PYTHON

Introduction à la programmation pour les biologistes

À QUOI SERVENT LES FONCTIONS?

Vous avez peut-être envisagé la possibilité de réutiliser un morceau de code avec quelques valeurs différentes. Au lieu de réécrire tout le code, il est beaucoup plus simple de définir une fonction, qui peut ensuite être utilisée à plusieurs reprises.

Le cas de nos TP

```
1. def tax(bill):
2.   """Adds 8% tax to a restaurant bill."""
3.   bill *= 1.08
4.   print "With tax: %f" % bill
5.   return bill
6.
7. meal_cost = 100
8. meal_with_tax = tax(meal_cost)
```

Exécuter

```
With tax: 108.000000hon!
```

STRUCTURE D'UNE FONCTION

Les fonctions sont définies avec trois composants:

- L'en-tête, qui comprend le mot-clé **def,** le nom de la fonction et tous les paramètres requis par la fonction.Voici un exemple:

```
def MyFunction ()
# Il n'y a pas de paramètres
```

- Le corps, qui décrit les procédures que la fonction effectue. Le corps est en retrait, tout comme les instructions conditionnelles.

```
print "Hello World!"
```

```
1. def MyFunction():
2. """Prints 'Hello World!' to the
3. console."""
   1. print "Hello World!

MyFunction ()
```

Exécuter

```
"Hello World!
```

PARAMÈTRES ET ARGUMENTS

- Les valeurs passés dans une fonction sont appelées les arguments.
- Un argument est une variable qui est une entrée dans une fonction. Il dit: "Plus tard, lorsque square sera utilisé, vous pourrez entrer n'importe quelle valeur."
- Mais pour l'instant, nous appellerons cette valeur future n

```
def MyFunction (n) :
```

```
1. def MyFunction(n):
    1. print " le n = " , n

   MyFunction (10)
```

Exécuter

```
Le n = 10
```

APPEL ET RÉPONSE

- Lors de la définition d'une fonction, les variables sont appelées paramètres.

- Après avoir défini une fonction, celle-ci doit être appelée pour être implémentée. Dans l'exercice précédent, MyFunction (10) dans la dernière ligne a dit au programme de rechercher la fonction appelée MyFunction et d'exécuter le code qu'il contient.

- Lorsque vous utilisez ou appelez une fonction, les entrées de cette fonction sont appelées arguments.

- Une fonction peut avoir un nombre quelconque de paramètres.

```
1. def MyFunction(n):
    1. print " le n = " , n

   MyFunction (10)
```

Exécuter

```
Le n = 10
```

PARAMÈTRES ET ARGUMENTS

- Le but des fonctions en général est de prendre des entrées et de retourner quelque chose.
- L'instruction return est utilisée lorsqu'une fonction est prête à renvoyer une valeur à l'appelant.
- L'instruction return provoque la fermeture de votre fonction et le renvoi d'une valeur à l'appelant.

```
1. def Carre(n):
    1. sqr = n ** 2
    2. print “ le carré de “+ n +” et " + sqr
    3. return sqr
2. Carre(10)
```

Exécuter

```
le carré de 10 et 100
```

FONCTION APPEL FONCTION

- Nous avons vu des fonctions permettant d'imprimer du texte ou de faire de l'arithmétique simple, mais ces fonctions peuvent être beaucoup plus puissantes que cela.

- Par exemple, une fonction peut appeler une autre fonction

```
def fun_one(n):
return n * 5

def fun_two(m):
return fun_one(m) + 7
```

```
1. def ajouterUn(n):
2.   return n + 1
3.
4. def etUnAutre(n):
5.   return ajouterUn(n) + 1
6. print etUnAutre(3)
```

Exécuter

PARAMÈTRES ET ARGUMENTS

- Premièrement, définissez une fonction appelée cube qui prend un argument appelé nombre. N'oubliez pas les parenthèses et les deux points!
- Faites que cette fonction retourne le cube de ce nombre (c'est-à-dire que ce nombre multiplié par lui-même et multiplié par lui-même une fois de plus).
- Définissez une deuxième fonction appelée by_three qui prend un argument appelé nombre.
- si ce nombre est divisible par 3, by_three doit appeler un cube (nombre) et renvoyer son résultat. Sinon, by_three devrait retourner False.

```
def cube (number):
  return number**3
def by_three(number):
  if number % 3 == 0:
   return cube(number)
  else:
   return False
```

Exécuter

le carré de 10 et 100

PARAMÈTRES ET ARGUMENTS

```
def list_function(x):
  return x
n = [3, 5, 7]
print list_function(n)
```

```
def double_first(n):
  n[0] = n[0] * 2
numbers = [1, 2, 3, 4]
double_first(numbers)
print numbers
```

```
number = 5
def my_function(x):
  return x * 3
print my_function(number)
```

```
m = 5
n = 13
def add_function(x,y):
  return x + y
print add_function(m, n)
```

```
n = "Hello"
def string_function(s):
  return s + 'world'
print string_function(n)
```

PARAMÈTRES ET ARGUMENTS

```
n = [3, 5, 7]
def list_extender(lst):
 lst.append(9)
 return lst
print list_extender(n))
```

```
def my_function(x):
  for i in range(0, len(x)):
    x[i] = x[i]
  return x
print my_function(range(3))
```

```
n = [3, 5, 7]
def print_list(x):
 for i in range(0, len(x)):
   print x[i]
print_list(n)
```

```
n = [3, 5, 7]
def double_list(x):
 for i in range(0, len(x)):
   x[i] = x[i] * 2
# Don't forget to return your new list!
 return x
print double_list(n)
```

PARAMÈTRES ET ARGUMENTS

```
m = [1, 2, 3]
n = [4, 5, 6]
def join_lists(x,y) :
        return x + y
print join_lists(m, n)
```

```
n = ["Michael", "Lieberman"]
def join_strings(words):
  result=""
  for item in words :
    result = result + item
  return result
print join_strings(n)
```

```
n = [3, 5, 7]
def total(numbers):
  result = 0
  for i in range (0,len(numbers)):
    result = result + numbers[i]
  return result
print total(n)
```

PARAMÈTRES ET ARGUMENTS

```
n = [[1, 2, 3], [4, 5, 6, 7, 8, 9]]


def flatten (lists):
  result=[]
  for numbers in lists:
    for item in numbers:
      result.append(item)
  return result
print flatten(n)
```

PARAMÈTRES ET ARGUMENTS

- Maintenant que vous comprenez ce que sont les fonctions et comment, examinons certaines des fonctions intégrées à Python (aucun module requis!).
- Vous connaissez déjà certaines des fonctions intégrées que nous avons utilisées avec des chaînes, telles que .upper (), .lower (), str () et len ().
- Celles-ci sont excellentes pour travailler avec des chaînes, mais qu'en est-il de quelque chose de plus analytique?

```
def biggest_number(*args):
  print max(args)
  return max(args)

def smallest_number(*args):
  print min(args)
  return min(args)

def distance_from_zero(arg):
  print abs(arg)
  return abs(arg)

biggest_number(-10, -5, 5, 10)
smallest_number(-10, -5, 5, 10)
distance_from_zero(-10)
```

PARAMÈTRES ET ARGUMENTS

```
1. print type(42)
2. print type(4.2)
3. print type('spam')
```

- Enfin, la fonction type () renvoie le type des données qu'elle reçoit sous forme d'argument. Si vous demandez à Python de procéder comme suit:

```
1. <type 'int'>
2. <type 'float'>
3. <type 'str'>
```

SUPPRIMER DES ÉLÉMENTS DES LISTES

- Pour une liste appelée n:
  - n.pop (index) supprimera l'élément à l'index de la liste et vous le retournera
  - n.remove (item) supprimera l'élément réel s'il le trouve
  - del (n [1]) est comme .pop en ce sens qu'il supprimera l'élément à l'index donné, mais ne le retournera pas

```
1. n = [1, 3, 5]
2. n.pop(1)   # Returns 3 (the item at index 1)
3. print n      # prints [1, 5]
```

```
1. n.remove(1) # Removes 1 from the list, # NOT the item at index 1
2.  print n # prints [3, 5]
```

```
1. del(n[1]) # Doesn't return anything
2. print n # prints [1, 5]
```

FIN DES LISTES

- La méthode count () renvoie le nombre de fois où obj apparaît dans la liste.

  Voici la syntaxe de la méthode count ()

```
1. st = "Python is awesome, isn't it?“
2. subst = "is“
3. count = st.count(subst)
4. print("The count is:", count)
```

```
1. stg = "Python is awesome, isn't it?“
2. substg = "i“
3. count = stg.count(substg, 8, 25)
4. print("The count is:", count)
```

RANGE
UNE FOIS POUR TOUT

- Supposons que vous vouliez construire une liste des nombres de 0 à 50 (inclus). Nous pourrions le faire assez facilement:

```
1. my_list = range(51)
2. my_list = range(1,51)
3. my_list = range(1,51,2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
50]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50]

[1, 3, 5, 7, 9, 11, 13, 15, 17,
19, 21, 23, 25, 27, 29, 31, 33,
35, 37, 39, 41, 43, 45, 47, 49]
```

LES FICHIER

FONCTION APPEL FONCTION

- Si vous voulez imprimer quelque chose dans la nouvelle ligne, faites comme ceci:

1. **Print** "Docteur \n house"

Console

1. Docteur
2. house

FONCTION APPEL FONCTION

- La méthode strip () renvoie une copie de la chaîne dans laquelle tous les caractères ont été supprimés à partir du début et de la fin de la chaîne (caractères d'espacement par défaut).

```
1. str = "0000000this is example....wow!!!0000000";
2. print str.strip( '0' )
```

Console

```
1. this is string example....wow!!!
```

- La méthode rstrip () ou lstrip () (r: right; l:left) renvoie une copie de la chaîne dans laquelle tous les caractères ont été supprimés **à la fin ou au début de la chaîne** (caractères d'espacement par défaut).

```
1. str = "     this is example....wow!!!     ";
2. print str.rstrip()
3. str = "8888this is example....wow!!!888";
4. print str.rstrip('8')
```

Console

```
this is example....wow!!!
88888888this is example....wow!!!
```

POUR QUOI LES FICHIER

Premièrement, une note rapide sur ce que nous entendons par texte.

- En biologie, nous avons de la chance, car de nombreux types de données avec lesquelles nous travaillons sont stockés dans des fichiers texte qui sont relativement simples à traiter avec Python.

- Parmi celles-ci, les principales sont bien sûr les données sur les séquences d'ADN et de protéines, qui peuvent être stockées sous divers formats.

- Mais il existe de nombreux autres types de données - lectures de séquençage, scores de qualité, SNP, arbres phylogénétiques, cartes lues, données d'échantillons géographiques, matrices de distances génétiques - auxquelles nous pouvons accéder à partir de nos programmes Python.

POUR QUOI LES FICHIER

Premièrement, une note rapide sur ce que nous entendons par texte.

- En programmation, lorsque nous parlons de fichiers texte, nous ne parlons pas nécessairement de quelque chose qui est lisible par l'homme.
- Nous parlons plutôt d'un fichier contenant des caractères et des lignes, quelque chose que vous pouvez ouvrir et afficher dans un éditeur de texte, que vous puissiez ou non donner un sens au fichier.
- Voici des exemples de fichiers texte que vous avez éventuellement rencontrés:
  - Fichiers FASTA de séquences d'ADN ou de protéines
  - fichiers contenant les sorties des programmes en ligne de commande (par exemple, BLAST)
  - Fichiers FASTQ contenant des lectures de séquençage d'ADN
  - Fichiers HTML
  - et bien sûr, le code Python lui-même

LIRE UN FICHIER TEXTE

- En Python, comme dans le monde physique, nous devons ouvrir un fichier avant de pouvoir lire ce qu'il contient.

```
f = open("dna.txt")
```

  - Cela a dit à Python d'ouvrir output.txt. Nous avons stocké le résultat de cette opération dans un objet fichier, f.

  - Python utilise la valeur par défaut, "r" pour la lecture.

- Vous pouvez ouvrir des fichiers dans l'un des modes suivants:
  - mode écriture seule ("w"). Si le fichier n'existe pas python le crée.
  - mode lecture seule ("r")
  - mode lecture et écriture ("r +"). Si le fichier n'existe pas python le crée.
  - mode append ("a"), pour une ouverture en mode ajout à la fin du fichier (Append). Si le fichier n'existe pas python le crée.

```
f = open("dna.txt", "w")
```

LIRE UN FICHIER TEXTE

- La première chose à faire est de lire le contenu du fichier.
- Il existe deux façons de lire un fichier :

1. Lire tout le contenu du fichier line par line dans une seul chaine de caractère.

- La méthode read () lit le contenu d'un fichier ouvert dans une chaîne que nous pouvons stocker dans une variable.

```
f = my_file.read()
```

- Une fois que le contenu du fichier est lu dans une variable, nous pouvons les traiter comme n'importe quelle autre chaîne. Par exemple, nous pouvons les imprimer:

```
1. my_file_name = "dna.txt"
2. my_file = open(my_file_name)
3. file_contents = my_file.read()
4. print(file_contents)
```

Ce qu'il est important de comprendre à propos de ce code, c'est qu'il existe trois variables distinctes qui ont des types différents et stockent trois choses très différentes:

1. **my_file_name** est une chaîne qui stocke le nom d'un fichier sur le disque.
2. **my_file** est un objet fichier et représente le fichier lui-même.
3. ***file_contents*** est une chaîne qui stocke le texte contenu dans le fichier.

dna.txt

ACTGTACGTGCACTGATC

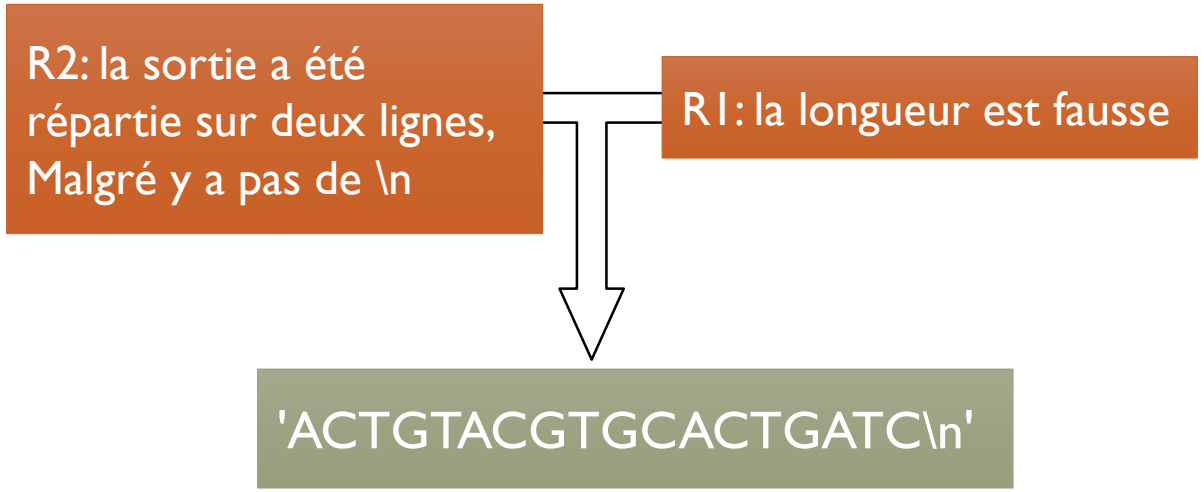
```
1.  # ouvrir le fichier
2.  my_file = open("dna.txt")

3.  # lire le contenu
4.  my_dna = my_file.read()

5.  # calculer la longueur
6.  dna_length = len(my_dna)

1.  # imprimer la sortie
2.  print("sequence is " + my_dna +  " and length is " + str(dna_length))
```

R1: la longueur est fausse

sequence is ACTGTACGTGCACTGATC
and length is 19

R2: la sortie a été répartie sur deux lignes, Malgré y a pas de \n

LIRE UN FICHIER TEXTE



- La même explication est donnée à ces deux problèmes:
  - Python a inclus le caractère de nouvelle ligne (\n) à la fin du fichier dna.txt dans le contenu.

```
1.  # ouvrir le fichier
2.  my_file = open("dna.txt")

3.  # lire le contenu
4.  my_dna = my_file.read().replace('\n', '')

5.  # calculer la longueur
6.  dna_length = len(my_dna)

1.  # imprimer la sortie
2.  print("sequence is " + my_dna +  " and length is
    " + str(dna_length))
```

LIRE UN FICHIER TEXTE LIGNE PAR LIGNE

- La première chose à faire est de lire le contenu du fichier.
- Il existe deux façons de lire un fichier :

2. Lire tout le contenu du fichier ligne par ligne, chaque ligne comme une seul chaine de caractère.

dna.txt

```
ACTGTACGT
GCACTGATC
```

- Nous pouvons utiliser la fonction readlines.

```
1. f = open('dna.txt', "r")
2. lines = f.readlines()
```

- La fonction readlines de Python lit tout le contenu du fichier texte et les place dans une liste de lignes après l'ouverture du fichier

```
[ 'ACTGTACGT\n' , ' GCACTGATC' ]
```

- Une autre façon de lire les lignes à la fois est d'utiliser simplement

```
1. f = open('dna.txt', "r")
2. lines = liste(f)
```

LIRE UN FICHIER TEXTE LIGNE PAR LIGNE

- Voici comment lire un fichier texte ligne par ligne en utilisant l'instruction «While» et la fonction readline de python.

- Comme nous lisons une ligne à la fois avec readline, nous pouvons facilement gérer de gros fichiers sans nous soucier des problèmes de mémoire.

```
1.  f = open('my_text_file.txt')

2.  # utilise readline () pour lire la première ligne
3.  line = f.readline()

# utilisez la ligne de lecture pour lire plus loin. Si
# le fichier n'est pas vide, continuez à lire
# une ligne à la fois jusqu'à ce que le fichier
# soit vide

1.  while line:
2.      print(line)

3.  # utilise readline () pour lire la prochaine ligne
4.  line = f.readline()

5.  f.close()
```

LIRE UN FICHIER TEXTE LIGNE PAR LIGNE

- Une autre variante de la lecture d'un fichier avec l'instruction while et l'instruction readline est la suivante.

- Ici, while teste booléen et lit ligne par ligne jusqu'à la fin du fichier et la ligne sera vide.

```
1.  fh = open('my_text_file.txt')

2.  while True:

3.      line = fh.readline()
4.      print line

        # vérifier si la ligne n'est pas vide
        if not line:
5.          break

6.  fh.close()
```

LIRE UN FICHIER TEXTE LIGNE PAR LIGNE

- Lire un fichier texte ligne par ligne à l'aide d'un itérateur en Python
- On peut également utiliser un itérateur pour lire un fichier texte ligne par ligne.
- Voici comment faire.

```
1. fh = open('my_text_file.txt')
2. for line in fh:
       print(line)
3. fh.close()
```

LIRE UN FICHIER TEXTE LIGNE PAR LIGNE

- Vous devez fermer le fichier.
- Vous faites cela simplement en appelant my_file.close ()
- Si vous ne fermez pas votre fichier, Python n'y écrira pas correctement.
- A partir de maintenant, vous devez fermer vos fichiers!

```
1. fh.close()
```

SPLIT

- À un moment donné, il se peut que vous deviez diviser une grande chaîne en petits morceaux ou en chaînes. C'est l'opposé de la concaténation qui fusionne ou combine des chaînes en une seule.

- Pour ce faire, vous utilisez la fonction split. Il divise ou décompose une chaîne et ajoute les données à un tableau de chaînes à l'aide d'un séparateur défini.

```
x = 'blue,red,green'
x.split(",")

['blue', 'red', 'green']
```

- Si aucun séparateur n'est défini lorsque vous appelez la fonction, les espaces seront utilisés par défaut. En termes plus simples, le séparateur est un caractère défini qui sera placé entre chaque variable.

LIRE UN FICHIER TEXTE

- Bon travail! Il est maintenant temps d'écrire des données dans un nouveau fichier .txt.
- Nous pouvons écrire dans un fichier Python comme ceci:

```
my_file.write("information")
```

- La méthode writeelines () écrit une séquence de chaînes dans le fichier. La séquence peut être généralement une liste de chaînes.

```
my_file.writelines(List)
```

1. fh = open("hello.txt","w")
2. fh.write("Put the text you want to add here")
3. fh.write("and more lines if need be.")
4. fh.close()

1. fh = open("hello.txt","w")
1. lines_of_text = ["One line of text here","and another line here","and yet another here","and so on and so forth"]
2. fh.writelines(lines_of_text)
3. fh.close()

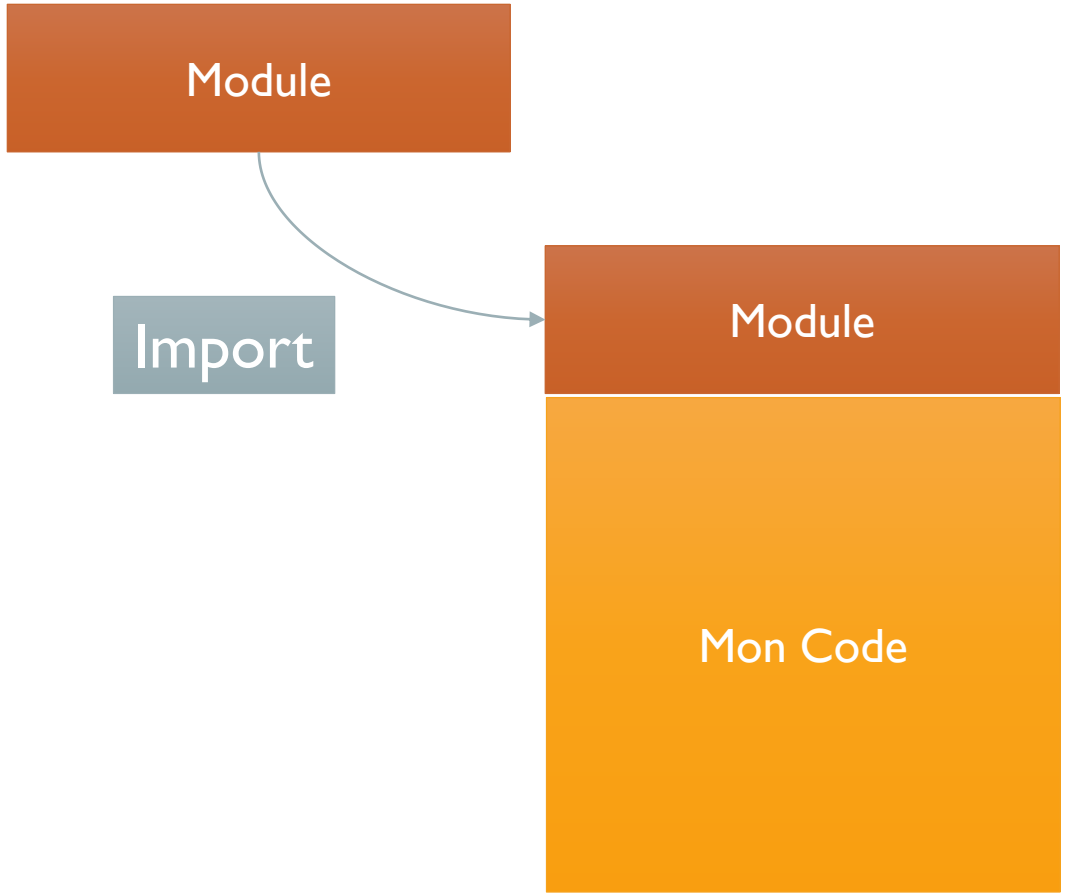
IMPORTATIONS

len("AAACGTR")

Les **fonctions intégrées** au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des **modules**.

Les modules sont donc des fichiers qui regroupent un ensembles de fonctions. Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python.

Pour utiliser des fonctions de modules dans un programme, il faut au début du fichier **importer** ceux-ci.

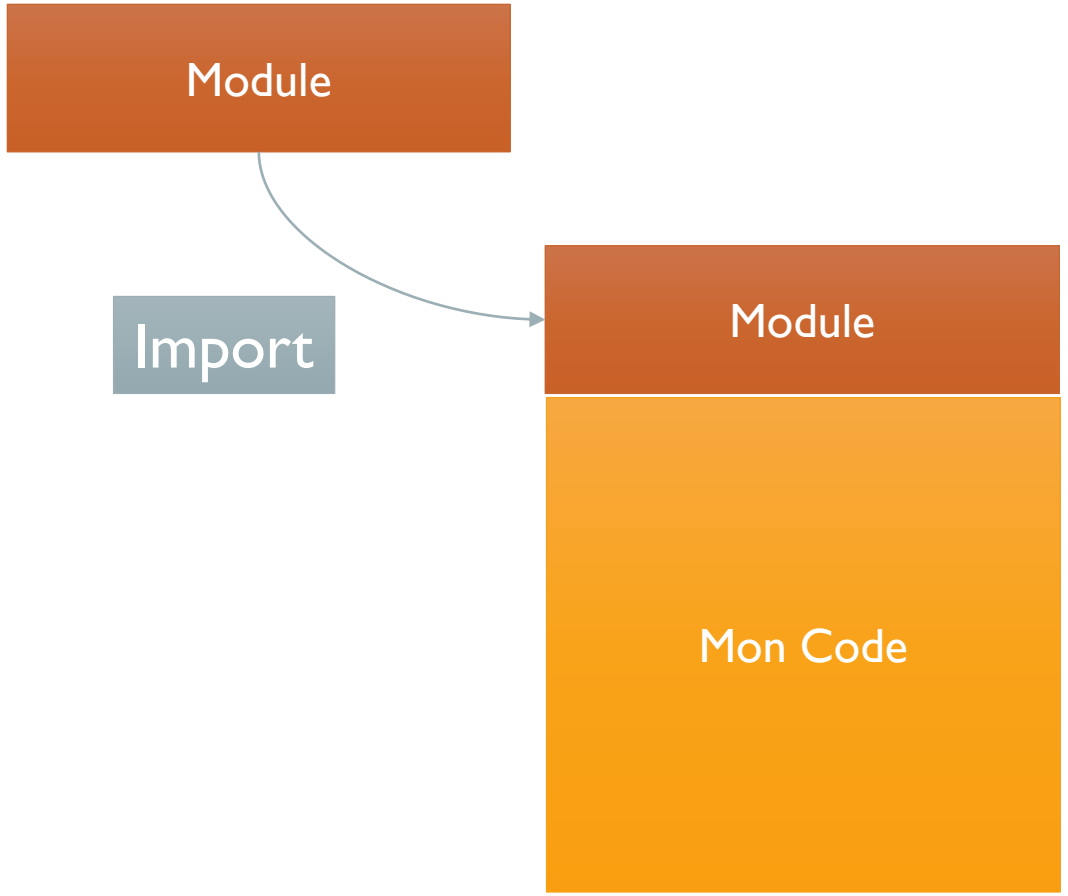


IMPORTATIONS: IMPORTATION GÉNÉRIQUE

Il existe un module Python nommé **math** qui inclut un certain nombre de variables et de fonctions utiles, et **sqrt ()** est l'une de ces fonctions. Pour accéder aux mathématiques, tout ce dont vous avez besoin est le mot-clé **import**. Lorsque vous importez simplement un module de cette façon, cela s'appelle une **importation générique**.

```
import math
print math.sqrt(25)
```

Pour utiliser des fonctions de modules dans un programme, il faut au début du fichier **importer** ceux-ci.



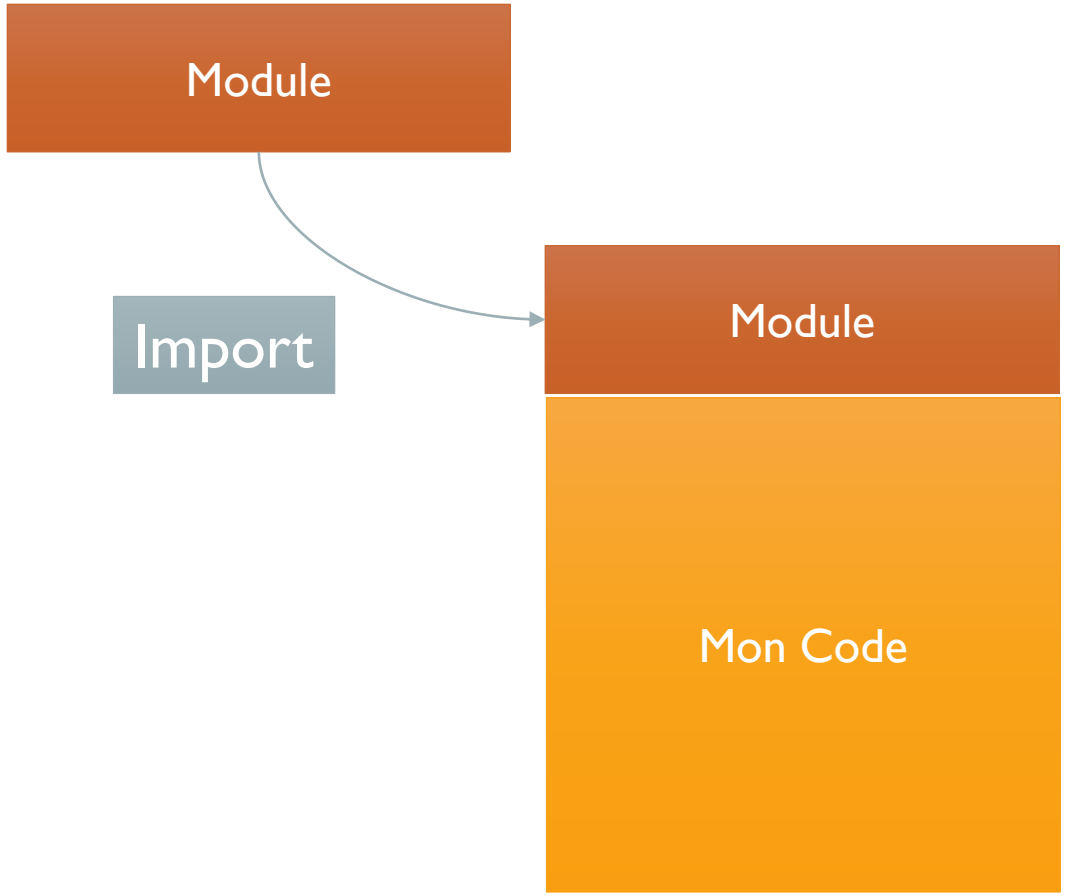
IMPORTATIONS DE FONCTIONS

Il est possible d'importer uniquement certaines variables ou fonctions d'un module donné. Extraire une seule fonction d'un module s'appelle une importation de fonction, et c'est fait avec le mot-clé **from**:

```
from math import sqrt
print sqrt (25)
```

Maintenant, vous pouvez simplement taper sqrt () pour obtenir la racine carrée d'un nombre - plus math.sqrt ()!

Pour utiliser des fonctions de modules dans un programme, il faut au début du fichier **importer** ceux-ci.

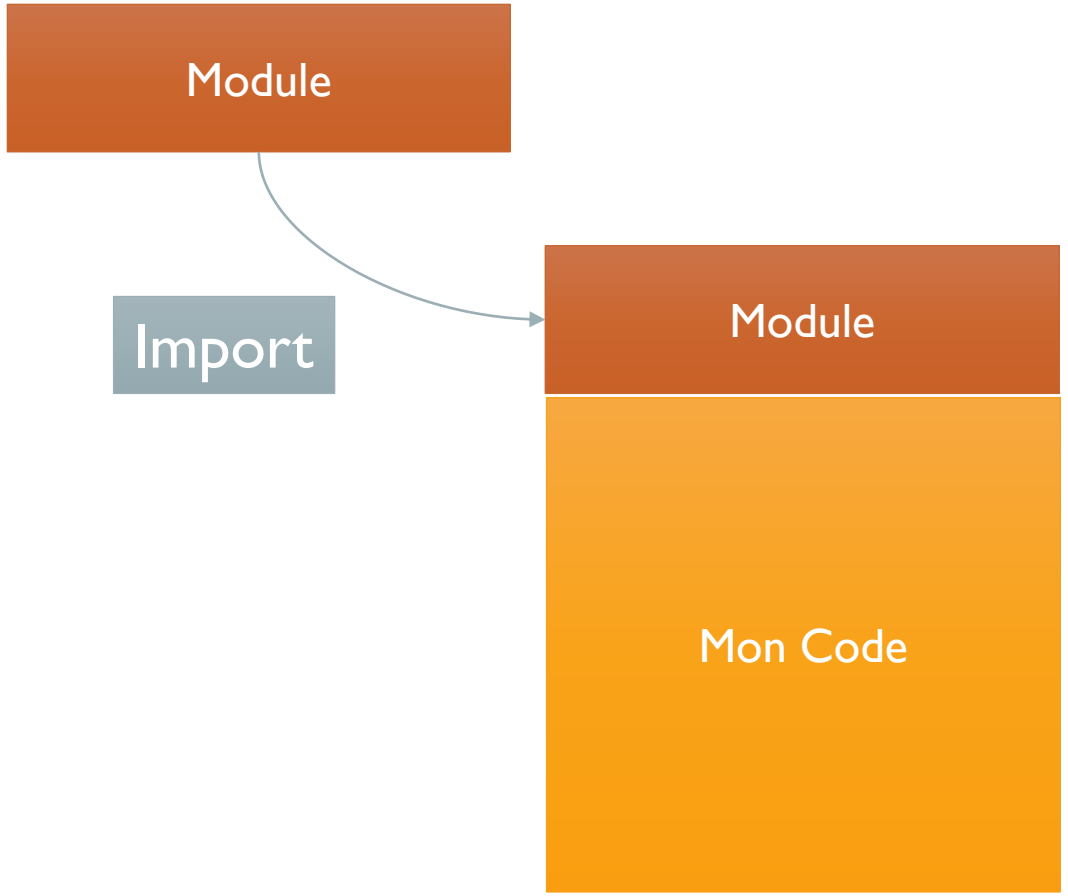


IMPORTATIONS UNIVERSELLES

Que faire si nous voulons toujours toutes les variables et les fonctions dans un module, mais ne veulent pas avoir à taper constamment des maths.?

```
from math import *
print sqrt (25)
```

Pour utiliser des fonctions de modules dans un programme, il faut au début du fichier **importer** ceux-ci.



IMPORT

- Vous souhaiterez peut-être modifier un nom de module pour l'abréger:

```
import [module] as [another_name]
```

- Dans le programme, nous appelons maintenant la constante pi m.pi plutôt que math.pi.

```
import math as m

print(m.pi)
```

- Pour certains modules, il est courant d'utiliser des alias. La documentation officielle du module matplotlib.pyplot appelle à l'utilisation de plt comme alias

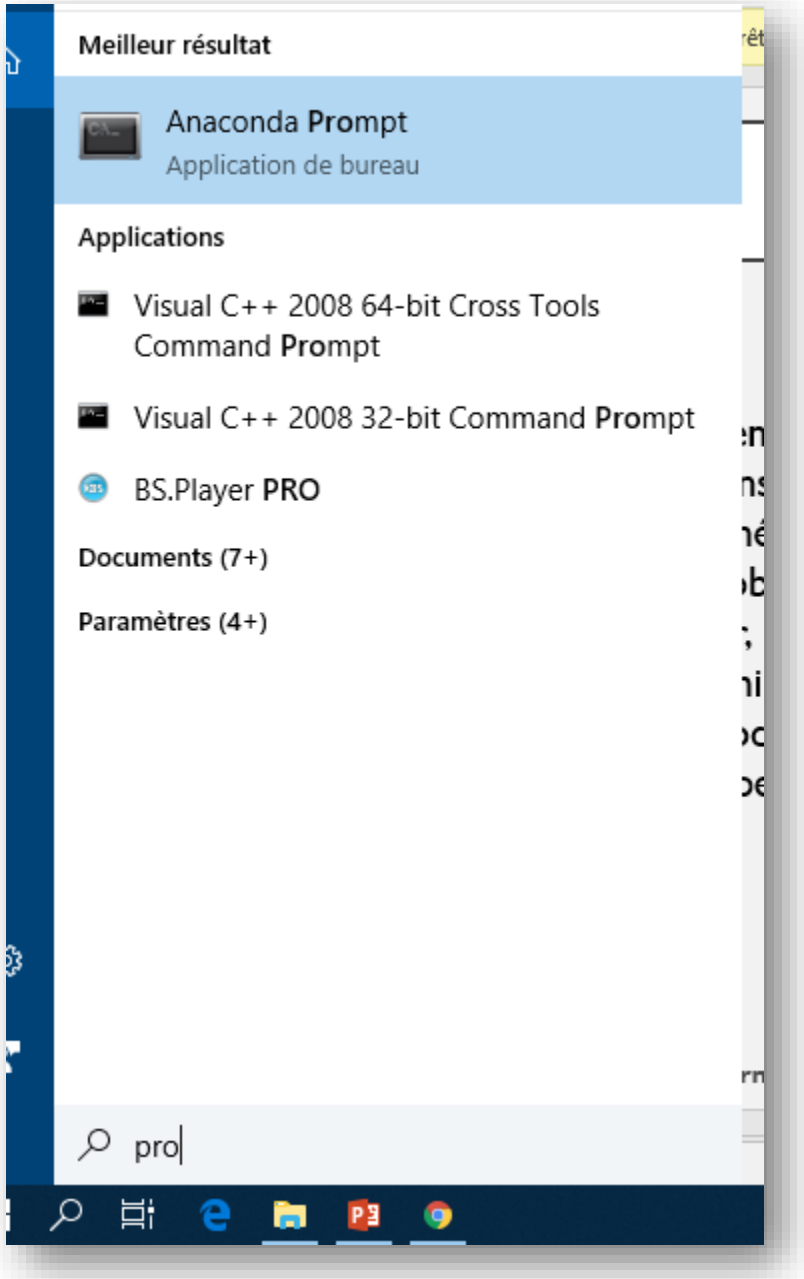
```
import matplotlib.pyplot as plt
```

- Cela permet aux programmeurs d'ajouter le mot court plt à n'importe laquelle des fonctions disponibles dans le module, comme dans plt.show ().

PYTHON ET PACKAGES

- Bien que les applications Python puissent être constituées d'un seul fichier, elles consistent généralement en une série de fonctions, d'objets (classes), d'outils pratiques et, bien sûr, de variables réparties sur plusieurs fichiers, placées à l'intérieur de modules. Ces modules constituent ensemble ce que l'on appelle un **package**.

- Pour installer d'autres packages non inclus dans Anaconda, il existe plusieurs solutions, pour les 3 méthodes présentées ici et adaptées à Anaconda, il faut lancer la fenêtre de commande Anaconda.



IMPORT

- conda et pip sont des outil de gestion de packages permettant d'installer des module scientifiques et analytiques écrits en Python.

- Dans la fenêtre command prompt Anaconda tapez :

```
pip install nom_package
```

- Cette commande marche si on 'est connecté a l'internet sinon pour installation manuelle nous devons télécharger le package sous format (.zip) puis utiliser les commande :

```
cd emplacement de package
python setup.py install

#setup.py est le prg d'installation de votre librairie contenu dans le directorie du package
```

IMPORT

- conda et pip sont des outil de gestion de packages permettant d'installer des module scientifiques et analytiques écrits en Python.

- Dans la fenêtre command prompt Anaconda tapez :

```
conda install nom_package
```

- C'est sous conda que l'on fait les mises à jour de Anaconda :

```
conda update Anaconda
```

IMPORT

- On a généralement besoin des librairies suivantes :
  - NumPy
  - SciPy
  - Matplotlib
  - Pandas
  - Biopython
  - Statsmodels

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


fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

Édition Recherche Source Exécution Déboguer Consoles Projets Outils Affichage Aide

C:\Users\Brahi\.spyder

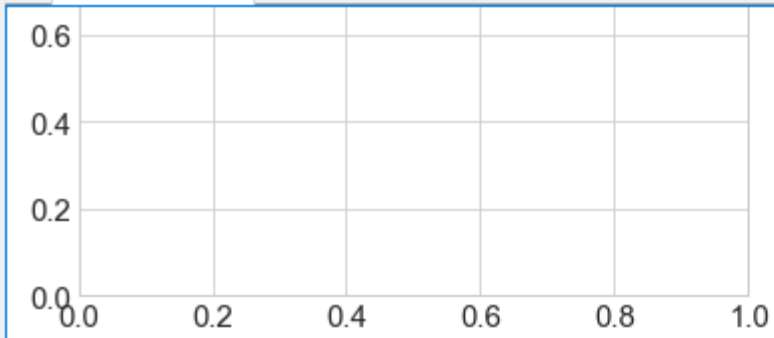
C:\Users\Brahi\.spyder\temp.py

emp.py

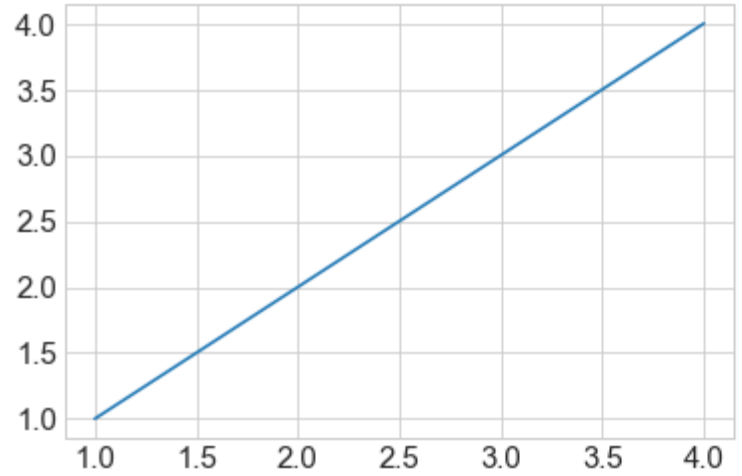
```
mport matplotlib.pyplot as plt
lt.style.use('seaborn-whitegrid')
mport numpy as np
 Chanegr la taille de police par défaut



 = 8
 = np.zeros(N)
rint y
1 = np.linspace(0, 10, N, endpoint=True)
rint x1
2 = np.linspace(0, 10, N, endpoint=False)

lt.plot([1,3,4], [1,3,4], '*')
```

Console IPython

Console 1/A




```
In [51]:

In [51]: runfile('C:/Users/Brahi/.spyder/temp.py', wdir='C:/Users/Brahi/.spyder')
[0. 0. 0. 0. 0. 0. 0. 0.]
[ 0.          1.42857143  2.85714286  4.28571429  5.71428571  7.14285714
  8.57142857 10.        ]
```




```
In [52]: runfile('C:/Users/Brahi/.spyder/temp.py', wdir='C:/Users/Brahi/.spyder')
```

Console IPython | Historique

Droits d'accès : **RW** | Fins de ligne : **CRLF** | Encodage : **UTF-8** | Ligne : **16** | Colonne : **30** | Mémoire : **6**

RÉFÉRENCES

- https://www.anaconda.com/download/
- http://www.programmingforbiologists.org/programming/
- https://www.codecademy.com/