5 Les modules standards

Un *module* est un script Python contenant des définitions de variables, fonctions et autres objets dans l'optique de les utiliser dans d'autres programmes. Le fait, dans un script Python, d'aller chercher des objets d'un module se nomme l'*importation*. Python dispose nativement d'une grande collection de modules pré-existants, appelés « modules standards », aux applications diverses et variées : le module `math` implémente les constantes et fonctions mathématiques usuelles, le module `random` permet la génération de nombres aléatoires, le module `datetime` permet de manipuler et convertir des dates et heures …

5.1 Utilisation d'un module

5.1.1 Importation d'un module

Parmi les modules standards se trouve par exemple le module `math`. Celui-ci contient, entre autres choses, la variable `pi` (représentant la constante mathématique π) ainsi que la plupart des fonctions mathématiques usuelles : `sin`, `cos`, `exp`, … L'importation de variables ou fonctions depuis un module se fait à l'aide du mot-clé `import`. Celui-ci peut s'utiliser des trois manières suivantes.

1. **Importation basique.** La façon la plus simple d'importer tout le contenu d'un module est d'utiliser l'instruction `import nom` avec `nom` le nom du module à importer. Une fois cette instruction saisie, on accède aux objets (variables ou fonctions) du module en saisissant le nom du module et le nom de l'objet séparés par un point.

   À tester. Essayez le code suivant et observez les résultats dans l'explorateur de variables. (Si vous ne l'avez pas fait lors du TP précédent, décochez la case « *Exclure les appelables et les modules* » dans les options de l'explorateur de variables.)

   ```
   import math

   radius = 3
   disk_area = math.pi * radius ** 2
   angle = 5 * math.pi / 6
   y = math.sin(angle)
   ```

   Remarque. Le fait d'utiliser le nom du module comme préfixe permet d'éviter les collisions. En effet, si jamais vous importez deux modules `A` et `B` distincts ayant défini chacun une fonction appelée `func`, il n'y aura pas d'ambiguïté possible entre `A.func` et `B.func`.

2. **Importation en utilisant un alias.** La syntaxe précédente peut-être complétée en rajoutant le mot-clé `as` suivi d'un *alias*, c'est-à-dire un nouvel identifiant que vous voulez utiliser par la suite à la place du nom du module. Cela permet d'utiliser ensuite les objets du module en les préfixant par l'alias plutôt que par le nom complet du module.

À tester. Essayez le code suivant et observez les résultats dans l'explorateur de variables.

```
import math as m

radius = 3
disk_area = m.pi * radius ** 2
angle = 5 * m.pi / 6
y = m.sin(angle)
```

Remarque. L'intérêt d'utiliser un alias est en général de choisir un préfixe *plus court* que le nom du module, mais en pratique n'importe quel alias est autorisé du moment qu'il respecte les règles communes à tous les identifiants.

3. **Importation sélective.** Les deux syntaxes précédentes permettent en une instruction d'importer *tous* les objets du module choisi. Il est aussi possible d'importer seulement certains objets avec la syntaxe `from nom import objets` avec `nom` le nom du module dans lequel aller chercher les objets à importer, et `objets` les noms des objets à importer (séparés par des virgules s'ils sont plusieurs). Dans ce cas, on appellera ensuite les objets par leurs identifiants **sans les préfixer par le nom du module**.

À tester. Essayez le code suivant et observez les résultats dans l'explorateur de variables.

```
from math import pi, sin

radius = 3
disk_area = pi * radius ** 2
angle = 5 * pi / 6
y = sin(angle)
```

Attention. Quelle que soit la méthode que vous utilisez pour importer un module, il est recommandé de réaliser toutes vos importations au tout début du fichier (peu importe si vous utilisez les objets importés beaucoup plus loin dans le programme). En particulier, évitez à tout prix d'importer des modules dans le corps d'une fonction : l'importation est une opération assez lourde, vous n'avez pas envie de la répéter à chaque appel de la fonction. Il vaut mieux importer le module un fois pour toutes en dehors de la définition de la fonction (vous pourrez l'utiliser sans problème dans le corps de la fonction ensuite).

Exercice 5.1. On considère les trois façons suivantes d'importer (totalement ou partiellement) un module appelé `foo` contenant une fonction nommée `bar` :

1. `import foo as bar`
2. `from foo import bar`
3. `import foo`

Pour chacune des méthodes ci-dessus, quelle syntaxe dans la liste suivante permettra d'appeler la fonction `bar` du module `foo` ?

a `foo()`
b `bar()`
c `foo.bar()`
d `bar.foo()`
e `foo.foo()`
f `bar.bar()`

5.1.2 Aide sur un module

Vous pouvez obtenir de l'aide sur un module ou sur une fonction de ce module en passant son nom en paramètre à la fonction `help`. De plus, la fonction `dir` renvoie la liste des fonctions que contient le module passé en argument.

Attention. Il faut avoir déjà importé le module au préalable. Si de plus vous avez utilisé un alias pour ce module (à l'aide du mot-clé as), pensez bien à utiliser cet alias et non pas le nom du module.

À tester. Effectuer les import suivants :

```
import math
import random as rnd
```

Observez ensuite les résultats produits par les instructions suivantes dans la console :

1. `help(math)`
2. `help(random)`
3. `help(rnd)`
4. `help(math.sin)`
5. `help(rnd.randint)`
6. `dir(rnd)`

5.2 Quelques modules standards

5.2.1 Le module math

Comme expliqué en introduction, le module `math` implémente des constantes et fonctions mathématiques usuelles. Les plus courantes d'entres elles sont listées dans le tableau 5.1.

Objet mathématique	Notation en Python	Remarque
π, τ	`pi`, `tau`	τ est la constante égale à 2π.
e	`e`	On parle ici de la constante $e \approx 2,71828$. Pour la fonction, voir plus bas.
cos, sin, tan	`cos`, `sin`, `tan`	Ces fonctions attendent des angles en **radians**.
arccos, arcsin, arctan	`acos`, `asin`, `atan`	Ces fonctions renvoient des angles en **radians**.
exp, ln	`exp`, `log`	Les fonctions logarithmes en base 2 et 10 s'obtiennent respectivement avec `log2` et `log10`.
Racine carrée	`sqrt`	Déclenche une erreur si appliquée à un nombre strictement négatif.

TABLE 5.1 – Quelques fonctions du module `math`.

Remarque. Dans la suite de votre cursus d'ingénieur, vous délaisserez rapidement le module `math` pour le module `numpy` bien plus complet et performant. Ce dernier sera introduit en cours de mathématiques lorsque vous commencerez les travaux pratiques. En attendant, que cela ne vous empêche pas d'étudier le tableau 5.1 car les noms de fonctions sont exactement les mêmes dans les deux modules.

5.2.2 Le module random

Le module `random` permet la génération de nombres dits *pseudo-aléatoires*[1], et plus généralement propose des fonctions au comportement aléatoire. Les plus communes sont listées dans le tableau 5.2 (sauf mention contraire, tous les tirages aléatoires sont équiprobables).

Fonction	Description
`random()`	Renvoie un flottant aléatoire dans l'intervalle $[0, 1[$.
`uniform(a, b)`	Renvoie un flottant aléatoire compris entre a et b
`randrange(a, b, step)`	Renvoie un entier aléatoire dans l'intervalle `range(a, b, step)` (comme pour `range`, on peut ne donner que a ou que a et b).
`randint(a, b)`	Renvoie un entier aléatoire entre a et b (inclus). Équivalent à `randrange(a, b+1)`.
`choice(seq)`	Renvoie un élément aléatoire de la séquence `seq` (liste, tuple, chaîne de caractère, ...).
`sample(seq, k)`	Renvoie une liste de k éléments aléatoires de la séquence `seq` (sans répétition, déclenche une erreur si k dépasse la taille de `L`).
`shuffle(L)`	Mélange la liste `L` (cela ne renvoie pas une nouvelle liste mais modifie directement `L`).

TABLE 5.2 – Quelques fonctions du module `random`.

Exercice 5.2. Écrire un programme qui crée les listes suivantes :

- une liste `L1` contenant 10 entiers aléatoires compris entre 1 (inclus) et 10 (exclus),
- une liste `L2` contenant 8 couples de réels $(x, y) \in [-1, 1]^2$ aléatoires,
- une liste `L3` contenant un nombre aléatoire (entre 5 et 15) d'entiers aléatoires (entre -5 et 5 inclus).

5.2.3 Le module turtle

Le module `turtle` implémente en Python le principe de la *tortue graphique*, introduite dans le langage Logo. Il faut imaginer une tortue se déplaçant dans le plan avec un stylo, à laquelle on va pouvoir donner des instructions du type :

- avancer ou reculer d'une distance donnée,
- tourner sur soi-même d'un certain angle,
- lever ou baisser son stylo afin d'arrêter ou recommencer de laisser une trace écrite derrière elle lorsqu'elle se déplace,
- ...

Le tableau 5.3 recense les fonctions les plus courantes du module `turtle`.

1. On les appelle ainsi car, techniquement, il ne sont pas purement aléatoire mais calculés par un algorithme qui essaie d'imiter le mieux possible le hasard.

Fonction	Description
`forward(x)` *ou* `fd(x)`	Demande à la tortue d'avancer de x pixels.
`backward(x)` *ou* `bk(x)`	Demande à la tortue de reculer de x pixels.
`left(x)` *ou* `lt(x)`	Demande à la tortue de tourner de x degrés sur sa gauche.
`right(x)` *ou* `rt(x)`	Demande à la tortue de tourner de x degrés sur sa droite.
`penup()` *ou* `up()`	Demande à la tortue de lever le stylo.
`pendown()` *ou* `down()`	Demande à la tortue de baisser le stylo.
`speed(s)`	Règle la vitesse de la tortue entre 1 (le plus lent) et 10 (le plus rapide). On peut aussi donner la valeur 0 pour supprimer les animations (dans ce cas les fonctions de déplacement font bouger la tortue instantanément).

TABLE 5.3 – Quelques fonctions du module `turtle`.

Attention. Lorsque vous utilisez la tortue graphique, il faut terminer votre programme en appelant la fonction `done()` (pour libérer la fenêtre et pouvoir la fermer proprement) puis la fonction `bye()` (pour dire au revoir à la tortue et éviter quelques dysfonctionnements entre `turtle` et Spyder).

À tester. Testez le programme suivant, puis modifiez-le et observez les résultats obtenus.

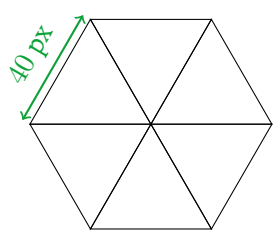
```
import turtle as tt

tt.forward(60)
for i in range(4):
    tt.right(90)
    tt.forward(20)
tt.forward(40)
```

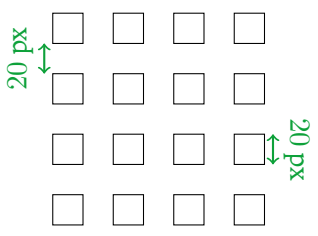
```
for i in range(2):
    tt.left(120)
    tt.forward(100)

tt.done()
tt.bye()
```

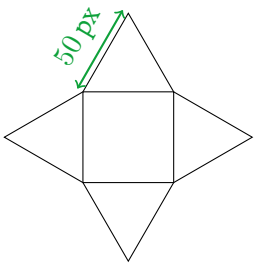
Exercice 5.3. Importez le module `turtle` puis créez les fonctions suivantes.

1. `triangle(side)` qui fait dessiner à la tortue un triangle équilatéral dont chaque côté sera de longueur `side` (argument optionnel valant 50 par défaut).
2. `square(side)` qui fait dessiner à la tortue un carré de côté `side` (optionnel, par défaut 50).

Utilisez ensuite vos fonctions pour créer les figures ci-dessous (en utilisant des boucles !).







5.2.4 Autres modules

Il existe encore une grande variété de modules standards dont vous trouverez la liste exhaustive dans la section dédiée de la documentation officielle de Python 3 [2]. Citons notamment :

— le module `os` qui permet, entre autres choses, d'interagir avec le système de fichiers de votre système d'exploitation : lister les fichiers d'un dossier, les renommer ou les supprimer, ...

— le module `datetime` qui fournit de nouveaux types de variables conçus pour stocker des dates, des heures, ...

— le module `fractions` qui ajoute un nouveau type `Fraction` permettant de stocker des rationnels de manière exacte,

— des modules spécialisés chacun dans le traitement d'un type de fichier précis : `csv`, `zipfile`, `gzip`, `json`, `wave`, ...

5.3 Le module Matplotlib

Terminons enfin ce chapitre en présentant un module qui ne fait pas partie des modules standards mais qui est fourni par défaut avec la suite Anaconda : le module Matplotlib. Ce dernier est spécialisé dans la représentation de données (graphiques, diagrammes, etc.) et vous le rencontrerez donc souvent lorsque vous utiliserez Python pour le calcul scientifique. Il est extrêmement riche, nous n'en présenterons ici qu'une infime partie mais vous pouvez aller consulter sa documentation [3] pour vous faire une idée de tout ce qu'il est possible de faire avec.

5.3.1 La fonction plot

La fonction `plot` du sous-module `matplotlib.pyplot` sert à représenter des points $(x_0, y_0), (x_1, y_1), \ldots$ dans un graphique 2d. Son utilisation la plus basique prend la forme suivante :

```
plot(x, y, fmt)
```

avec `x` une liste contenant les abscisses $x_0, x_1, \ldots$ des points à tracer, `y` une liste contenant leurs ordonnées $y_0, y_1, \ldots$, et `fmt` une chaîne de caractères précisant le format à utiliser pour le tracé. Cette chaîne de caractères est formée de plusieurs caractères donnant respectivement la couleur du tracé, le style de marqueur, et le style de ligne. Les caractères les plus courants sont donnés dans la table 5.4.

Notons les particularités suivantes concernant le format.

— Si on fournit un style de trait sans fournir de style de marqueur (par exemple `"r--"`), alors les points sont reliés entre eux mais aucun marqueur n'est affiché.

— Si on fournit un style de marqueur sans donner de style de trait (par exemple `"b*"`), alors les points ne sont plus reliés entre eux.

— Si on ne fournit ni style de marqueur, ni style de trait, alors le tracé est effectué en trait plein et sans marqueur.

— Si on ne fournit pas de couleur, alors c'est la prochaine couleur non-encore utilisée qui sera appliquée.

2. https://docs.python.org/3/library/
3. https://matplotlib.org/stable/api/index

Couleurs	Styles de marqueur	Styles de trait
`b` Bleu (*blue*)	`.` Point (par défaut)	`-` Trait simple
`r` Rouge (*red*)	`o` Cercle	`--` Tirets
`g` Vert (*green*)	`x` Croix	`:` Pointillés
`c` Cyan (*cyan*)	`s` Carré (*square*)	`-.` Alternance tiret/point
`m` Magenta (*magenta*)	`^` Triangle (vers le haut)	
`y` Jaune (*yellow*)	`v` Triangle (vers le bas)	
`k` Noir (*black*)	`*` Étoile	

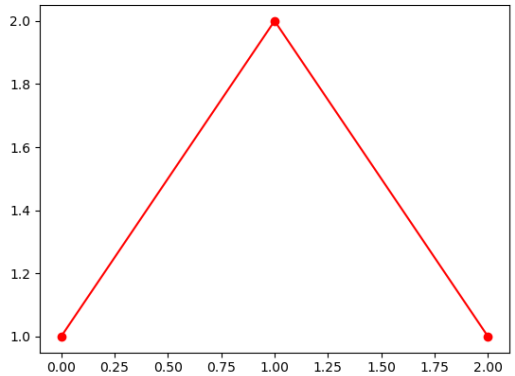
TABLE 5.4 – Caractères utilisé pour indiquer le format lors de l'utilisation de la fonction `plot`.

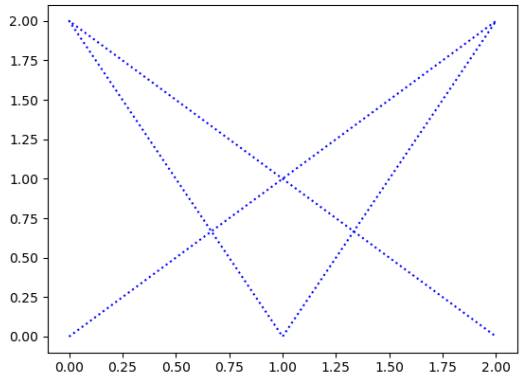
Par défaut, tous les appels à `plot` dessinent sur une même figure. Toutefois, on peut ouvrir une nouvelle figure entre deux tracés à l'aide de la fonction `figure()`. Une fois tous les tracés terminés, il faut appeler la fonction `show()` pour afficher les fenêtres dans lesquelles on a dessiné [4].

À tester. Essayez le code suivant.

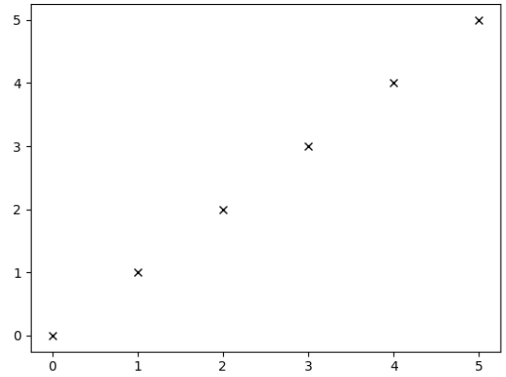
```
import matplotlib.pyplot as plt

plt.plot([-1, 1, 1, -1], [-2, -2, 3, 3], "r")
plt.plot([-2, 2], [0, 0], "k--")
plt.plot([-1, 0, 1, 2], [-1, 0, 1, 0], "bo" )
plt.figure()
plt.plot([1, 0, -1, 0, 1], [0, 1, 0, -1, 0], "g:*")

plt.show()
```

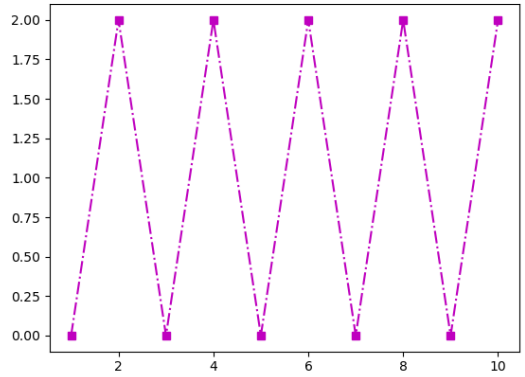
Exercice 5.4. Essayez de reproduire les figures suivantes à l'aide de la fonction `plot`.





4. Sous Spyder ce n'est pas nécessaire car c'est fait automatiquement par le logiciel, mais il vaut mieux prendre quand même l'habitude d'utiliser `show()` au cas où vous enverriez un jour votre code à quelqu'un qui utilise un autre éditeur.





Exercice 5.5. Écrivez une fonction `polygon` qui prend en arguments :

— `N` (obligatoire) : un entier,

— `R` (optionnel, par défaut 1) : un entier ou un flottant,

— `fmt` (optionnel, par défaut `"ro-"`) : une chaîne de caractères représentant un format à donner à `plot`,

et qui dessine sur un graphique le polygone régulier à N cotés obtenu en reliant entre eux les points

$$(x_k, y_k) = (R\cos(2k\pi/N), R\sin(2k\pi/N)), \qquad 0 \leqslant k \leqslant N.$$

Vous pouvez importer les fonctions et constantes mathématiques nécessaires depuis `math` *ou* `numpy`*, au choix.*

Application utile : graphe d'une fonction. On peut utiliser `plot` pour tracer le graphe d'une fonction f sur un intervalle $[a, b]$. L'astuce est simplement de relier entre eux des points de la forme $(x_k, f(x_k))$ avec

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$

des abscisses suffisamment rapprochées les unes des autres.

À tester. Considérons le code suivant.

```
import matplotlib.pyplot as plt

xs = []
ys = []

step = 2
x = -5
while x <= 5:
    xs.append(x)
    ys.append(1/(1 + x ** 2))
    x += step

plt.plot(xs, ys, "r")
plt.show()
```

1. Observez les résultats obtenus en prenant `step` égal à 2, puis 1, puis 0,5 et enfin 0,1.
2. De quelle fonction a-t-on tracé le graphe ? Sur quel intervalle ?

Exercice 5.6. Sur une même figure, tracez en bleu le graphe de la fonction cosinus et en rouge le graphe de la fonction sinus sur l'intervalle $[0, 6\pi]$.

Application utile : courbe paramétrée. De la même manière, pour tracer une courbe paramétrée de la forme

$$\begin{cases} x(t) = \dots \\ y(t) = \dots \end{cases} \qquad t \in [a, b],$$

il suffit de relier entre eux des points de la forme $(x(t_k), y(t_k))$ avec $a = t_0 < t_1 < \cdots < t_n = b$ des instants suffisamment rapprochés les uns des autres.

À tester. Considérons le code suivant.

```
import matplotlib.pyplot as plt
from numpy import cos, sin, pi

xs = []
ys = []

step = 1
t = 0
while t <= 2 * pi:
    xs.append(cos(t))
    ys.append(sin(t))
    t += step

plt.plot(xs, ys)
plt.show()
```

1. Observez les résultats obtenus en prenant `step` égal à 1, puis $0{,}5$, puis $0{,}1$ et enfin $0{,}02$.
2. Quelle courbe paramétrée a-t-on tracé ?

Exercice 5.7. Écrire une fonction `Lissajous` qui prend en argument deux entiers `p` et `q`, et qui utilise la fonction `plot` pour tracer la courbe paramétrée :

$$\begin{cases} x(t) = \cos(pt), \\ y(t) = \sin(qt), \end{cases} \qquad t \in [0, 2\pi].$$

5.3.2 Personnalisation du graphique

Voici quelques fonctions du module `matplotlib.pyplot` qui vous permettront de personnaliser vos graphiques créés avec la fonction `plot`.

- `title(S)` affiche comme titre du graphique la chaîne de caractères `S`.
- `xlabel(S)` affiche comme légende pour l'axe des abscisses la chaîne de caractères `S`.
- `ylabel(S)` affiche comme légende pour l'axe des ordonnées la chaîne de caractères `S`.
- `legend()` affiche une légende. Pour cela il faut, lors des appels de `plot` précédents, utiliser l'argument optionnel `label` pour donner un nom à la courbe ou aux points tracés (voir exemple ci-dessous).
- `grid()` affiche une grille en fond du graphique.
- `xlim((a, b))` restreint l'axe des abscisses entre les valeurs a et b.
- `ylim((a, b))` restreint l'axe des ordonnées entre les valeurs a et b.

Le code page suivante donne un exemple d'utilisation de ces fonctions. N'hésitez pas à modifier certaines lignes pour comprendre leurs effets respectifs.

```
import matplotlib.pyplot as plt
from numpy import cos, exp

X = []
Y_cos = []
Y_top = []
Y_bottom = []

x = -2
while x <= 5:
    X.append(x)
    k = exp(-x)
    Y_cos.append(k * cos(5*x))
    Y_top.append(k)
    Y_bottom.append(-k)
    x += 0.05

plt.plot(X, Y_cos, "r", label="exp(-t)cos(5t)")
plt.plot(X, Y_top, "b--", label="exp(-t)")
plt.plot(X, Y_bottom, "g--", label="- exp(-t)")
plt.title("Régime pseudo-périodique")
plt.xlabel("t (en s)")
plt.ylabel("X(t) (en m)")
plt.ylim((-2, 2))
plt.legend()
plt.grid()
plt.show()
```

À retenir

— Connaître les différentes façons d'importer un module et de l'utiliser ensuite.
— Savoir utiliser la fonction `plot` de Matplotlib et connaître les options de format classiques (table 5.4).
— Savoir tracer le graphe d'une fonction ou une courbe paramétrée avec Matplotlib.
— Mots-clés à connaître : `import`, `as`, `from`.
— Fonctions et constantes du module `math` à connaître : toute la table 5.1.
— Fonctions du module `random` à connaître : toute la table 5.2.
— Fonctions du module `turtle` à connaître : toute la table 5.3.
— Fonctions du module `matplotlib.pyplot` à connaître : `figure`, `grid`, `legend`, `plot`, `show`, `title`, `xlabel`, `xlim`, `ylabel`, `ylim`.
— Autres fonctions à connaître : `dir`, `help`.