Modélisation des Systèmes Physiques

Alessandro Torcini

LPTM

Université de Cergy-Pontoise

Le Cours

1. CM Mardi :14.00 - 16.00 (Amphithéatre E2) - Alessandro Torcini
2. TP Mercredi : 9.00-12.00 (Salle informatique B525) (30p) - M. Segneri
3. TP Mercredi : 14.00-17.00 (Salle informatique B530-B532) (30p) - M. Segneri

MonUCP - ENT

MonUCP - ENT de l'université de Cergy-Pontoise: Accueil - Mozilla Firefox

MonUCP
Environnement numérique de travail

Accueil | Culture | Administratif | Utilitaires | Pédagogie

ucp.net La lettre d'information interne

Retour

ARCHIVE LETTRES INFORMATION INTERNE

- ucp.net n°131 du 5 novembre 2018 (PDF – 572.2 ko)
  A la Une
  Transition Sociale et Environnementale au sein de l'UCP
- ucp.net 130 du 8 octobre 2018 (PDF – 362.1 ko)
  A la une
  Venez partager votre rêve d'Europe !
- Numéro 129 - 24 septembre 2018 (PDF – 212.7 ko)
  A la Une
  Programme 2018/2019 de l'Université Ouverte
- Numéro 128 - 10 septembre 2018 (PDF – 246 ko)
  A la Une
  Comité de site valorisation de la CoMUE : Appel à projets pour valoriser vos travaux
- Numéro 127 - 18 juin 2018 (PDF – 313.8 ko)

flash info

Retour

Plus d'archives

ANNÉES 2014-2018

- Numéro 45 du 21 novembre 2018 (PDF – 54 ko)
  Un nouveau nom pour notre université
- Numéro 44 du 8 novembre 2018 (PDF – 37.5 ko)
  En chemin vers le Grand Etablissement
- Numéro 43 du 25 octobre 2018 (PDF – 37.6 ko)
  Bilan de la rentrée universitaire 2018. Merci pour votre engagement !
- Numéro 42 du 29 mars 2018 (PDF – 101.2 ko)
  L'UCP signe une alliance académique et stratégique avec l'université de Warwick
- Numéro 41 du 16 février 2018 (PDF – 37.8 ko)
  La loi "Orientation et réussite étudiante"

MonUCP - ENT

Cours : TD Modélisation des systèmes physiques [TD] - Mozilla Firefox

https://cours.u-cergy.fr/course/view.php?id=3322

CoursUCP 2018-2019 · Mes cours · Besoin d'aide ? · Torcini Alessandro

Catalogue: Ressources, Activités, Inscriptions, Notes, Rapports, Blocs

Favoris

Administration du cours

TD Modélisation des systèmes physiques [TD]

Tableau de bord / TD Modélisation des systèmes physiques [TD]

- Description générale du cours
- Chapitre 1
- Chapitre 2
- Chapitre 3
- Chapitre 4
- Chapitre 5
- Chapitre 6
- Chapitre 7
- Chapitre 8

Programme du Cours

1. Le langage de programmation *Python*
2. Générateurs de Nombres Aléatoires
3. Statistique et Probabilités (Loi de Bernoulli, binomiale et normale)
4. Marche Aléatoire
5. Calcul d'intégrales
6. Recherche du zéro des fonctions
7. Introduction au chaos déterministe
8. Intégration numérique d'équations différentielles

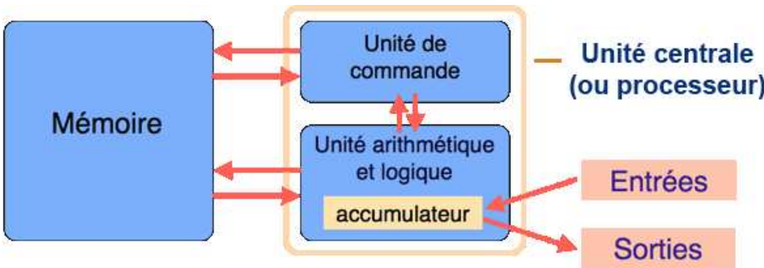
Contrôle Continu (CC) pendant le semestre et Examen Final (EF)

Vote Final = Max[EF,(CC+2*EF)/3]

Ordinateur

Un ordinateur consiste en

1. l'unité arithmétique et logique
2. l'unité de contrôle
3. la mémoire vive
4. les périphériques (y compris la mémoire permanente).



Normalement, l'unité arithmétique et logique et l'unité de contrôle sont intégrées au processeur (Central processing unit - CPU).
Si vous voulez effectuer un calcul ou une simulation, vous devez donner des instructions à l'unité de contrôle.
Le processeur marche avec un langage (code) machine, mais normalement on travaille avec un langage de haut niveau.

Langages

Les langages de haut niveau se divisent en

1. les langages compilés
2. les langages interprétés

Langage compilé (Fortran, C etc)

Un programme écrit dans un langage dit compilé va être traduit une fois pour toutes par un programme annexe, appelé compilateur, afin de générer un nouveau fichier qui sera autonome, c'est-à-dire qui n'aura plus besoin d'un programme autre que lui pour s'exécuter ; on dit d'ailleurs que ce fichier est exécutable.

Langage interprété (MatLab, Python, Mathematica etc)

Un programme écrit dans un langage interprété a besoin d'un programme auxiliaire (l'interpréteur) pour traduire les instructions du programme.

Un compilateur traduit un langage de haut niveau en code machine avant l'exécution, un langage interprète le fait en temps réel.

Python

Ici nous travaillerons avec le Python, créé début des années quatre-vingt dix par Guido van Rossum

1. Né en Hollande en 1956 (60 ans)
2. A travaillé pour Google pour dévelloper Python et après pour Dropbox
3. gvanrossum.github.io

Le nom Python vient de

Monthy Python's Flying Circus (BBC) 1960-70

Python en ligne en français

1. https ://openclassrooms.com/courses/apprenez-a-programmer-en-python
2. http ://www.science-emergence.com/Articles/Python-27-tutoriel/
3. https ://docs.python.org/fr/3.5/tutorial/

Python

Les avantages de Python sont :

- ⊕ Python est un langage interprété. Donc, il faut pas attendre une phase de compilation et on peut l'utiliser de manière interactive.
- ⊕ On peut réaliser des projets simples très rapidement.
- ⊕ Python existe pour tout les systèmes les plus importantes. Donc, votre logiciel (c'est l'ensemble des instructions) est multiplateforme, soit s'il marche sur un système, il fonctionne normalement aussi sans modifications sur des autres systèmes.
- ⊕ Il y a beaucoup des bibliothèques, y compris des bibliothèques numériques et des interfaces graphique (p. ex. pour créer des traces),
- ⊕ Python est libre. Donc, vous avez des outils à votre disposition pour lesquelles il faut parfois payer beaucoup d'argent.

Python

Bien sûr, Python a aussi des désavantages, comme :

- ⊖ Python est un langage interprété. Donc, Python peut être un peu lent. Toutefois, les bibliothèques sont normalement très efficaces et si eux sont chargés du plupart du travail, votre logiciel reste efficace.
- ⊖ (Peut-être) Python n'est pas le langage le plus joli. Plus précisément, Python est un langage très flexible. D'une côté, ça rend le début facile. De l'autre côté, il est possible que Python fait des choses que ne correspondent pas à vos attentes et donc il peut être difficile de trouver les erreurs dans votre logiciel.
- ⊖ Il y a quelques problèmes de compatibilité des bibliothèques avec l'interprète. Par conséquent, il faut parfois avoir plusieurs installations de Python sur un système et il faut faire attention laquelle on utilise.

On utilise le Python version 3.5

Démarrer en Python

Pour l'ordinateurs de UCP on utilise IDLE

Tapez Python , et la réponse est

```
Python 3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul 2 2016,
17:53:06) [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux Type
"help", "copyright", "credits" or "license" for more information.
```

On veut écrire Bonjour Paris !

```
>>> print("Bonjour Paris!")
Bonjour Paris!
>>> "Bonjour Paris!"
Bonjour Paris!
>>> 'Bonjour Paris!'
Bonjour Paris!
```

Les guillemets anglais « " » doit être utiliser pour une chaîne de caractéres

Calculer avec Python

Les opération numériques : - + * / **

Attention division en entiers

$$100//3 = 33 \qquad 5//6 = 0$$

Division en flottantes

```
>>> 100/3
33.333333333333336
```

Représentation approximative de flottantes en Python : double précision (float)

```
>>> float(100)/float(3)
33.333333333333336
```

Précision	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$	24 bits	environ 7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$	53 bits	environ 16

Le Opérateur Modulo

Le Modulo est defini par

$$m \ \mathrm{Mod}(n) = m - \left(\left[\frac{m}{n}\right] * n\right) \quad m, n \in Z$$

$[\cdot]$ Partie Entière

En Python %

>>> $9\%4$

$9 - \left(\left[\frac{9}{4}\right] * 4\right) = 9 - 2 * 4 = 1$

>>> $800\%6$

$800 - \left(\left[\frac{800}{6}\right] * 6\right) = 800 - 133 * 6 = 800 - 798 = 2$

>>> $325\%4$

1

Les Variables

Le modulo $325\%4$ fait avec les variables

```
>>> A=325
>>> B=4
>>> rapport=A/B
>>> AmoduloB = A -rapport*B
>>> print (AmoduloB)
1
```

Donner des valeurs à plusieurs variables au même temps :

```
>>> A,B=325,4
>>> rapport=A/B
>>> AmoduloB = A -rapport*B
>>> print (A,"modulo", B,"=",AmoduloB)
325 modulo 4 = 1
```

Les Variables

Une fois defińi, le valeur d'une variable peut être toujours modifier

```
>>> A,B=1,10
>>> A=A-1
>>> print (A,B)
0 10
>>> C=A*B
>>> print (A,B,C)
0 10 0
```

```
>>> A=1
>>> A -=1 équivalent à A = A -1
>>> print (A)
0
```

```
>>> A,B=1,2
>>> A *=B équivalent à A = B*A
>>> print (A)
2
```

Les Variables

L'ordre est important

```
>>> A,B=325,4
>>> A %=B équivalent à A = A%B
>>> print (A)
1

>>> A,B=325,4
>>> B %=A équivalent à B = B%A
>>> print (A)
325
>>> print (B)
4
```

Les Boucles

Les boucles servent à réaliser des répétitions

Probablement la construction la plus connue dans tous les langages de programmation est le `for`

En Python, un tel boucle se construit comme suit :

```
for variable in range(debut, fin, increment):
  operation1
  operation2
```

1. Il faut faire attention à l'indentation d'opérations parce que ça définit la domaine de la boucle.
2. Aussi les deux-points à la fin de la ligne avec le « `for` » sont importants.

Le code suivant affiche tous les nombres impairs entre 0 et 30 :

```
for n in range(1, 30, 2):
  print (n)
```

Si l'incrément est 1, on peut l'omettre (si on supprime aussi le `debut`, la séquence commence avec 0).

La Boucle `for`

Autres exemples

```
for n in range (1,3):
... print (n)
...
1
2
```

ATTENTION

Le `range` termine avant le dernier entier : 3-1 = 2

```
for k in range (-5,2):
... print (k)
...
-5
-4
-3
-2
-1
0
1
```

La Boucle `while`

Il y à un deuxième type de boucle : la boucle `while`. Elle se construit de la manière suivante :

```
while condition:
  operation1
  operation2
  operationN
```

1. L'effet est que toutes les opérations sont effectués aussi longtemps que la `condition` soit vraie
2. Comme dans le cas precedent, l'indentation sert à définir le domaine de la boucle.

```
sum=0
while sum < 4:
... sum=sum+1
... print (sum)
...
1
2
3
```

Opérateurs de comparaison

Évidemment, nous avons besoin de *conditions*. Normalement, elles ce construisent comme suit :

```
variable operateur valuer
```

Le résultat est vrai (`True`) ou faux (`False`)

Les opérateurs de comparaison sont :

opérateur	sens littéral
<	strictement inférieur à
>	strictement supérieur à
<=	inférieur ou égal à
>=	supérieur ou égal à
==	égal à
!=	différent de (aussi : <>)

La factorielle !

Prenons la factorielle pour illustrer la boucle `while` et les conditions :

```
n = 10
factorielle, temp = 1, n
while temp > 1:
  factorielle=factorielle*temp        Je multiplie par temp
  temp=temp-1                         Je soutrais 1
print (n, "! = ", factorielle)
```

```
n = 10
factorielle, temp = 1, n
while temp > 1:
  factorielle *= temp
  temp -= 1
print (n, "! = ", factorielle)
```

Erreurs avec la boucle `while`

```
a = 1/3.0
somme = 0
while somme != 20:
  print (somme)
  somme += a
```

représentation approximative des flottantes

```
n=0
while n==0:
  print (n)
  n += 1
```

erreur

Idle

Les structures conditionnelles

Les structures conditionnelles permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.
La construction générale est

```
if condition:
   operationVrai(s)
else:
   operationFausse(s)
```

1. Si la `condition` est vraie (`True`), Python exécute les `operationVrai(s)`,
2. Si elle est fausse (`False`), Python exécute les `operationFausse(s)`.
3. Notez encore une fois les indentations qui définissent la domaine des opérations correspondantes.
4. On peut supprimer la part `else` quand on n'a a pas besoin.

Les structures conditionnelles

P. ex. les lignes suivantes affichent si un nombre `n` est pair ou impair :

```
n = 1
if (n%2) == 0:
   print (n, "est pair")
else:
   print (n, "est impair")
```

Faisons-le pour tous les nombres entre 1 et 100 :

```
for n in range (1,101):
 if (n%2) == 0:
   print (n, "est pair")
 else:
   print (n, "est impair")
```

Notez ici aussi les indentations croissantes

Les variables booléennes

1. Le valeurs `True` et `False` définissent le *type booléen*
2. on peut les stocker dans des variable de type `bool`

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

```
>>> est = 2 < 3
>>> est
True
>>> est = 1 == 1
>>> est
True
>>> type(est)
<type 'bool'>
>>> est = 1 > 100
>>> est
False
```

Les variables booléennes

```
for n in range (1,101):
  est = (n%2) == 0   # La variable est sait si n est pair ou pas
  if est:
     print (n, "est pair")
  else:
     print (n, "est impair")
```

Les commentaires sont des messages qui sont ignorés par l'interprète et qui permettent de donner des indications sur le code.

En Python, un commentaire débute par un dièse (« # ») et se termine par le prochain saut de ligne.

Opérateurs booléens

Quelquefois, vous devez vérifier des conditions plus compliquées.
Pour ce but il y a les Opérateurs booléens

`and` (et) `or` (ou) `not` (négation)

1. `condition1 and condition2`
   est vrai si `condition1` et `condition2` sont vraies toutes les deux.
2. `condition1 or condition2`
   est vrai si une des `condition1` ou `condition2` est vraie.
3. `not condition`
   est vrai si la `condition` est fausse et faux si la condition est vraie.

Opérateurs booléens

Supposons que nous voulons savoir de chaque nombre entre 1 et 100 s'il est divisible par 2 *ou* par 3

```
for n in range (1,101):
 if (n%2) == 0 or (n%3) == 0:
   print (n, "est divisible par 2 ou par 3")
 else:
   print (n, "n'est divisible ni par 2 ni par 3")
```

ou plus précisément, vous pouvez utiliser `elif` (else if – autre si) :

```
for n in range (1,101):
  if (n%2) == 0:
     print (n, "est divisible par 2")
  elif (n%3) == 0:
    print (n, "est divisible par 3")
  else:
    print (n, "n'est divisible ni  par 2 ni par 3")
```

Vous avez observé qu'il y a beaucoup de nombres premiers sur la liste de nombres qui ne sont ni divisibles par 2 ni par 3

Résumé de la syntaxe du si

L'instruction conditionnelle en python dans sa version la plus générale possède la syntaxe suivante :

```
if <condition1> :
 <instructions_a_executer_si_la_condition1_est_vraie>
elif <condition2>:
 <instructions_à_exécuter_si_la_condition1_est_fausse
 _et_si_la_conditon2_est_vraie>
elif <condition3>:
 <instructions_à_exécuter_si_les_conditions_1_et_2_sont_fausses
 _et_si_la_conditon3_est_vraie>
...
else:
 <instuctions_à_exécuter_si_aucune_des_conditions_précédentes_n_est_vraie>
```

1. Il peut y avoir autant de `elif` qu'on le souhaite ;
2. on peut omettre la partie `else`. On peut omettre les `elif`.

break et continue

L'instruction `break` arrête la boucle la plus intérieure

```
for n in range(2, 10000):
    for x in range(2, n):
        if n % x == 0:
            print (n, 'égal', x, '*', n/x)
            break
    print (n, "est un nombre premier")
```

Faitez attention a l'indentation ! ! !

IDLE

L'instruction `continue` poursuit avec la prochaine itération de la boucle :

```
for num in range(2, 10):
    if num % 2 == 0:
        print ("nombre pair", num)
        continue
    print ("nombre", num)
```

Mesurer le temps d'exécution

Si vous voulez mesurer le temps d'exécution de votre programme, vous devez utiliser la fonction `time.clock()` :

```
import time
start = time.clock()

for n in range (1,101):
  if (n%2) == 0:
     print (n, "est divisible par 2")
  elif (n%3) == 0:
    print (n, "est divisible par 3")
  else:
    print (n, "n'est divisible ni  par 2 ni par 3")

runtime = (time.clock() - start)
print ("CPU temps d'execution: %.3f sec" % runtime  # temps pour l'execution)
```

IDLE

Les fonctions

Maintenant il faut commencer à organiser un peu les choses.
La factorielle peut être utile dans des contextes différentes et il est très ennuyant s'il faut la taper plusieurs fois.
On peut l'éviter avec les *fonctions*.
On crée une fonction selon le schéma suivant :

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, parametreN):
  operation1
  operation2
  operationM
return valeur
```

1. Le mot-clé `def` sert à *définir* la fonction.
2. La liste des paramètres est fournie lors d'un appel à la fonction. Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.
3. L'instruction serve à renvoyer une valeur, comme les fonctions en sens mathématique.

La fonction factorielle

```
def factorielle(n):          # definition de la factorielle
 resultat = 1
 while n > 1:
   resultat *= n
   n -= 1                    # ici nous n'avons plus besoin du contenu de n
 return resultat             # renvoyer le resultat
print ("4! = ", factorielle(4))          # Calculer 4!
print ("10! = ", factorielle(10))        # Calculer 10!
print ("100! = ", factorielle(100))      # Calculer 100!
```

Cet exemple montre aussi comment on utilise une fonction.

La fonction factorielle II

Si vous vous rendez compte que la factorielle satisfait à

$$n! = (n-1)! \times n \qquad \text{et} \qquad 0! = 1! = 1$$

On peut encore arriver à une version plus compacte :

```
def factorielle(n):           # definition de la factorielle
 if n<=1:
   return 1                   # 0! = 1! = 1
 return n*factorielle(n-1)    # n! = (n-1)! * n

print ("4! = ", factorielle(4))          # Calculer 4!
print ("10! = ", factorielle(10))        # Calculer 10!
print ("100! = ", factorielle(100))      # Calculer 100!
```

1. C'est un exemple pour une récursivité : la définition de la fonction contient la fonction même.
2. Naturellement, il faut faire attention que la récursivité se termine (essayez que se produit quand vous supprimez le test au début de la définition de la fonction `factorielle`).

Limites pour la récursivité

Vous auriez probablement des difficultés avec $n = 1000$.

La raison est que Python connaît des limites pour les récursivités. Vous pouvez modifier ces limites comme suit :

```
import sys
sys.setrecursionlimit(limite)
```

Maintenant Python peut être utilisé jusqu'à la `limite`

```
import sys
sys.setrecursionlimit(10000)
def factorielle(n):            # definition de la factorielle
 if n<=1:
   return 1                    # 0! = 1! = 1
 return n*factorielle(n-1)     # n! = (n-1)! * n


print ("  = ", factorielle(1000))      # Calculer 1000!
```

Variable locale et globale

1. Les variables locales sont modifiées dans au fonction, mais ils restent les mêmes hors de la fonction
2. Les variables globales sont modifiées partout

Idle

Il y a plusieurs options pour éviter la répétition des opérations que vous avez déjà faites.
Une utilise des variables globales.
Si vous rajoutez une ligne

```
global variable1,variable2
```

dans votre fonction, tous les `variable1,variable2` peuvent être modifiées dans la fonction

Les Modules

Les bibliothèques pour Python s'appellent modules.
D'abord, il faut les importer avec

```
import nom_de_module
```

Après, vous pouvez utiliser tous les fonctions et variables du module avec

`nom_de_module.fonction()` ou `nom_de_module.variable`

Parfois, on préfère un autre nom (p. ex. un nom plus court). Dans ce cas, on fait l'`import` comme suit :

```
import nom_de_module as autre_nom
```

Maintenant vous utilisez
`autre_nom.fonction()` ou `autre_nom.variable`.
Il y a encore une autre possibilité :

```
from nom_de_module import element(s)
# ou plus simple
from nom_de_module import *
```

Ceci importe seulement les `element(s)` (fonctions ou variables) donnés et vous pouvez les utiliser directement avec leurs noms (sans antéposer le « `nom_de_module.` »).

Le Module Math

Le module math est toujours disponible et contient des fonctions et constantes importantes, comme la racine, les fonctions trigonométriques et les constantes π et e. Par exemple, le module `math` contient une définition `math.sqrt(x)` pour la racine de x, et la constante π comme `math.pi`

```
import math                     # importer le module
valE1 = math.e                  # le nombre d'Euler
valE2 = math.exp(1)             # fonction exponentielle d'un
print ("e = ", valE1, "=", valE2) # devrait etre egal
```

Regardons aussi un exemple un peu plus élaboré avec un nom court pour le module :

```
import math as m                # importer le module comme "m"
valR2 = m.sqrt(2)               # la racine de 2
valSinPiQuart = m.sin(m.pi/4)   # sinus de pi/4
print ("sin(pi/4) = sqrt(2)/2 ?")
print (valSinPiQuart, valR2/2)    # devrait etre egal
valR3 = m.sqrt(3)               # la racine de 3
valCosPiSur6 = m.cos(m.pi/6)    # cosinus de pi/6
print ("cos(pi/6) = sqrt(3)/2 ?")
print (valCosPiSur6, valR3/2)     # devrait etre egal
```

Partie Entière

La partie entière d'un nombre flottant peut être obtenu de différentes façons

int() floor() ceil()

1. int() donne un nombre entier
2. floor() et ceil() donnent un nombre flottant
3. int(floor()) donne un nombre entier

```
from math import *
f=3.9
print ('int',f,int(f))
print ('ceil',f,ceil(f))
print ('floor',f,floor(f))
```

Le Module Numpy

NumPy est la bibliothèque fondamentale pour le calcul scientifique avec Python. Cet module contient entre autres l'algèbre linéaire.

Vous pouvez aussi l'utiliser pour remplacer le module `math`, voir l'exemple suivant :

```
import numpy as np                # importer le module comme "np"
valE1 = np.e                      # le nombre d'Euler
valE2 = np.exp(1)                 # fonction exponentielle d'un
print ("e = ", valE1, "=", valE2) # devrait etre egal
```

On aura probablement besoin d'autre fonctions de NumPy, mais pour l'instant je vous renvois

1. à la page d'accueil officiel de NumPy en anglais http ://www.numpy.org/
2. le chapitre « Introduction à PyLab » d'un cours en français www.courspython.com/introduction-pylab.html
3. introduction française a Numpy/Scipy/Matplotlib math.mad.free.fr/depot/numpy/essai.html

Tableaux - array()

Création

Les tableaux (en anglais, array) peuvent être créés avec array().

On utilise des crochets pour délimiter les listes d'éléments dans les tableaux.

```
from numpy import *
a = array([[1, 2, 3],[4, 5, 6]])
print(a)
[[1 2 3]
 [4 5 6]]
```

Accès aux éléments d'un tableau

```
from numpy import *
print (a[0,1])
2
print (a[1,2])
6
```

Faitez attention que le début de l'index est à partir de zéro

La fonction arange()

La fonction arange() crée une matrice (un tableau) qui contient les entiers compris entre 3 et 14 avec des étapes de 2

```
>>> m = arange(3,15,2)
>>> m
array([ 3,  5,  7,  9, 11, 13])
>>> print(m)
[ 3  5  7  9 11 13]
```

arange() accepte des arguments qui ne sont pas entiers.

```
>>> arange(0,11*pi,pi)
array([  0.        ,   3.14159265,   6.28318531,   9.42477796,
        12.56637061,  15.70796327,  18.84955592,  21.99114858,
        25.13274123,  28.27433388,  31.41592654])
```

on peut faire aussi des opérations d'algébre et des opérations avec les vecteurs

La fonction linspace()

linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments.

```
>>> linspace(3,9,10)
array([ 3.        ,  3.66666667,  4.33333333,  5.        ,  5.66666667,
        6.33333333,  7.        ,  7.66666667,  8.33333333,  9.        ])
```

Action d'une fonction mathématique sur un tableau

Dans ce cas, la fonction est appliquée à chacun des éléments du tableau.

```
>>> x = linspace(-pi/2, pi/2, 3)
>>> x
array([-1.57079633,  0.        ,  1.57079633])
>>> y = sin(x)
>>> y
array([-1.,  0.,  1.])
```

Tracer les courbes avec Matplotlib

Le module Matplotlib sert à tracer des courbes.
Un l'utilise surtout en combinaison avec NumPy. Commençons par tracer la fonction sinus :

```
import matplotlib.pyplot as plt # importer le module Matplotlib comme "plt"
import numpy as np              # importer le module NumPy comme "np"
x=np.arange(0, 2*np.pi, 0.01)   # valeurs sur l'abscisse
s=np.sin(x)                     # calculer les valeurs sur l'ordonnee
plt.plot(x, s)                  # creer le graphe
plt.xlabel("x")                 # appellation de l'abscisse
plt.ylabel("sin(x)")            # appellation de l'ordonnee
plt.xlim(0,2*np.pi)             # assurer des bonnes limites pour l'abscisse
plt.title("Fonction sinus")     # le titre du graphe
plt.show()                      # montrer la courbe
```

1. on crée avec `arange` les valeurs sur l'abscisse et calcule les valeurs correspondantes sur l'ordonnée,
2. Ces deux `array` (vecteurs) définissent le graphe de la fonction $\sin(x)$.
3. Après avoir fait la trace encore un peu plus joli, nous avons l'affiché sur l'écran.

Tracer les courbes

Je vous donne encore un exemple un peu plus avancé avec deux courbes qui sont marqués aves des points :

```
import matplotlib.pyplot as plt  # importer le module Matplotlib comme "plt"
from matplotlib.legend_handler import HandlerLine2D  # pour les legendes
from numpy import *                                  # importer le module NumPy
def fc(t):                                           # Une definition du fonction
  return(sqrt(1-sin(t)**2 +cos(t)**3))
x=linspace(0, pi/2, 200)                       # valeurs sur l'abscisse avec linspace
s=sin(x)                                        # calculer les valeurs de sinus
c=fc(x)                                              # et de la fonction
plt.plot(x, s, marker='o', label='sin')
# creer le graphe de sinus avec cercles
plt.plot(x, c, marker='v', label='fonction')
# creer le graphe fc avec triangles
plt.xlabel("x")                                     # appellation de l'abscisse
plt.xlim(0,pi/2)            # assurer des bonnes limites pour l'abscisse
plt.legend()                                        # afficher les legendes
plt.savefig("Figure1")                               # et stocker dans uns fichier
```

Tracer les courbes

1. Ici nous avons défini notre propre fonction fc()
2. L'exemple vous montre aussi comment on peut afficher des légendes.
3. À la fin d'exemple, nous avons stocké la figure dans le fichier `Figure1.png`.