Algorithmique

Introduction à Python

Anicet E. T. Ebou, ediman.ebou@inphb.ci

Ce travail est soumis à une licence internationale Creative Commons Attribution 4.0.

01

Introduction

Introduction

Python est un langage de haut niveau, c'est-à-dire un langage de programmation orienté vers les problèmes à résoudre, permettant d'écrire facilement des programmes à l'aide de mots usuels (en anglais) et de symboles mathématiques.

A contrario, un langage de bas niveau se rapproche du langage machine (dit binaire) et permet de programmer à un niveau très avancé, ce qui induit des temps de calculs réduits pour un problème donné par rapport à un langage de haut niveau.

Introduction

Ce langage a été développé par Guido Von Russom à la fin des années 80 et au début des années 90. Celui-ci a nommé le langage en référence à la troupe d’humoristes britanniques des Monthy Python.

Introduction

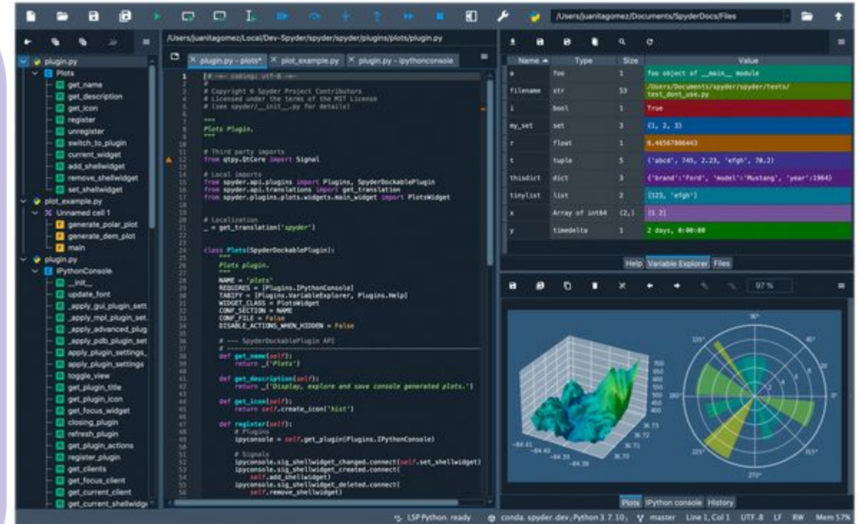
C’est un langage de programmation impérative, orientée objets, permettant aussi l’utilisation de la programmation fonctionnelle. Il est mutli-plateformes, c’est-à-dire qu’il peut être utilisé dans des environnements Unix, Mac-Os ou Windows, ou encore Android et iOS.

02

Environnement de travail: IDE Spyder

Spyder

Spyder est un environnement scientifique gratuit et open source écrit en Python, pour Python, et conçu par et pour des scientifiques, des ingénieurs et des analystes de données.



Spyder: installation

Pour installer Spyder:

1. Téléchargez le programme d'installation du fichier exécutable Windows 64 de Spyder à partir de la page de téléchargements pour Windows et macOS de Spyder.
2. Ensuite, double-cliquez sur le fichier téléchargé pour ouvrir le programme d'installation. Si un avertissement de sécurité s'affiche, vous devrez peut-être cliquer sur Oui, OK, Ouvrir, Autoriser ou autre.

Spyder: installation

3. Sous Windows, si une boîte de dialogue SmartScreen apparaît, cliquez sur Plus d'infos puis sur Exécuter quand même, et suivez les étapes du programme d'installation. Sous macOS, ouvrez l'image disque et faites glisser Spyder dans votre dossier Applications.

Spyder: installation

Pour exécuter Spyder, vous pouvez simplement utiliser la méthode habituelle de votre système d'exploitation pour lancer des applications, par exemple à partir du menu Démarrer de Windows (ou de la barre des tâches, si vous l'avez épinglée à cet endroit), ou à partir de Launchpad, Spotlight ou du dossier Applications de macOS (ou du Dock, si vous l'y avez ajouté).

Spyder: installation

Sur macOS, la première fois que vous ouvrez Spyder, vous pouvez voir un message indiquant qu'il ne peut pas être ouvert car le développeur ne peut pas être vérifié. Si tel est le cas, cliquez avec le bouton droit de la souris sur l'application, sélectionnez Ouvrir, puis cliquez sur Ouvrir dans la boîte de dialogue résultante, et l'avertissement ne s'affichera plus. Vous pouvez également cliquer sur Ouvrir de toute façon sous Sécurité et confidentialité ‣ Général dans les Préférences système.

Environnement de développement

L'environnement de développement est constitué de plusieurs fenêtres:

- Un éditeur de texte destiné à la saisie des programmes qui offre par défaut un certain nombre de fonctionnalités qui améliore l'expérience utilisateur.
  - Coloration syntaxique: Les mots clé du langage sont colorés ce qui améliore la lisibilité des programmes.
  - Indentation automatique: Automatiquement, un espace est créé lors de l'écriture du corps d'un bloc.

Environnement de développement

- Complétion automatique: Lors de la saisie, différents termes permettant de compléter le mot sont proposés à l’utilisateur.

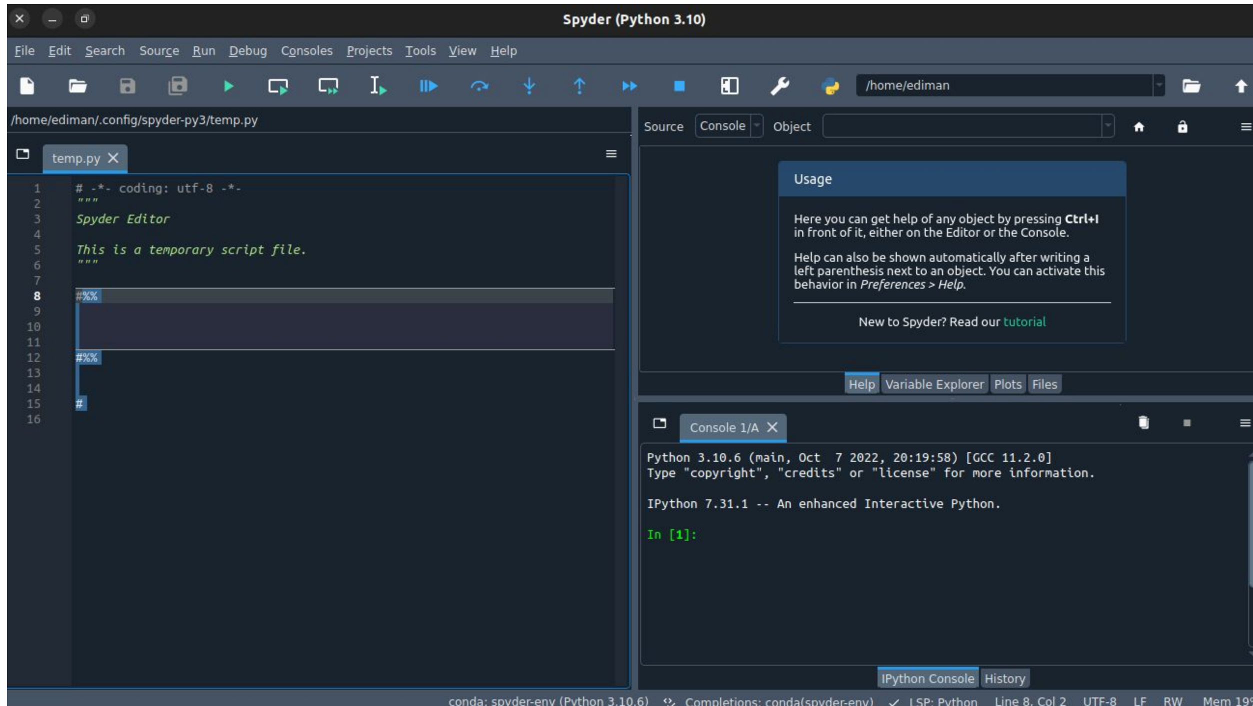
Un interpréteur interactif (appellé console) qui permet d’exécuter des instructions en ligne de commande.

Il suffit pour cela de taper du code directement à la suite de l’invite >>>. L’instruction tapée est évaluée et le résultat est alors affiché dans la console.

Environnement de développement

- Un explorateur de variables, qui permet de connaître les valeurs contenues dans les variables en cours d'utilisation.
- Un explorateur de fichiers qui permet de se déplacer dans l'arborescence de fichier et d'en charger dans l'éditeur de texte.

Environnement de développement



Explorateur de variables et de fichiers (en cliquant sur les différents onglets).

Interpréteur interactif appelé aussi console

Editeur de texte

03

Expressions

Expressions

Une expression est une suite de caractères définissant une valeur. Pour connaître cette valeur, la machine doit évaluer l'expression. Voici quelques exemples numériques :

Console

```
>>> 1+4              # 5
>>> 2.1+7            # 9.1
>>> 5/2              # 2.5
>>> 5//2*4.5         # 9.0
```

Expressions

Les valeurs possèdent ce qu'on appelle un ***type***: par exemple entier, flottant, booléen, chaîne de caractères, liste, fonction...

Le type détermine les propriétés formelles de la valeur (par exemple, les opérations qu'elle peut subir) et matérielles (par exemple, la façon dont elle est représentée en mémoire et la place qu'elle occupe).

Expressions

Pour connaître le type d'une expression après évaluation, il suffit de le demander à Python à l'aide de type :

Console

```
>>> type(1+4)        # <class 'int'>
>>> type(2.1+7)      # <class 'float'>
>>> type(5/2)        # <class 'float'>
>>> type(4<7)        # <class 'bool'>
>>> type("blabla")   # <class 'str'>
```

Expressions

Comme dans la plupart des langages de programmation, une expression en Python est soit:

- Une constante comme 2 ou 3.5;
- Un nom de variable comme x, i, ou compteur;
- Le résultat d’une fonction appliquée à une ou plusieurs expressions, comme PGCD(5,9).

Expressions

Comme dans la plupart des langages de programmation, une expression en Python est soit:

- La composée de plusieurs expressions réunies à l’aide d’opérateurs, comme not a, 3**6, (6+7)*8. Les parenthèses servent comme en mathématiques à préciser quels opérateurs doivent être évalués en premier.

04

Affectation

Rôle de l'affectation

L'affectation consiste à mettre la valeur d'une expression dans une case mémoire repérée par le nom de la variable (penser à une boîte).

Notations de l’affectation

En algorithmique: `nom_variable <- expression`

En Python: `nom_variable = expression`

= se lit « prend la valeur » ou « prend pour valeur ».

Exemple (à exécuter dans la console Python)

Console

```
>>> ma_var = 1               # simple affectation
>>> ma_var
1
>>> mon_autre_var = 2 + 2   # prend la valeur de l’expression
>>> mon_autre_var
4
>>> mon_autre_var = ma_var # prend la valeur contenue dans ma_var
```

Exemple (à exécuter dans la console Python)

Console

```
>>> mon_autre_var
1
>>> ma_var = ma_var + 1     # incrémentation
>>> ma_var
2
>>> mon_autre_var          # n'a pas subi l'incrémentation
1
```

Remarques: Erreurs fréquentes

- Il faut que la partie gauche de l’affectation soit bien un nom de variable : attention en Python à ne pas mettre de tiret - dans les noms de variable. Le signe - est compris comme une soustraction.
- Seul le contenu de la variable dont le nom est à gauche est changé, et ceci seulement au moment de l’affectation. L’affectation n’attache pas à la variable le comportement à venir de l’expression (même si c’est parfois le cas dans certains langages quand la valeur est « passée par référence »).

Remarques: Incohérences avec les mathématiques

- L’expression est souvent calculée entièrement avant que le résultat ne prenne place dans la case correspondant au nom de la variable, ce qui amène à des incohérences avec le monde des mathématiques.
- Le signe = en Python a un sens totalement différent de celui qu’il a en mathématiques.

Remarques: Incohérences avec les mathématiques

Texte à comparer	**Maths**	**Informatique (Python)**
x = 1	égalité, proposition comparant x et 1	c’est une affectation
x == 1	ne veut rien dire	expression qui compare x et 1, retourne souvent un booléen
x = x + 1	proposition toujours fausse	la valeur de la variable x est augmentée de 1 (incrémentation)

Exemple (à exécuter dans la console Python)

Console

```
>>> 0 = 0
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 0 == 0
True
>>> 1 == 0
False
```

Exemple (à exécuter dans la console Python)

Console

```
>>> x == 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 0
>>> x == 0
True
```

Exemple (à exécuter dans la console Python)

Console

```
>>> x == 1
False
>>> x = 0 == 0
>>> x
True
```

05

Types

Introduction

Les opérateurs travaillent souvent sur des valeurs du même type. Par exemple, `4 / 2` signifie quelque chose, alors que `Faux / Vrai` ne signifie rien. Nous aborderons en détails une liste non exhaustive des types que l'on peut rencontrer en informatique.

Booléens

Les booléens peuvent prendre deux valeurs.

Faux	Vrai	Français, Algo
0	1	Electr…
False	True	Python

Booléens

Il existe trois grands types d'opérations qu'on peut effectuer sur les booléens.

Opérations	**Algo**	**Python**
négation	non	not
disjonction	et	and
conjonction	ou	or

Exemple (à exécuter dans la console Python)

Console

```
>>> True and False
False
>>> (True or False) and (False or True)
```

Remarque: Polymorphisme en Python

En Python, nous verrons que n’importe quelle valeur peut servir (passer pour) un booléen. Par exemple, parmi les nombres, 0 est faux et tous les autres nombres sont vrais.

Entiers: opérations sur les entiers

Opérations	**Algo**	**Python**
Addition	+	+
Soustraction	-	-
Multiplication	*	*
Division (euclidienne ou entière)	div	/ ou //
Reste	mod	%
Puissance	2**3	2**3

Entiers: opérations sur les entiers

Incrémenter signifie « augmenter d'une valeur entière fixée », en général de 1. On peut par exemple incrémenter `n` grâce à l'instruction Python `n = n + 1`, que l'on peut abréger en `n += 1` (et non avec `n++` qui n'existe pas en Python).

Exemple (à exécuter dans la console Python)

Console

```
>>> n = 0
>>> n += 1
>>> n
1
>>> n *= 2
>>> n
2
```

Exemple (à exécuter dans la console Python)

Console

```
>>> n //= 2
>>> n
1
>>> n -= 1
>>> n
0
```

Exemple (à exécuter dans la console Python)

Console

```
>>> 2.5 + 2.5
5.0
>>> 2.0 - 2.
0.0
>>>  5/2
2
```

Exemple (à exécuter dans la console Python)

Console

```
>>> 5./2.
2.5
>>> 5//2
2
>>> 5.//2.
2.0
```

Les flottants: Attention à la précision des flottants

Les flottants ne permettent pas des calculs en précision arbitraire.

D’une part, leur taille est plafonnée, à la fois vers les grands nombres et vers les petits. D’autre part leur granularité est, disons, « subtile » (Voir exemples slides suivants).

Exemple (à exécuter dans la console Python)

Console

```
>>> 5./2
2.5
>>> 5/2.
2.5
>>> 2 + 2.
4.0
```

Exemple (à exécuter dans la console Python)

Console

```
>>> 2**52
4503599627370496
>>> 2.**52 + 1
4503599627370497.0
>>> 2**53
9007199254740992
```

Exemple (à exécuter dans la console Python)

Console

```
>>> 2.**53 + 1
9007199254740992.0
>>> 10**-322/10
1e-323
>>> 10**-323/10
0.0
```

Exemple (à exécuter dans la console Python)

Console

```
>>> 0.1*2
0.2
>>> 0.3
0.3
>>> 0.1*3
0.30000000000000004
```

Les flottants: Attention à la précision des flottants

Ils restent assez précis pour travailler par exemple avec des valeurs physiques, puisque par exemple la constante de gravité n’est connue qu'à 6 décimales près. En revanche, certains systèmes financiers préfèrent éviter de les utiliser.

Les flottants: de nouvelles opérations

De nouvelles opérations sont possibles (parfois uniquement grâce au type du résultat).

Exemple (à exécuter dans la console Python)

Console

```
>>> 5/2
2.5
>>> int(2.5)# D'autres fonctions existent pour la partie entière.
2           # Ici le résultat est de type « entier ».
>>> import math      # Certaines fonctions sont dans des modules.
>>> math.floor(2.5)
2
```

Exemple (à exécuter dans la console Python)

Console

```
>>> math.sqrt(2)     # racine carrée
1.4142135623730951
>>> math.exp(1)
2.718281828459045    # exponentielle
```

Caractères: les opérations

On peut passer d'un caractère à l'entier qui le code et inversement en utilisant respectivement les fonctions ord et chr. Attention, les résultats dépendent bien sûr de l'encodage.

En Python, la valeur booléenne d'un caractère est toujours True.

Exemple (à exécuter dans la console Python)

Console

```
>>> ord('A')
65
>>> chr(97)
'a'
>>> [chr(i) for i in range(32, 127)]
```

Chaînes de caractères

En Python, les chaînes de caractères sont des listes de caractères et héritent à ce titre de toutes les opérations liées aux listes (Voir slides à venir). Elles en ont encore beaucoup d'autres qui leur sont spécifiques (minuscules, majuscules, formater, découper selon les espaces…).

Exemple (à exécuter dans la console Python)

Console

```
>>> bonjour = "salut"
>>> txt = bonjour
>>> print(txt)
salut
>>> txt = "bonjour"
>>> print(txt)
bonjour
```

Exemple (à exécuter dans la console Python)

Console

```
>>> print(salut)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'salut' is not defined
```

Listes

Les listes contiennent une suite ordonnée (au sens où l'ordre est important) de valeurs.

Exemple (à exécuter dans la console Python)

Console

```
>>> len([False, True, False])
3
>>> sorted([3, 1, 2])
[1, 2, 3]
>>> sorted([3, 1, 2], reverse=True)
[3, 2, 1]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> [1, 2] + [3, 4]  # Concaténation (réutilisation du signe +).
[1, 2, 3, 4]
>>> [1, 2] + []     #[] est la liste vide et est l'élément neutre
[1, 2]                # de la concaténation.
>>> zip([1, 2], [True, False]) #«zip» signifie «fermeture éclair»
[(1, True), (2, False)]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> l = [1, 2]
>>> l.append(3)
>>> print(l)
[1, 2, 3]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> [False, True][1]
True
>>> l = [1, 2, 3, 4]
>>> l[0]    # contenu à la position 0
1
>>> l[1]    # à la position 1
2
```

Exemple (à exécuter dans la console Python)

Console

```
>>> l[-1]   # à la dernière position (-2 -> avant dernière…)
4
>>> l[1:3]  # sous-liste de la pos 1 jusqu'à avant la pos 3
[2, 3]
>>> l[1:]   # sous-liste à partir de la position 1
[2, 3, 4]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> l[:3]   # sous-liste avant la position 3 (strictement)
[1, 2, 3]
>>> l[:-1]  # sous-liste avant la dernière position (strictement)
[1, 2, 3]
>>> l[-2:]  # sous-liste à partir de l’avant-dernière position
[3, 4]
```

Listes

Dans certains langages, les listes sont appelées tableaux ou vecteurs, ont une taille fixe et ne peuvent parfois contenir que des valeurs de même type.

Exemple (à exécuter dans la console Python)

Console

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> range(1, 11, 4)
[1, 5, 9]
```

Exemple (à exécuter dans la console Python)

Console

```
>>> bool([])
False
>>> bool([False])
True
```

La liste vide passe pour le booléen Faux. Toutes les autres listes sont Vraies. Cela permet surtout de tester si une liste est vide de manière concise avec `if liste:` plutôt qu'avec `if len(liste) == 0`.

06

Notion de module

Notion de module

- Les fichiers contenant les programmes sont enregistrés avec l'extension .py. Ils sont appelés **module**.
- Un module doit être compris comme une boîte à outil regroupant des fonctions.
- En regroupant différents modules, on forme ce que l'on appelle **un package**.

Notion de module

- Le terme **library** que l'on traduit par **bibliothèque** (même si on trouve parfois l'anglicisme « librairie ») est un terme générique utilisé pour désigner tout regroupement de code conçu dans le but de pouvoir être utilisé par d'autres utilisateurs. Ainsi, tout module ou package publié sera communément appelé bibliothèque.

Notion de module

- **Python** est fourni de base avec une bibliothèque standard. Elle regroupe des dizaines de modules et permet de faire un grand nombre de tâches (allant de petits calculs mathématiques jusqu'à des communications réseaux ou des accès à des bases de données).

Bibliothèques classiques

Il est à noter qu'il existe une grande communauté de développeurs autour du langage Python. On peut donc facilement trouver, en libre accès sur internet, des bibliothèques publiées mises à disposition par des développeurs. Il est donc probable que ce que l'on cherche à faire en Python a déjà été produit par d'autres. L'usage répandu dans cette communauté est de récupérer ces bibliothèques mises à disposition, de construire dessus et d'éventuellement contribuer en publiant des ajouts ou des bibliothèques entières.

Bibliothèques classiques

Évidemment, le cadre du concours est un peu particulier. On s’attend à ce que vous ayez conçu tout votre code. On se limitera donc à charger quelques bibliothèques classiques. Plus précisément, on utilisera cette année les bibliothèques suivantes.

Bibliothèques classiques

- numpy: essentiellement, ce module fournit des fonctions permettant la manipulation efficace de tableaux. Le comportement de ces tableaux est défini par la classe ndarray (que l'on connaît aussi sous l'alias array).
- math: ce module permet d'avoir accès aux fonctions mathématiques usuelles telles que cos, √, sin, tan, , exp, . . .

Bibliothèques classiques

- random: module qui contient la fonction random qui implémente un générateur pseudo-aléatoire. On utilisera ce module dans le cadre de simulation de variable aléatoire réelle.
- matplotlib.pyplot: module qui regroupe des fonctions permettant d'effectuer des graphiques en tout genre.

Bibliothèques classiques

- scipy: package qui contient des outils scientifiques et numériques pour Python. Parmi ces outils, on trouve notamment un solveur d'équations différentielles, des fonctions pour de l'intégration numérique ou encore des outils de programmation parallèle. On utilisera essentiellement le module stats qui fournit les fonctions pour manipuler et simuler des variable aléatoire réelle suivant une loi normale.

Espaces de nom et import

Les espaces de nom font partie de la philosophie de Python.

- Afin de pouvoir profiter des fonctionnalités offertes par une bibliothèque, il faudra la charger dans votre environnement de travail. Pour ce faire, on utilise le mot clé import.

La syntaxe de base est la suivante: `import module`

ou, si le module se retrouve au sein d'un package:

```
import package.module
```

Espaces de nom et import

- Un module définit ce qu'on appelle un espace de nom. Une fois un module chargé, les variables et fonctions définies à l'intérieur d'un module sont alors accessibles via la syntaxe suivante: `module.variable`
- Si la bibliothèque que l'on charge est un package, la syntaxe pour accéder à une variable au sein d'un module est donc la suivante: `package.module.variable`

Espaces de nom et import

Cette syntaxe étant un peu lourde, on utilise la notion d'alias qui permet, lors du chargement d'un module, de le renommer.

```
import package.module as mod
```

Espaces de nom et import

L'intérêt d'un espace de nom est qu'il permet une isolation parfaite d'une variable au sein d'un module. Imaginons que l'on souhaite utiliser une variable var définie dans un module nommé module1 et que cette variable soit aussi définie (autrement !) dans un autre module module2 que l'on a chargé. Alors l'accès à cette variable se fera sans aucune ambiguïté : l'appel module1.var permettra d'accéder à la variable var définie au sein du premier module tandis que l'appel module2.var permet d'accéder à la variable au sein du deuxième module.

Danger du import *

L’utilisation de l’instruction `import *` au sein d’un module est fortement déconseillée. Plusieurs raisons à cela :

- Cela peut provoquer des conflits de nom de fonctions ou de variables. En effet, cela ajoute certains objets dont on ne voulait pas a priori.
- Cela peut s’avérer coûteux en temps. En effet, le nombre d’objets peut être très important.
- Cela ne permet plus de documenter précisément l’origine des fonctions utilisées.

Exemple (à exécuter dans la console Python)

Console

```
>>> import math
>>> math.sqrt(10)
>>> import numpy as np
>>> np.zeros(7)
```