pointe vers une valeur particulière `nil` de fin de liste). L'accès à une donnée d'indice i nécessite donc de parcourir la liste depuis son début jusqu'à la donnée recherchée. Le coût est alors en $\mathrm{O}(i)$. Ce qui peut constituer un inconvénient. En revanche, la structure même de la liste autorise l'ajout et la suppression de données de manière dynamique. Il suffit de modifier les pointeurs.



On peut résumer les principales caractéristiques des listes et de tableaux qui constituent des avantages ou des inconvénients selon les besoins.

	Liste	Tableau
Structure de donnée	dynamique	statique
Accès aux données	coût variable	coût constant

Au sens strict du terme, les listes contiennent des données de même type. Les **listes Python**, de type `list`, sont une structure de donnée plus complexe. Elles conservent le caractère dynamique des listes, elles peuvent contenir des données de types différents et offrent un accès aux données de coût constant. Elles sont donc hybrides entre une liste au sens strict du terme et un tableau !

2.3. Module matplotlib

Le module `matplotlib` est l'une des bibliothèques python les plus utilisées pour représenter des graphiques. Nous nous intéresserons ici au sous-module `pyplot` de `matplotlib` qui regroupe un grand nombre de fonctions. Il s'agit d'un module particulièrement riche dont il n'est pas possible de décrire ici toutes les possibilités. Le lecteur intéressé est invité à lire la documentation disponible sur internet sur le site http://matplotlib.org.

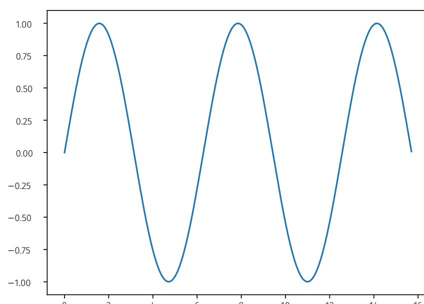
2.3.1. Graphe d'une fonction

L'un des premiers besoins graphiques est souvent le tracé d'une courbe représentative d'une fonction. Le script suivant présente le tracé d'un morceau de sinusoïde. L'affichage du graphique est obtenu grâce à l'appel de la fonction `show()`. Noter l'utilisation de la fonction `arange` du module `numpy` pour définir le tableau `x` des abscisses et l'application de la fonction sinus directement sur le tableau `x` pour obtenir

le tableau des ordonnées. La fonction `plot` réalise alors automatiquement l'association abscisse-ordonnée.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5*np.pi, 0.1);
y = np.sin(x)
plt.plot(x, y)
```

```
plt.show()
```

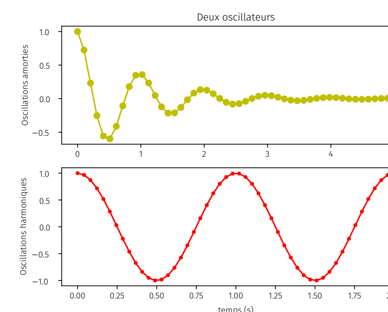


2.3.2. Plusieurs graphes

Plusieurs figures peuvent être représentées à l'aide de la fonction `subplot`.

```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('Deux oscillateurs')
plt.ylabel('Oscillations amorties')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('temps (s)')
plt.ylabel('Oscillations harmoniques')
```

```
plt.show()
```



SOLUTIONS DES EXERCICES