esprit
Se former autrement

Formation PYHTON

Année universitaire : 2019-2020

United Nations Educational, Scientific and Cultural Organization
UNESCO Chair "Project-based learning" ESPRIT School of engineering, Tunisia
European Accreditation of Engineering Programmes EUR-ACE®
CONFÉRENCE DES GRANDES ÉCOLES
EXCELLENCE FOR A COMPLEX WORLD
cdio
CONCEIVE DESIGN IMPLEMENT OPERATE

Programmation

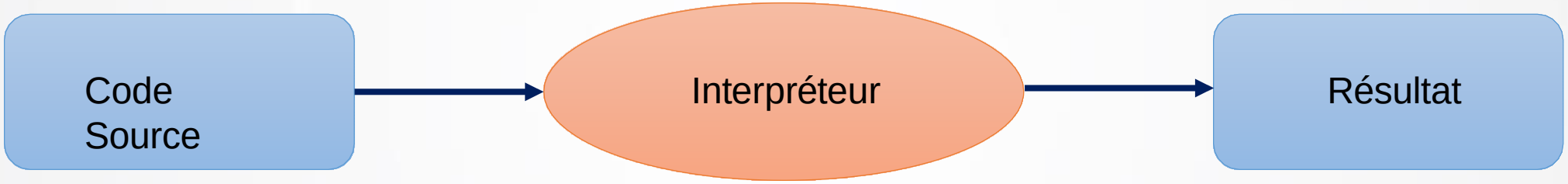
Programmation

- ***Programme :*** Suite d'instructions définies dans un langage donné.
- ***Langage machine :*** directement compréhensible par la machine.
- ***Langage d'assemblage (ou assembleur)*** : très facilement traduisible pour être compris par la machine.
- ***Langage de programmation*** : doit être compilé ou interprété pour être compris par la machine.

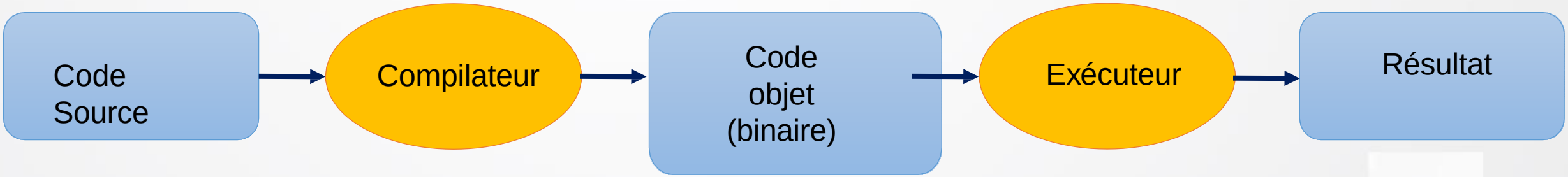
Programmation

Il existe deux méthodes pour effectuer la traduction du code source en langage machine :

- ***Interprétation***



- ***Compilation***



Mode compilé & mode interprété

Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Mode compilé & mode interprété

Les avantages d'un langage interprété sont :

- **la simplicité :** on ne passe pas par une étape de compilation avant d'exécuter son programme.
- **la portabilité :** un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre. Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété : la traduction à la volée de votre programme ralentit l'exécution bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

Votre premier programme sous Python

On va commencer par création d'un simple programme pour afficher « Hello World! »

En langage python on utilise la fonction **print** pour faire sortir ou afficher un texte :

```
>>> print  ( 'Hello World!')
Hello World!
```

« >>> » Ces trois chevrons signifient : « je suis prêt à recevoir les instructions ».

Les langages de programmation respectent une syntaxe claire. Si par exemple je doit afficher un message pour dire qu'on va commencer à développer, sans utiliser la fonction **print:**

```
>>> On va commencer pour développer avec Python
SyntaxError: invalid syntax
```

Les bases du langage Python

Python pour calculer

Addition, soustraction, multiplication, division:

- Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

- La division par Zéro en Python produit bien évidemment une erreur

```
>>> -5
-5
>>> 8
8
>>>  ( -5 + 3)  *9
-18
>>> 10/3
3.3333333333333335
>>> 8-5
3
>>> 11/0
Traceback  ( most recent call last)  :
  File "<pyshell#13>", line 1, in <module>
    11/0
ZeroDivisionError: division by zero
```

Python pour calculer

Division entière et modulo

- **Float** est utilisé en Python pour représenter les nombres qui ne sont pas des entiers.

Les ordinateurs ne peuvent pas stocker les types floats parfaitement, et on ne peut pas écrire un décimale complet (par exemple 1/3 (0.3333333333333333...).

- Pour déterminer le **quotient** et le **reste** d'une division, on utilise les opérateurs "**floor division**" *(//)* et "**modulo**" *(%)*

```
>>> 10 / 3
3.3333333333333335
>>> 10 // 3
3
>>> 10 % 3
1
>>>
```

Float

Quotient

Reste

*Les chaînes de caractères :*String

Pour entrer un texte dans un langage de programmation, on utilise le type **string.** Ce type de donnée permet de stocker une série de lettres.
On peut écrire une chaîne de caractères de différentes façons :

- ✓ Entre guillemets **(**"*ceci est une chaîne de caractères*")
- ✓ Entre apostrophes ('*ceci est une chaîne de caractères*')
- ✓Entre triples guillemets ("""*ceci est une chaîne de caractères*""")

Exemple:

```
>>> print  ( "Hello Python!")
Hello Python!
>>> print  ( 'Hello Python!')
Hello Python!
>>> print  ( """Hello Python""")
Hello Python
>>>
```

Les chaînes de caractères : String

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez chaine = 'J ‘aime le Python!', vous obtenez le message suivant :

```
>>> print  ( 'j'aime le python')

SyntaxError: invalid syntax
```

On insère ainsi un caractère antislash « \ » avant les apostrophes contenues dans le message.

```
>>> print ( ' j\'aime le  python ')

 j'aime le  python
>>>
```

Les chaînes de caractères : String

Python facilite le retour à la ligne sans écrire « \n » pour écrire des mots dans une chaine caractère avec retour à la ligne.

Créer une chaine de caractère string en utilisant **triples guillemets,** et la nouvelle ligne sera crée en tapant « entrer »

```
>>> print  ( ' Hello \n are you fine ?')

 Hello
 are you fine ?
>>> print  ( """Hello
are you fine?""")

Hello
are you fine?
>>>
```

Les chaînes de caractères : String

Python ne supporte pas un type de caractère, ils sont traités comme des chaînes de longueur « 1 », également considérées comme une sous-chaîne.

Nous utilisons des crochets pour le découpage avec l'index ou les indexer pour obtenir une sous-chaîne.

```
var1 = "Python!"

var2 = "Python Training"

print ("var1 [0]:", var1 [0])

print ("var2 [1: 6]:", var2 [1: 6])
```

Les chaînes de caractères : String

Vous pouvez mettre à jour Python String en réaffectant une variable à une autre chaîne. La nouvelle valeur peut être liée à la valeur précédente ou à une chaîne complètement différente.

```
x = "Bonjour Tout le monde!"

print (x [: 7])

print (x [0: 7] + " les developpeurs")
```

Les chaînes de caractères : String

Méthode Python String replace ()

La méthode replace () renvoie une copie de la chaîne dans laquelle les valeurs de l'ancienne chaîne ont été remplacées par la nouvelle valeur.

```
oldstring = 'Python Developper'

print (oldstring)

newstring = oldstring.replace ('Python', 'C')

print (newstring)
```

Les chaînes de caractères : String

Changer les chaînes majuscules et minuscules

En Python, vous pouvez même changer la chaîne en majuscule ou en minuscule.

```
string = "python à stc"

print (string.upper ())

string = "PYTHON À STC"

print (string.lower ())
```

Les chaînes de caractères : String

Utilisation de la fonction "join" pour la chaîne

La fonction ***join*** est un moyen plus souple de concaténer une chaîne. Avec la fonction join, vous pouvez ajouter n'importe quel caractère dans la chaîne.

Exemple:

```
print (":". join ("Python"))
```

Les chaînes de caractères : String

Chaîne d'inversion

En utilisant la fonction reverse, vous pouvez inverser la chaîne. Par exemple, si nous avons la chaîne « Python developper", puis si vous appliquez le code pour la fonction inverse, comme indiqué ci-dessous.

```
string = "Python developper"

print (''. join (reversed (string)))
```

Les chaînes de caractères : String

Startswith and **endswith**

```
x = "Hello world!"

print(x.startswith("Hello"))

print(x.endswith("Hi"))
```

Les entrées / Sorties (Inputs & outputs)

Chaque programme prend des entrées (Inputs) et produit des sorties (Outputs)

✓ **Output**

En Python, on utilise la fonction **print** pour obtenir la sortie. Cette fonction permet d'afficher une représentation textuelle d'une chose écrite.

✓ **Input**

Pour obtenir une entrée d'un utilisateur en Python, on utilise la fonction **input**

➔ Les fonction **print** et **input** ne sont pas très utilisées dans la console de Python qui fait la sortie et l'entrée,

Les opérations sur les chaines de caractères

Concatenation

- Comme pour les types **entier** et les types **float**, on peut associer des chaines de caractères (string) à l'aide d'une technique appelée **concatenation,** qu'on peut l'appliquer sur n'importe qulles deux chaines de caractères.
- On ne peut pas concaténer une chaine de caractères avec un nombre (entier, float…)
- Même si une chaîne de caractères contient des nombres, ils sont quand même ajoutés sous forme de chaînes plutôt que d'entiers.
- L'ajout d'une chaîne à un nombre produit une erreur, car même s'ils peuvent sembler similaires, il s'agit de deux types différents.

```
>>> print  ( "Python" + "," + "Training")

Python,Training
>>> print  ( "S" + "T" + "C")

STC
>>>
```

Les opérations sur les chaines de caractères

Multiplication

- Les chaînes peuvent être également multipliées par des entiers. Cela produit une version répétée de la chaîne d'origine.
- L'ordre de la chaîne et de l'entier n'a pas d'importance, mais la chaîne vient généralement en premier.
- Les chaînes ne peuvent pas être multipliées par des chaînes.
- Les chaînes ne peuvent pas être multipliées par des flottants, même dans le cas où les flottants sont des nombres entiers.

```
>>> print ( "Python"*3)

PythonPythonPython
>>> print ( 4 * '2')

2222
>>> print ( '17' * '35')

Traceback ( most recent call last) :
  File "<pyshell#54>", line 1, in <module>
    print ( '17' * '35')
TypeError: can't multiply sequence by non-int of type 'str'
>>> print ( 'Python' * 7.0)

Traceback ( most recent call last) :
  File "<pyshell#55>", line 1, in <module>
    print ( 'Python' * 7.0)
TypeError: can't multiply sequence by non-int of type 'float'
>>>
```

Conversion des Types

Multiplication

En Python, il est impossible d'effectuer certaines opérations en raison des types impliqués. Par exemple, on ne peut pas ajouter deux chaînes contenant les nombres 2 et 3 ensemble pour produire le nombre entier 5, car l'opération sera effectuée sur des chaînes, ce qui donne le résultat '23'.

➔ La solution à ce problème est la **conversion de type**.

Dans cet exemple, on utilise la fonction **int**.

```
>>> "2" + "3"

'23'
>>> int ("2") + int ("3")

5
>>>
```

Conversion des Types

Un autre exemple de conversion de type consiste à convertir les entrées utilisateur (**String**) en nombres (**Integer** ou **float**),

afin de permettre l'exécution des calculs.

```
>>> float ( input ( "Enter a number: ") )   + float ( input ( "Enter another number: ") )

Enter a number: 50
Enter another number: 2
52.0
>>>
```

➔ Passer des valeurs non-entières ou flottantes provoquera une erreur.

Les variables

- Les variables jouent un rôle très important dans la plupart des langages de programmation, et Python ne fait pas exception.
- Une variable vous permet de stocker une valeur en lui attribuant un nom, qui peut être utilisé pour faire référence à la valeur ultérieurement dans le programme.
- Pour assigner une variable, utilisez un **signe égal**. Contrairement à la plupart des lignes de code que nous avons examinées jusqu'à présent, il ne produit aucune sortie sur la console Python.

```
>>> x = 5

>>> print (x)

5
>>> print (x + 10)

15
>>>
```

Les variables

- Les variables peuvent être réaffectées autant de fois que vous le souhaitez afin de modifier leur valeur.
- En Python, les variables n'ont pas de types spécifiques. Vous pouvez donc affecter une **chaîne** à une **variable**, puis affecter un **entier** à la même **variable**.

```
>>> x = 7

>>> print (x)

7
>>> x = "Hello World!"

>>> print (x)

Hello World!
>>>
```

- Cependant, ce n'est pas une bonne pratique. Pour éviter les erreurs, Eviter l'affectation de la même variable avec différents types de données.

Les variables

- Essayer de référencer une variable qu'on n'a pas affecté provoque une erreur.
- On peut utiliser l'instruction **del** pour supprimer une variable, ce qui signifie que la référence du nom à la valeur est supprimée

```
>>> Formation = "Python"

>>> Formation

'Python'
>>> del Formation

>>> Formation

Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    Formation
NameError: name 'Formation' is not defined
>>> Formation = "Python"

>>> Formation

'Python'
>>>
```

Les variables

- Certains mots-clés de Python sont réservés, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.
- Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Les structures de controle

Les structures conditionnelles

Boolean

Un autre type de variable sous Python est le type **boolean**. Il existe deux valeurs booléennes: **True** et **False**. Ils peuvent être créés en comparant des variables, par exemple en utilisant l'opérateur égal ==.

```
>>> my_boolean = True

>>> my_boolean

True
>>> "Python" == "Python"

True
>>> "Training" = "Training"

SyntaxError: can't assign to literal
```

Les structures conditionnelles

Comparaison

- Un autre opérateur de comparaison, l'opérateur non égal (! =), Est évalué à **True** si les éléments comparés ne sont pas égaux et **False** s'ils le sont.
- Python a également des opérateurs qui déterminent si un nombre (float ou entier) est supérieur ou inférieur à un autre.

Ces opérateurs sont > et < respectivement.

```
>>> 2 != 2

False
>>> "Hello" != "Hi"

True
>>> 2 != 8

True
>>>
```

```
>>> 7 > 1

True
>>> 1 < 0

False
>>> 10 > 10

False
>>>
```

Les structures conditionnelles

- Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Mais nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas.

- Dans cette partie, on va parler des **structures conditionnelles**, qui vont nous permettre de faire des tests et d'aller plus loin dans la programmation.

- Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

Les structures conditionnelles

La structure conditionnelle « if »:

- On peut utiliser les instructions **if** pour exécuter du code si une condition donnée est remplie.
- Si une expression est évaluée à True, certaines instructions sont exécutées. Sinon, ils ne sont pas effectués.
- Une déclaration if ressemble à ceci:

```
if condition:
    instruction
```

Python utilise **l'indentation** (espace au début d'une ligne) pour délimiter des blocs de code. D'autres langages, tels que le C, utilisent des accolades pour accomplir cela, mais l'indentation en Python est obligatoire; les programmes ne fonctionneront pas sans cela.

Les structures conditionnelles

La structure conditionnelle « if »:

Exemple:

```
if a > 0:
    print ("a est supérieur à 0 ")
    if a < 0:
        print ("a est inférieur à 0 ")  print ("fin du programme")
```

Ecrire ce code sur votre IDE Python et afficher le résultat du programme

Les structures conditionnelles

La structure conditionnelle complète « if, elif et else »:

La première forme de condition que l'on vient de voir est pratique mais assez incomplète.

Considérons, par exemple, une variable a de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

```
>>> a = 5
>>> if a > 0:
        print("a est positif.")
    if a < 0:
        print("a est négatif.")
```

Amusez-vous à changer la valeur de a et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si a est égal à 0. En effet, aucune action n'a été prévue si a vaut 0.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable.

➔ La condition **if** est donc bien pratique mais insuffisante

Les structures conditionnelles

La structure conditionnelle complète « if, elif et else »:

✓ **L'instruction « else »**

Une instruction **else** suit une instruction **if** et contient le code appelé lorsque l'instruction **if** est évaluée à **False**.

Comme avec les instructions **if**, le code à l'intérieur du bloc doit être mis en retrait.

```
x = 4
if x == 5:
    print("Yes")
else:
    print("No")
```

Les structures conditionnelles

La structure conditionnelle complète « if, elif et else »:

On peut chaîner des instructions **if** et **else** pour déterminer quelle option d'une série de possibilités est vraie.

```
x= 7
if x == 5:
    print("Number is 5")
else:
    if x == 11:
        print("Number is 11")
    else:
        if x == 7:
            print("Number is 7")
```

```
Number is 7
```

Les structures conditionnelles

La structure conditionnelle complète « if, elif et else »:

✓ **L'instruction « elif »**

Le mot clé **elif** est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons de voir, l'idéal serait d'écrire :

- si a est strictement supérieur à 0, on dit qu'il est positif ;
- sinon si a est strictement inférieur à 0, on dit qu'il est négatif ;
- sinon, (a ne peut qu'être égal à 0), on dit alors que a est nul.

```
a=-14
if a>0:
    print("a est positif.")
elif a < 0:
    print("a est négatif.")
else: # Nul
    print("a est nul.")
```

```
a est négatif.
```

Les structures conditionnelles

La structure conditionnelle complète « if, elif et else »:

✓ **Les opérateurs de comparaison**

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Les structures conditionnelles

Logique booléenne:

- La logique booléenne est utilisée pour créer des conditions plus complexes pour les instructions qui reposent sur plusieurs conditions.
- Les opérateurs booléens de Python sont: **and**, **or**, and **not**.
- L'opérateur « **and** » prend deux arguments, et évalue comme True si, et seulement si, ses deux arguments sont True. Sinon, la valeur est False.
- L'opérateur « **or** » prend également deux arguments. Il est évalué sur True si l'un de ses arguments (ou les deux) est True et sur False si les deux arguments sont False.
- Contrairement aux autres opérateurs que nous avons vus jusqu'à présent, « **not** » prend un seul argument, et l'inverser. Le résultat de **not True** est **False** et **not False** devient **True**.

Les structures conditionnelles

Logique booléenne:

Exemple:

```
>>> 2 == 2

True
>>> "Python" == "Python"

True
>>> not 2 == 3

True
>>> "Python" == "Python" and "Code" == "Training"

False
>>> "Python" == "Python" or "Code" == "Training"

True
>>> not 1 > 7

True
>>> not 2 < 5

False
>>>
```

Les structures conditionnelles

Priorité des opérateurs

- La priorité des opérateurs est un concept très important en programmation. C'est une extension de l'idée mathématique d'ordre des opérations (la multiplication étant effectuée avant l'addition, etc.) pour inclure d'autres opérateurs, tels que ceux de la logique booléenne.
- Le code ci-dessous montre que **equal** (= =) a une priorité supérieure à **or** :

```
>>> False == False or True

True
>>> False == True or False

False
>>> False ==  ( True or False)

False
>>>  ( False == False)   or True

True
>>>
```

Les structures conditionnelles

➢ Le tableau suivant liste tous les opérateurs de Python, de la priorité la plus élevée à la plus basse.

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR'
|	Bitwise 'OR'
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparison operators, equality operators, membership and identity operators
not	Boolean 'NOT'
and	Boolean 'AND'
or	Boolean 'OR'
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators

Les boucles

Boucle while

- Une instruction **if** est exécutée une fois si sa condition est évaluée à True et jamais si elle est évaluée à False.
- Une instruction **while** est similaire, mais elle peut être exécutée plusieurs fois. Les instructions qu'il contient sont exécutées à plusieurs reprises, tant que la condition est vraie. Une fois la valeur False évaluée, la section de code suivante est exécutée.
- La syntaxe de **while** est :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

- Le code suivant présente une boucle **while** contenant une variable dont le nombre va de 1 à 6, point auquel la boucle se termine.

```
>>> i = 1

>>> while i <=6:
    print ( i)
    i = i + 1


1
2
3
4
5
6
>>>
```

Les boucles

Boucle while

Boucle infinie :

- La boucle infinie est un type spécial de boucle **while**; elle ne s'arrête jamais de courir. Sa condition reste toujours vraie.

- Un exemple de boucle infinie:

```
while 2==2:

print("Hello world!")
```

- On peut arrêter l'exécution du programme en utilisant le raccourci Ctrl-C ou en fermant le programme.

Les boucles

Boucle while

break:

Pour terminer prématurément une boucle **while**, on peut utiliser l'instruction break.

Lorsqu'elle est rencontrée à l'intérieur d'une boucle, l'instruction break entraîne son achèvement immédiat.

```
>>> i = 0

>>> while 1==1:
    print (i)
    i = i + 1
    if i >= 5:
        print ("Breaking")
        break


0
1
2
3
4
Breaking
>>>
```

Les boucles

Boucle while

- Exécuter ce code sur votre machine

```
while 1: # 1 est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

Les boucles

Boucle while

continue:

Le mot-clé **continue** permet de continuer une boucle, en repartant directement à la ligne du **while** ou **for**. Un petit exemple s'impose :

```
>>> i = 0

>>> while True:
    i = i +1
    if i == 2:
        print ( "Skipping 2")
        continue
    if i == 5:
        print ( "Breaking")
        break
    print ( i)


1
Skipping 2
3
4
Breaking
>>>
```

➔ **continue** est une declaration qui termine l'itération en cours et continue avec la suivante

Les boucles

Boucle while

- Exécuter ce code sur votre machine

```
i = 1
while i < 30: # Tant que i est inférieure à 30
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i est maintenant égale à", i)
        continue # On retourne au while sans exécuter les autres Lignes
    print("la variable i = ", i)
    i += 1
```

Les listes

- Les listes sont un autre type d'objet en Python. Ils sont utilisés pour stocker une liste indexée d'éléments.
- Une liste est créée en utilisant des crochets avec des virgules séparant les éléments.
- Vous pouvez accéder à certains éléments de la liste en utilisant son index entre crochets.

Exemple:

```
>>> Python = ["LangageScript", "Portable",  "Extensible", "Modulable", "Orienté Objet"]

>>> print  ( Python[0])

LangageScript
>>> print  ( Python[1])

Portable
>>> print  ( Python[2])

Extensible
>>> print  ( Python[3])

Modulable
>>> print  ( Python[4])

Orienté Objet
>>>
```

Liste vide

```
>>> empty_list = []

>>> print ( empty_list)

[]
>>>
```

Les listes

Les operations sur les listes

- En règle générale, une liste contient des éléments d'un seul type, mais il est également possible d'inclure plusieurs types différents.

- Les listes peuvent également être imbriquées dans d'autres listes.

```
>>> num = 3

>>> things = ["string", 0, [1, 3, num], 3.17, ["hello", "python", 1 ,3]]

>>> print ( things[0])

string
>>> print ( things[1])

0
>>> print ( things[2])

[1, 3, 3]
>>> print ( things[3])

3.17
>>> print ( things[4])

['hello', 'python', 1, 3]
>>>
```

```
>>> print ( things[4][0])

hello
>>> print ( things[4][1])

python
>>> print ( things[4][2])

1
>>>
```

Les listes

Les operations sur les listes

- L'indexation en dehors de limite des valeurs possibles de liste provoque **IndexError**.
- Certains types, tels que les chaînes, peuvent être indexés comme des listes. Les chaînes d'indexation se comportent comme si on indexe une liste contenant chaque caractère de la chaîne.
- Pour d'autres types, tels que les entiers, leur indexation est impossible et provoque une erreur TypeError.

```
>>> str = "Hello world"

>>> print ( str[6])

w
>>> print ( str[5])


>>> print ( str[2])

l
>>> print ( str[15])

Traceback (most recent call last) :
  File "<pyshell#242>", line 1, in <module>
    print ( str[15])
IndexError: string index out of range
>>>
```

Les listes

Les operations sur les listes

- L'élément à un certain index dans une liste peut être réaffecté.
- Les listes peuvent être ajoutées et multipliées de la même manière que les chaînes.

```
>>> things = ["string", 0, [1, 3, num], 3.17, ["hello", "python", 1 ,3]]

>>> things [1] = 6

>>> print ( things)

['string', 6, [1, 3, 3], 3.17, ['hello', 'python', 1, 3]]
>>> Adding = things + [1,2,3]

>>> print  ( Adding)

['string', 6, [1, 3, 3], 3.17, ['hello', 'python', 1, 3], 1, 2, 3]
>>> Multi = things * 2

>>> print  ( Multi)

['string', 6, [1, 3, 3], 3.17, ['hello', 'python', 1, 3], 'string', 6, [1, 3, 3], 3.17, ['hello', 'python', 1, 3]]
>>>
```

Les listes

Les operations sur les listes

- Pour vérifier si un élément est dans une liste, l'opérateur in peut être utilisé. Il renvoie **True** si l'élément apparaît une ou plusieurs fois dans la liste, et **False** sinon..
- Pour vérifier si un élément ne se trouve pas dans une liste, on utiliser l'opérateur **not**

```
>>> things = ["string", 0, [1, 3, num], 3.17, ["hello", "python", 1 ,3]]

>>> print ( 0 in things)

True
>>> print ( 6 in things)

False
>>> print ( 3 in things)

False
>>>
```

```
>>> things = ["string", 0, [1, 3, num], 3.17, ["hello", "python", 1 ,3]]

>>> print ( not 3 in things)

True
>>>
```

Les listes

Les functions appliquées sur les listes

- Une autre façon de modifier les listes consiste à utiliser la méthode **append**. Cela ajoute un élément à la fin d'une liste existante
- Pour obtenir le nombre d'éléments dans une liste, on peut utiliser la fonction **len**
- La méthode **insert** est similaire à append, sauf qu'elle permet d'insérer un nouvel élément à n'importe quelle position de la liste, non seulement à la fin.
- La méthode **index** recherche la première occurrence d'un élément de liste et renvoie son index. Si l'élément ne figure pas dans la liste, il génère ValueError.

Les listes

Les fonctions appliquées sur les listes

➢ Il existe quelques fonctions et méthodes utiles pour les listes.

- max (list) : Renvoie l'élément de liste avec la valeur maximale
- min (list) : Renvoie l'élément de liste avec la valeur minimale
- list.count (obj) : Retourne le nombre de fois qu'un élément apparaît dans une liste
- list.remove (obj) : Supprime un objet d'une liste
- list.reverse () : Inverse les objets d'une liste

Les listes

Les fonctions appliquées sur les listes

- La fonction **range** crée une liste séquentielle de nombres.
- Si **range** est appelé avec un argument, il produit un objet avec des valeurs allant de 0 jusqu'à cet argument.
- S'il est appelé avec deux arguments, il produit des valeurs du premier au second.
- La fonction **range** peut avoir un troisième argument, qui détermine l'intervalle de la séquence produite. Ce troisième argument doit être un entier.

Les listes

Les fonctions appliquées sur les listes

all et any

```
nums = [100, 33, 10, 22, 25]

if all([i > 8 for i in nums]):
    print("All larger than 5")

if any([i % 2 == 0 for i in nums]):
    print("At least one is even")

for j in enumerate(nums):
    print(j)
```

Les listes

Boucle for

- Parfois, on doit exécuter du code sur chaque élément de la liste. C'est ce qu'on appelle l'itération, et cela peut être accompli avec une boucle **while** et une variable counter.

```
>>> words = ["hello", "world", "spam", "eggs"]

>>> counter = 0

>>> max_index = len ( words)  - 1

>>> while counter <= max_index:
    word = words[counter]
    print ( word + "!")
    counter = counter + 1


hello!
world!
spam!
eggs!
>>>
```

Les listes

Boucle for

- L’itération dans une liste à l'aide d'une boucle **while** nécessite pas mal de code. Python fournit donc la boucle **for** sous la forme d'un raccourci permettant d'obtenir le même résultat.

- Le même code de l'exemple précédent peut être écrit avec une boucle for, comme suit:

```
>>> words = ["hello", "world", "spam", "eggs"]

>>> for word in words:
  print ( word + "!")
```

Les listes

Boucle for

- La boucle for est couramment utilisée pour répéter du code un certain nombre de fois. Ceci est réalisé en combinant des boucles **for** avec **range** des objets.

```
>>> for i in range ( 5) :
  print ( "hello!")
```

- Exécuter ce code sur votre machine et observez le résultat

- On n’a pas besoin d'appeler la liste pour la plage des objets (**range)** lorsqu'il est utilisé dans une boucle for, car il n'est pas indexé. Par conséquent, une liste n'est pas obligatoire.

Les dictionnaires

- Les dictionnaires sont des structures de données utilisées pour mapper des clés arbitraires en valeurs.
- Les listes peuvent être considérées comme des dictionnaires avec des clés de type entier dans une certaine plage.
- Les dictionnaires peuvent être indexés de la même manière que les listes, en utilisant des crochets contenant des clés.
- Exemple:

```
ages = {"Ahmed": 24, "Aymen": 42, "Mehdi": 58, "Mohamed": 35}
print(ages["Ahmed"])
print(ages["Mohamed"])
```

- Essayer d'indexer une clé qui ne fait pas partie du dictionnaire renvoie une erreur **KeyError**.
- Essayer d'indexer une clé qui ne fait pas partie du dictionnaire renvoie une erreur KeyError.

Les tuples

- Les tuples ressemblent beaucoup aux listes, sauf qu'ils sont immuables (ils ne peuvent pas être changés).
- En outre, ils sont créés à l'aide de parenthèses plutôt que de crochets.

```
langage = ("python", "c", "c++","java")
```

Les tuples

- Les tuples peuvent être créés sans les parenthèses, en séparant simplement les valeurs par des virgules.

Exemple:

```
my_tuple = "hello", "Fine", "thanks"
print(my_tuple[0])
print(my_tuple[1])
print(my_tuple[2])
```

La programmation modulaire

Fonctions & modules

Réutilisation du code

- La réutilisation du code est une partie très importante de la programmation dans toutes les langues. L'augmentation de la taille du code rend la maintenance plus difficile.
- Pour qu'un grand projet de programmation réussisse, il est essentiel de respecter le principe «Ne vous répétez pas» (Don't Repeat Yourself, or **DRY**). Nous avons déjà examiné une façon d'appliquer ce principe : en utilisant des boucles. Dans ce module, nous explorerons deux autres: les **fonctions** et les **modules**.

Fonctions & modules

Fonctions

- On a déjà utilisé des fonctions dans la partie précédente.
- Toute instruction consistant en un mot suivi d'informations entre parenthèses est un appel de **fonction**.
- Voici quelques exemples que vous avez déjà vus:

```
print("Hello world!")
range(2, 20)  str(12)
range(10, 20, 3)
```

- Les mots devant les parenthèses sont des noms de fonctions et les valeurs séparées par des virgules à l'intérieur des parenthèses sont des **arguments** de fonction.
- Les fonctions permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom.

```
def nom_de_la_function ( parametre1,parametre2,parametre3, parametreN ) :
    # Bloc d'instructions
```

Fonctions & modules

Arguments

- Toutes les définitions de fonctions que nous avons examinées jusqu'ici ont été des fonctions de zéro argument, appelées avec des parenthèses vides. Cependant, la plupart des fonctions prennent des arguments.

- L'exemple ci-dessous définit une fonction qui prend un argument:

```
>>> def print_python_function ( word) :
    print ( word + "!")

print_python_function ( "Hi")
print_python_function ( "Hello")
print_python_function ( "How Are you?")
```

Fonctions & modules

Arguments

- Les arguments de fonction peuvent être utilisés en tant que variables dans la définition de fonction. Cependant, ils ne peuvent pas être référencés en dehors de la définition de la fonction. Ceci s'applique également aux autres variables crées dans une fonction.

- Exécuter le code suivant sur votre machine

```
def function(variable):
variable += 1
print(variable)

function(7)
print(variable)
```

Fonctions & modules

Retour des fonctions

- Certaines fonctions, telles que int ou str, renvoient une valeur pouvant être utilisée ultérieurement. Pour cela, on peut utiliser l'instruction **return**.
- Une fois que vous retournez une valeur d'une fonction, celle-ci cesse immédiatement d'être exécutée. Tout code après la déclaration de retour ne se produira jamais.

.

Fonctions & modules

Les fonctions comme des objets

- Bien qu'elles soient crées différemment des variables normales, les fonctions sont comme n'importe quel autre type de valeur. Elles peuvent être affectées et réaffectées à des variables, puis référencées par ces noms.
- Les fonctions peuvent être également utilisées comme arguments d'autres fonctions.

```
def multiply(x, y):
return x * y

a = 4
b = 7
operation = multiply
print(operation(a, b))
```

```
def add(x, y):
return x + y

def do_twice(func, x, y):
return func(func(x, y), func(x, y))

a = 5
b = 10

print(do_twice(add, a, b))
```

Fonctions & modules

Modules

- Les modules sont des morceaux de code développés pour remplir des tâches courantes, telles que générer des nombres aléatoires, effectuer des opérations mathématiques, etc.
- La méthode de base pour utiliser un module consiste à ajouter **import nom_module** en haut de votre code, puis à utiliser nom_module.var pour accéder aux fonctions et aux valeurs portant le nom **var** dans le module.

Fonctions & modules

La méthode import

➢Il existe un grand nombre de modules disponibles sous Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module math qui contient, comme son nom l'indique, des fonctions mathématiques.

➢Lorsque on ouvre l'interpréteur Python, les fonctionnalités du module **math** ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
>>> import math
```

Fonctions & modules

La méthode import

- Après l'importation du module **math**, toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
import math

#Afficher la racine carré de 25
print(math.sqrt(25))
```

- Un autre l'exemple utilise le module **random** pour générer des nombres aléatoires:

```
import random

for i in range(10):
    value = random.randint(1, 11)
    print(value)
```

Fonctions & modules

La méthode import

- Un autre exemple utilise le module **random** pour générer des nombres aléatoires en utilisant la fonction **randint**:

```
import random

for i in range(10):
    value = random.randint(1, 11)
    print(value)
```

➔Le code utilise la fonction randint définie dans le module random pour imprimer 5 nombres aléatoires allant de 1 à 11.

Fonctions & modules

La méthode import

- Mais comment suis-je censé savoir quelles fonctions existent et ce que fait **math.sqrt** ou **random.randint**???

Fonctions & modules

Utiliser un espace de noms spécifique

- En réalité, quand on tape **import math**, cela crée **un espace de noms** nommé « math », contenant les **variables** et **fonctions** du module math. Quand vous tapez **math.sqrt**(), vous précisez à Python que vous souhaitez exécuter la fonction **sqrt** contenue dans l'espace de noms math.
- On peut importer un **module** ou un objet sous un nom différent à l'aide du mot-clé **as**. Ceci est principalement utilisé lorsqu'un module ou un objet a un nom long.

```
import math as mathématiques

#Afficher la racine carré de 25
print(mathématiques.sqrt(25))
```

Fonctions & modules

Méthode d'importation : from … import …

- Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module math. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable.
- Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

Exemples:

```
from math import fabs
fabs(-2)
print(fabs(-2))
```

```
from math import pi

print(pi)
```

Fonctions & modules

Les bibliothèques standards & pip

- Il existe trois principaux types de modules dans Python, ceux que vous écrivez vous-même, ceux que vous installez à partir de sources externes et ceux qui sont préinstallés avec Python.
- Le dernier type s'appelle **la bibliothèque standard** et contient de nombreux modules utiles. Certains modules utiles de la bibliothèque standard incluent **string**, **re**, **datetime**, **math**, **random**, **os**, **multiprocessing**, **subprocess**, **socket**, **email**, **json**, **doctest**, **unittest**, **pdb**, **argparse** et **sys**.

- Certains des modules de la bibliothèque standard sont écrits en Python et d'autres en C.

- La plupart sont disponibles sur toutes les plateformes, mais certaines sont spécifiques à Windows ou Unix.

Les exceptions

Les exceptions

Qu’est ce que les exceptions

- Vous avez déjà vu des exceptions dans les parties précédentes. Ils se produisent lorsque quelque chose ne va pas, à cause d'un code ou d'une saisie incorrecte.

- Lorsqu'une exception se produit, le programme s'arrête immédiatement.

- Donner un exemple d’exception qu’on a vu.

Les exceptions

Les types d'exceptions

Les différentes exceptions sont soulevées pour différentes raisons.

Exceptions communes:

- **ImportError**: une importation échoue;
- **IndexError**: une liste est indexée avec un numéro en dehors de la plage;
- **NameError**: une variable inconnue est utilisée;
- **SyntaxError**: le code ne peut pas être analysé correctement;
- **TypeError**: une fonction est appelée sur une valeur d'un type inapproprié;
- **ValueError**: une fonction est appelée sur une valeur du type correct, mais avec une valeur inappropriée.

Python a d'autres plusieurs exceptions intégrées, telles que **ZeroDivisionError** et **OSError**.

Les exceptions

Gestion des exceptions

- Pour gérer les exceptions et appeler un code lorsqu'une exception se produit, on utiliser une instruction **try / except.**
- Le bloc **try** contient du code susceptible de générer une exception. Si cette exception se produit, le code du bloc **try** cesse d'être exécuté et le code du bloc **except** est exécuté. Si aucune erreur ne se produit, le code du bloc **d'exception** ne s'exécute pas.

Par exemple:

```
try:
    a = 7
    b = 0
    print (a / b)
    print("Finish Division operation ")
except ZeroDivisionError:
    print("An error occurred")
    print("due to zero division")
```

Gestion des exceptions

- Une instruction **try** peut avoir plusieurs blocs différents pour gérer différentes exceptions.
- Plusieurs exceptions peuvent également être placées dans un seul bloc à l'aide de parenthèses, pour que le bloc gère toutes les exceptions.

```
try:
    var = 10
    print(var + "hello")
    print(var / 2)
except ZeroDivisionError:
    print("Divided by zero")
except (ValueError, TypeError):
    print("Error occurred due to ValueError or TypeError ")
```

Les exceptions

Gestion des exceptions

- Une instruction **except** sans aucune exception spécifiée va capturer toutes les erreurs. Celles-ci doivent être utilisées avec modération, car elles peuvent capturer des erreurs inattendues et masquer des erreurs de programmation.

Par exemple:

```
try:
    coding = "python"
    print(coding / 0)
except:
    print("An error occurred")
```

Les exceptions

L'instruction finally

- Pour assurer que le code est exécuté quelles que soient les erreurs, on peut utiliser une instruction **finally**.
- L'instruction **finally** est placée au bas d'une instruction **try / except**.
- Le code dans une instruction **finally** s'exécute toujours après l'exécution du code dans les blocs **try** et éventuellement dans les blocs **except**.

```
try:
    print("Hello friends")
    print(2 / 0)
except ZeroDivisionError:
    print("Divided by zero")
finally:
    print("This code will run no matter what")
```

Les exceptions

L’instruction finally

- Le code dans une instruction **finally** est même exécuté même si une exception se produit dans l'un des blocs précédents

```
try:
    print(1)
    print(10 / 0)
except ZeroDivisionError:
    print(unknown_var)
finally:
    print("This is executed last")
```

Les exceptions

Assertions

- Une assertion est un contrôle d'intégrité qu'on peut activer ou désactiver lorsque on a terminé de tester le programme.
- Une expression est testée, et si le résultat est faux, une exception est générée.
- Les assertions sont effectuées à l'aide de l'affirmation d'assert.
- L'assertion peut prendre un deuxième argument qui est transmis à **AssertionError** déclenchée si l'assertion échoue.

```
temp = -10
assert (temp >= 0), "Colder than absolute zero!"
```

Les matrices

Les matrices

➢ Création d'une matrice

- On travaille avec les modules **numpy** et **numpy.linalg**.

- Pour définir une matrice, on utilise la fonction **array** du module **numpy**.

```
import numpy as np
```

```
A=np.array([[1,2,3],[2,2,2],[3,2,0]])
print("la matrice A est",A)
```

```
la matrice A est [[1 2 3]
 [2 2 2]
 [3 2 0]]
```

Les matrices

- L'attribut **shape** donne la taille d'une matrice : nombre de lignes, nombre de colonnes. On peut redimensionner une matrice, sans modifier ses termes, à l'aide de l'attribut **reshape**.

```
print(' les dimensions de A sont',A.shape)
print('.....................................')
B=A.reshape((1,9))
print('la matrice B est égale à',B)
```

```
 les dimensions de A sont (3, 3)
.....................................
la matrice B est égale à [[1 2 3 2 2 2 3 2 0]]
```

Les matrices

L'accès à un terme de la matrice **A** se fait à l'aide de l'opération d'indexage **A[i, j]** où **i** désigne la ligne et **j** la ,colonne.

Attention, les indices commencent à zéro ! À l'aide d'intervalles, on peut également récupérer une partie d'une matrice : ligne, colonne, sous-matrice.

```
print("l'element A[1,0] est",A[1,0])# afficher terme de la deuxième ligne, première colonne
print('.....................................................................')
print("la premiere ligne de  A est ",A[0,:])# afficher première ligne sous forme de tableau à 1 dimension
print(A[0,:].shape)
print('.....................................................................')
print("la sous matrice A[0:1,:] est",A[0:1,:]) # afficher première ligne sous forme de matrice ligne array([[1, 3]])
print('.....................................................................')
print("la deuxieme colonne est A[:,1] est", A[:,1])# afficher deuxième colonne sous forme de tableau à 1 dimension
print('.....................................................................')
print("la deuxieme colonne est A[:,1:2]",A[:,1:2]) # affihcer deuxième colonne sous forme de matrice colonne
print('.....................................................................')
print(" la sous matrice A[1:3,0:2] est", A[1:3,0:2]) #afficher sous-matrice lignes 2 et 3, colonnes 1 et 2
```

```
l'element A[1,0] est 2
.....................................................................
la premiere ligne de  A est  [1 2 3]
(3,)
.....................................................................
la sous matrice A[0:1,:] est [[1 2 3]]
.....................................................................
la deuxieme colonne est A[:,1] est [2 2 2]
.....................................................................
la deuxieme colonne est A[:,1:2] [[2]
 [2]
 [2]]
.....................................................................
 la sous matrice A[1:3,0:2] est [[2 2]
 [3 2]]
```

Les matrices

- ✓ Les fonctions **zeros** et **ones** permettent de créer des matrices remplies de 0 ou de 1.
- ✓ La fonction **eye** permet de créer une matrice du type **In** où n est un entier.
- ✓ La fonction **diag** permet de créer une matrice diagonale.

```
   np.zeros((2,3))
array([[ 0.,0.,0.],
       [0.,0.,0.]])
   np.ones((3,2))
array([[ 1.,  1.],
       [1.,   1.],
       [1.,   1.]])
.....................................................
   np.eye(4)
array([[ 1.,0.,0.,0.],
       [0.,1.,0.,0.],
       [0.,0.,1.,0.],
       [0.,0.,0.,1.]])
.....................................................
   np.diag([1,2,3])
array([[1,  0,  0],
       [0,  2,  0],
       [0,  0,  3]])
```

Les matrices

- La fonction **concatenate** permet de créer des matrices par blocs en superposant (**axis=0**) ou en plaçant côte à côte (**axis=1**) plusieurs matrices.
-

```
A = np.ones((2,3))
B = np.zeros((2,3))
.....
np.concatenate((A,B), axis=0)# superposition des lignes de la matrice B aux lignes de la matrice A
array([[ 1.,1.,1.],
       [1.,1.,1.],
       [0.,0.,0.],
       [0.,0.,0.]])
.....
np.concatenate((A,B), axis=1)# ajout cote à cote des colonnes de la matrice B à coté des colonnes de la matrice A
array([[ 1.,1.,1.],
array([[ 1.,1.,1.,0.,0.,0.],
       [1.,1.,1.,0.,0.,0.]])
```

Les matrices

- Pour copier un tableau, il est recommandé d'utiliser la méthode **copy**.

```
A = np.array([[1,-2,3], [-4,5,-6]])
B = A.copy()
B[1, 0] = 8
.....................................................
A
array([[ 1,2,3],
       [-4,5,-6]])
.....................................................
B
array([[ 1,2,3],
       [8,5,-6]])
```

• La commande **array_equal** permet de tester l'égalité terme à terme de deux tableaux de même taille.

```
np.array_equal(A,B)
False
```

Les matrices

- Les fonctions **amax**, **amin** et **mean** du module **numpy** permettent respectivement de calculer le maximum, le minimum et la moyenne des éléments d'un tableau.

```
A = np.array([[1,-2,3], [-4,5,-6]])
print('la matrice A est',A)
print('.....................................................')
print('le plus grand élement de la matrice A est',np.amax(A))
print('.....................................................')
print('le plus petit élement de la matrice A est',np.amin(A))
print('.....................................................')
print('la moyenne des élements de la matrice A est',np.mean(A))
```

```
la matrice A est [[ 1 -2  3]
 [-4  5 -6]]
.....................................................
le plus grand élement de la matrice A est 5
.....................................................
le plus petit élement de la matrice A est -6
.....................................................
la moyenne des élements de la matrice A est -0.5
```

Calcul matriciel

Le calcul matriciel

- Les opérations d'ajout et de multiplication par un scalaire se font avec les opérateurs **+** et ***.

```
A = np.array([[1,2], [3,4]])
B = np.eye(2)
.....................................
A+3*B
array([[ 4.,2.],
       [3.,7.]])
```

Le calcul matriciel

- Pour effectuer un produit matriciel (lorsque que cela est possible), il faut employer la fonction **dot**.

```
A = np.array([[1,2], [3,4]])
B = np.array([[1,1,1],[2,2,2]])
.....................................
np.dot(A, B)
array([[5,5,5],
       [11,11,11]])
```

Le calcul matriciel

- On peut également utiliser la méthode **dot** qui est plus pratique pour calculer un produit de plusieurs matrices.
- La fonction **matrix_power** du module **numpy.linalg** permet de calculer des puissances de matrices.

```
A = np.array([[1,2], [3,4]])
print('la matrice A est',A)
print('.....................................................')
C=np.linalg.matrix_power(A,3)
print('la matrice A  au cube est',C)
```

```
la matrice A est [[1 2]
 [3 4]]
.....................................................
la matrice A  au cube est [[ 37  54]
 [ 81 118]]
```

Le calcul matriciel

- La transposée d'une matrice **B** s'obtient avec la fonction **np.transpose**. L'expression **B.T ou B.transpose** renvoie aussi la transposée de **B**.

```
B=np.array([[1, 2],[1, 2], [1, 2]])
print(B)
print('.....................................................................')
print('La matrice transposée de B est', np.transpose(B))
print('.....................................................................')
print('La matrice transposée de B est', B.transpose())
print('.....................................................................')
print('La matrice transposée de B est', B.T)
```

```
[[1 2]
 [1 2]
 [1 2]]
.....................................................................
La matrice transposée de B est [[1 1 1]
 [2 2 2]]
.....................................................................
La matrice transposée de B est [[1 1 1]
 [2 2 2]]
.....................................................................
La matrice transposée de B est [[1 1 1]
 [2 2 2]]
```

Le calcul matriciel

- Le déterminant, le rang et la trace d'une matrice s'obtiennent par les fonctions **det**, **matrix_rank** du module **numpy.linalg** et **trace** du module **numpy**. Enfin la fonction **inv** du module **numpy.linalg** renvoie l'inverse de la matrice si elle existe.

```
A = np.array([[1,2], [3,4]])
print(A)
print('Le déterminant de A est',np.linalg.det(A))
print('La trace  de A est',np.trace(A))
print('La matrice inverse  de A est',np.linalg.inv(A))
```

```
[[1 2]
 [3 4]]
Le déterminant de A est -2.0000000000000004
La trace  de A est 5
La matrice inverse  de A est [[-2.   1. ]
 [ 1.5 -0.5]]
```

Le calcul matriciel

- Pour résoudre le système linéaire AX=b lorsque la matrice est inversible, on peut employer la fonction **np.linalg.solve.**

```
A=np.array([[1,2,3],[2,2,2],[3,2,0]])
b=np.array([1,2,0])
print('.....................................................................')
print(A)
print('.....................................................................')
print(b)
print('.....................................................................')
print("la solution du système linéaire AX=b est", np.linalg.solve(A,b))
```

```
.....................................................................
[[1 2 3]
 [2 2 2]
 [3 2 0]]
.....................................................................
[1 2 0]
.....................................................................
la solution du système linéaire AX=b est [ 4. -6.  3.]
```

Eléments propores d'une matrice

- La fonction **poly** du module **numpy** appliquée à une matrice carrée renvoie la liste des coefficients du polynôme caractéristique par degré décroissant.
- La fonction **eigvals** du module **numpy.linalg** renvoie les valeurs propres d'une matrice.
- Pour obtenir en plus les vecteurs propres associés, il faut employer la fonction **eig**. Cette fonction renvoie un **tuple** constitué de la liste des valeurs propres et d'une matrice carrée. La $i^{ème}$ colonne de cette matrice est un **vecteur propre** associé à la **$i^{ième}$ valeur propre** de la liste des valeurs propres.

```
A=np.array([[1,2,3],[2,2,2],[3,2,0]])
print('.....................................................................')
print('les coéffiients du polynome caractéristique de A sont',np.poly(A))
print('.....................................................................')
print('Les valeurs propres de A sont',np.linalg.eigvals(A))
print('.....................................................................')
print("La matrice des vecteurs propres de A son ", np.linalg.eig(A))
```

```
.....................................................................
les coéffiients du polynome caractéristique de A sont [  1.  -3. -15.  -2.]
.....................................................................
Les valeurs propres de A sont [ 5.6953739  -2.55809925 -0.13727466]
.....................................................................
La matrice des vecteurs propres de A son  (array([ 5.6953739 , -2.55809925, -0.13727466]), array([[-0.59506827, -0.61917918,
0.51235817],
       [-0.60708417, -0.07143949, -0.79141974],
       [-0.52663323,  0.78199331,  0.33338251]]))
```

Produit scalaire et produit vectoriel

- La fonction **vdot** permet de calculer le produit scalaire de deux vecteurs de l'espace.
- La fonction **cross** permet de calculer le produit vectoriel de deux vecteurs de l'espace. .

```
u=np.array([5,-1,2])
v=np.array([0,-1,1])
print('.....................................................')
print('le produit scalaire des vecteurs u et v est égal à', np.vdot(u,v))
print('.....................................................')
print('le produit vectoriel des vecteurs u et v est égal à', np.cross(u,v))
```

```
.....................................................
le produit scalaire des vecteurs u et v est égal à 3
.....................................................
le produit vectoriel des vecteurs u et v est égal à [ 1 -5 -5]
```

Réalisation de tracés

Réalisations de tracés

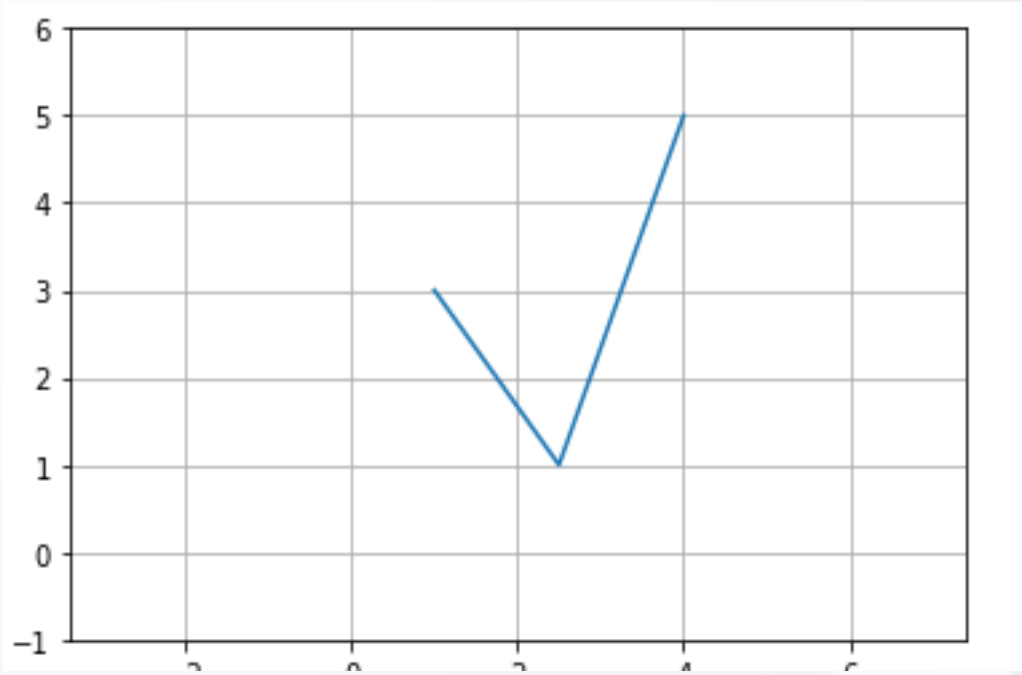
- Les fonctions présentées dans ce document permettent la réalisation de tracés. Elles nécessitent l'import du module **numpy** et du module **matplotlib.pyplot**. De plus pour effectuer des tracés en dimension 3, il convient d'importer la fonction **Axes3d** du module **mpl_toolkits.mplot3d**. Les instructions nécessaires aux exemples qui suivent sont listés ci-dessous.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

Tracés de lignes brisées et options de tracés

- On donne la liste des abscisses et la liste des ordonnées puis on effectue le tracé. La fonction **axis** permet de définir la fenêtre dans laquelle est contenue le graphique. L'option **equal** permet d'obtenir les mêmes échelles sur les deux axes. Les tracés relatifs à divers emplois de la fonction **plot** se superposent. La fonction **plt.clf()** efface les tracés contenus dans la fenêtre graphique.

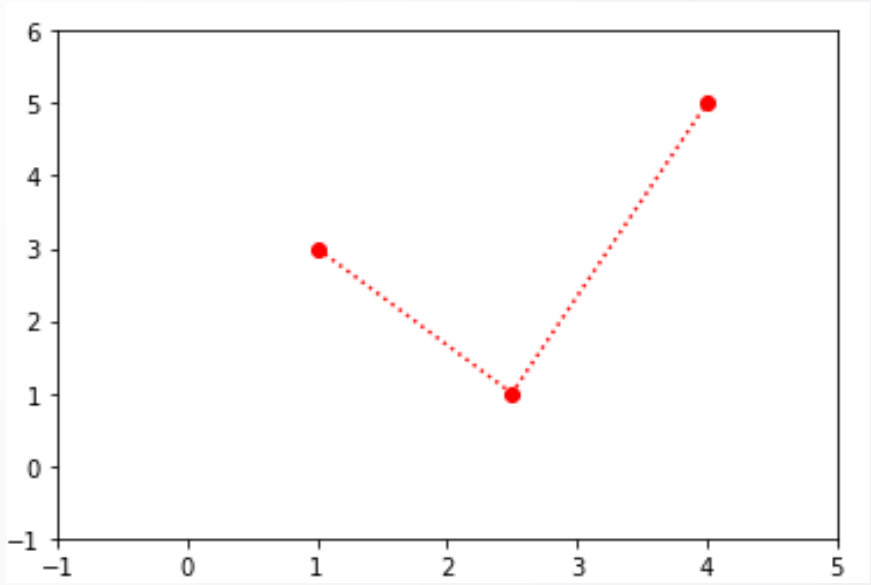
```
x = [1., 2.5, 4.]
y = [3., 1., 5.]
plt.axis('equal')
plt.plot(x, y)
plt.axis([-1., 5., -1., 6.])
plt.grid()
plt.show()
```



Tracés de lignes brisées et options de tracés

- La fonction **plot** admet de nombreuses options de présentation. Le paramètre **color** permet de choisir la couleur ('g' : vert, 'r' : rouge, 'b' : bleu, …). Pour définir le style de la ligne, on utilise **linestyle** ('-' : ligne continue, '- -' : ligne discontinue, ' :' : ligne pointillée, …). Si on veut marquer les points des listes, on utilise le paramètre **marker** ('+', '.', 'o', 'v' donnent différents symboles).

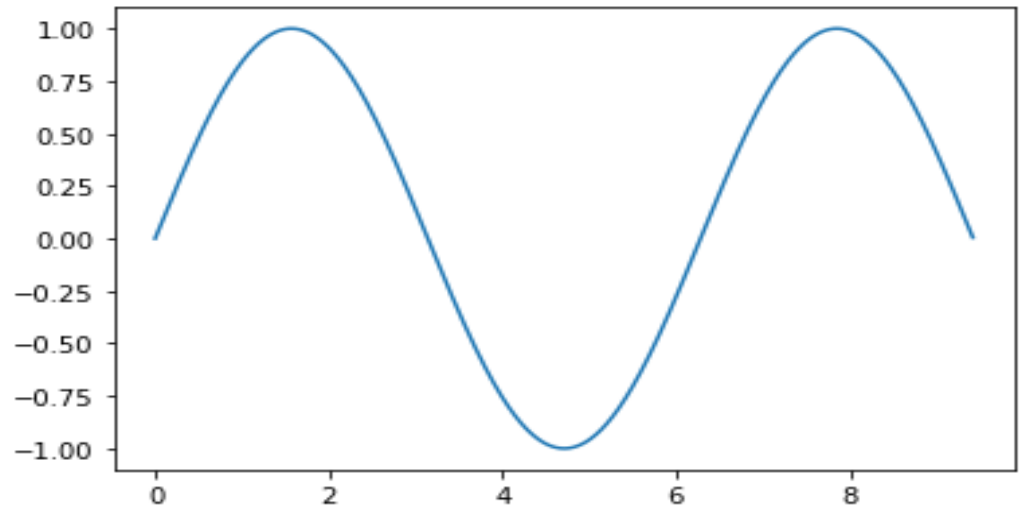
```
x = [1., 2.5, 4.]
y = [3., 1., 5.]
plt.axis([-1., 5., -1., 6.])
plt.plot(x, y, color='r', linestyle=':', marker='o')
plt.show()
```



Exemple: tracés des fonctions

- On définit **u↦ sin(u)** où u appartient à [0, 3*pi] puis on construit la liste des ordonnées correspondantes.

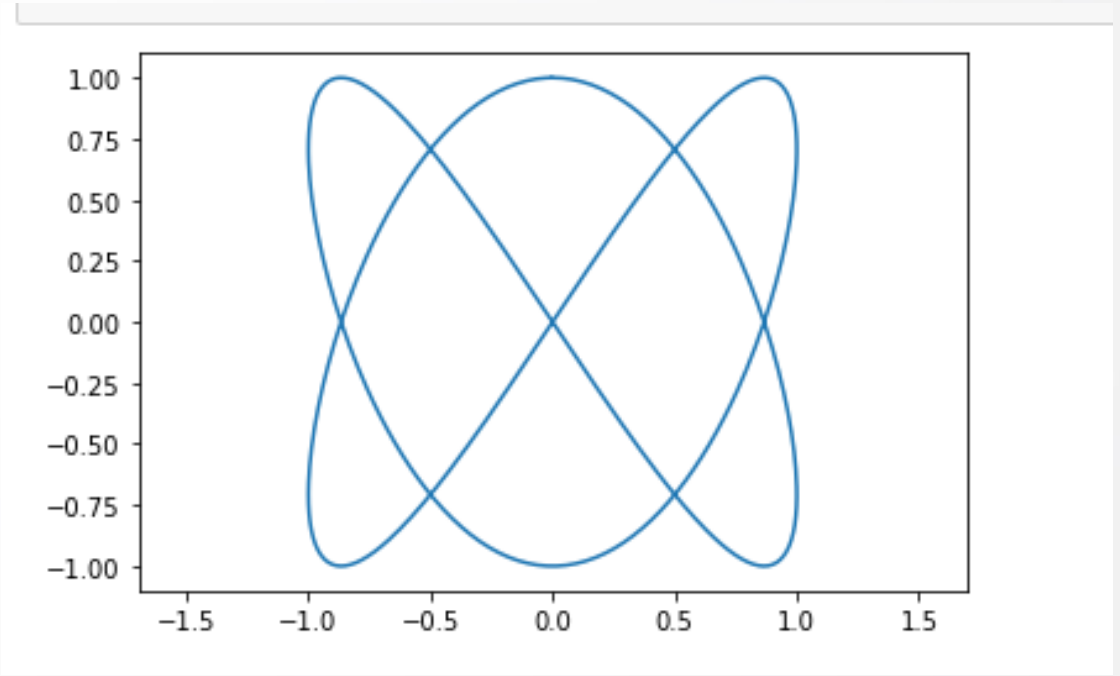
```
f=lambda x : np.sin(x)
X = np.arange(0, 3*np.pi, 0.01)
Y = [ f(x) for x in X ]
plt.plot(X, Y)
plt.show()
```



Tracés d'arcs paramétrés

- Dans le cas d'un arc paramétré du plan, on définit d'abord la liste des valeurs données au paramètre puis on construit la liste des abscisses et des ordonnées correspondantes. On effectue ensuite le tracé.

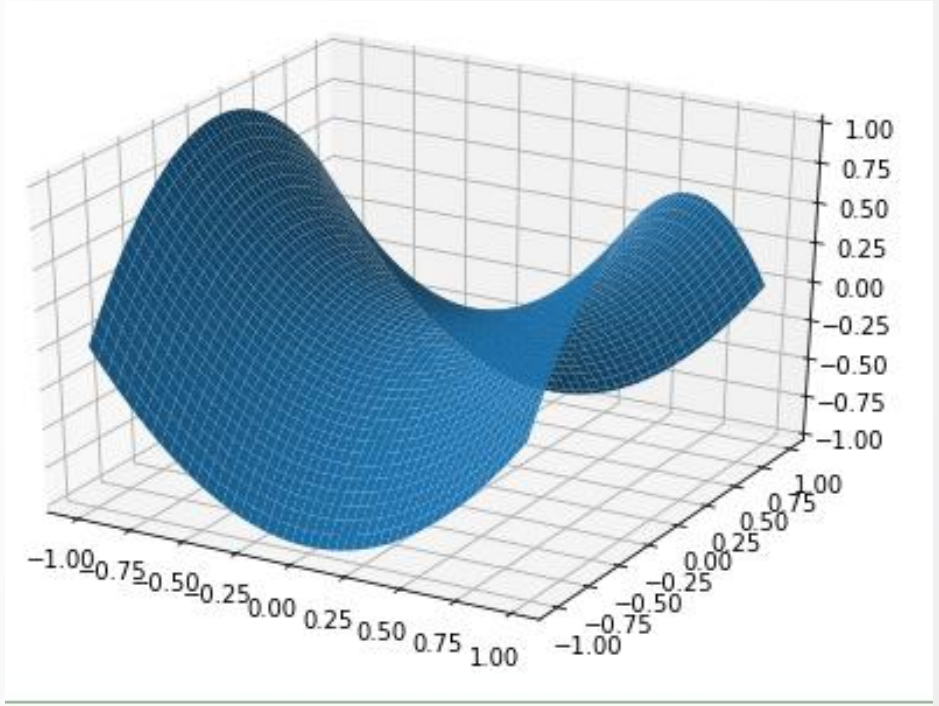
```
x=lambda t:np.sin(2*t)
y= lambda t :np.cos(3*t)
T =np.arange(0, 2*np.pi, 0.01)
X = x(T)
Y = y(T)
plt.axis('equal')
plt.plot(X, Y)
plt.show()
```



Exemple 3d

- Pour tracer une surface d'équation $z = f(x, y)$, on réalise d'abord une grille en (x, y) puis on calcule les valeurs de z correspondant aux points de cette grille. On fait ensuite le tracé avec la fonction plot_surface.

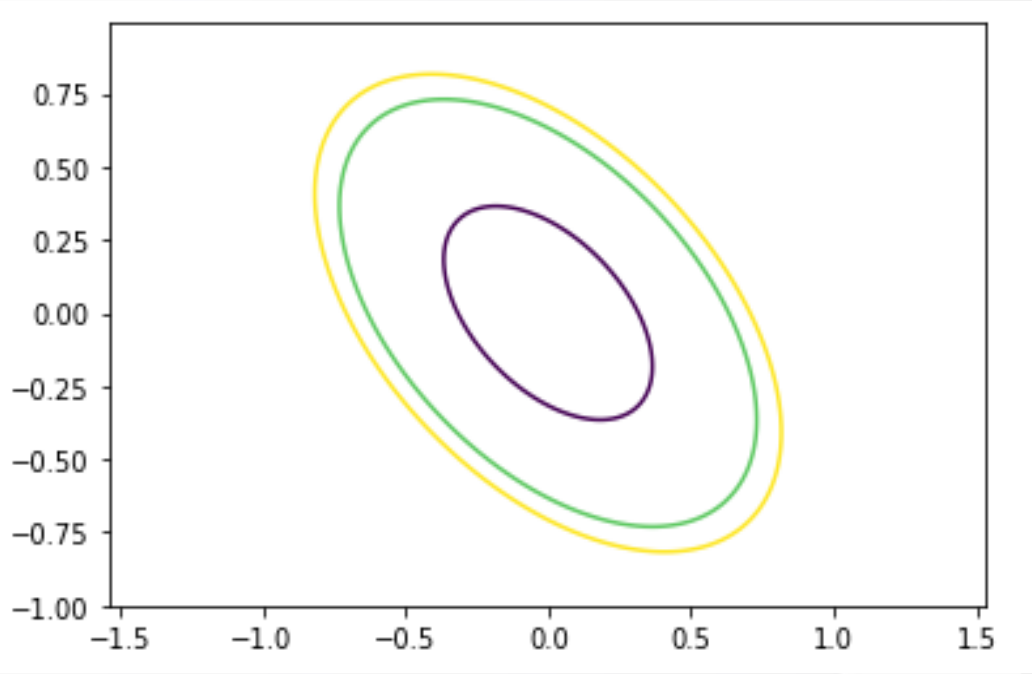
```
ax = Axes3D(plt.figure())
f= lambda x, y:x**2 - y**2
X = np.arange(-1, 1, 0.02)
Y = np.arange(-1, 1, 0.02)
X,Y=np.meshgrid(X, Y)
Z = f(X,Y)
ax.plot_surface(X, Y, Z)
plt.show()
```



Tracé de lignes de niveau

Pour tracer des courbes d'équation $f(x, y) = k$, on fait une grille en x et et y sur laquelle on calcule les valeurs de f. On emploie ensuite la fonction contour en mettant dans une liste les valeurs de k pour lesquelles on veut tracer la courbe d'équation $f(x, y) = k$.

```
def f(x,y) :
    return x**2 + y**2 + x*y
X = np.arange(-1, 1, 0.01)
Y = np.arange(-1, 1, 0.01)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
plt.axis('equal')
plt.contour(X, Y, Z, [0.1,0.4,0.5])
plt.show()
```



Analyse numérique

Analyse numérique

- La plupart des fonctions présentées dans cette section nécessitent l'import du module numpy et de sous-modules du module scipy. Les instructions nécessaires aux exemples suivants sont listés ci-dessous.

```
import numpy as np
import scipy.optimize as resol
import scipy.integrate as integr
import matplotlib.pyplot as plt
```

Nombres complexes

- Python calcule avec les nombres complexes. Le nombre imaginaire pur i se note 1j. Les attributs real et imag permettent d'obtenir la partie réelle et la partie imaginaire. La fonction abs calcule le module d'un complexe.

```
 a = 2 + 3j
 b = 5 - 3j
print('.....................................................................')
 print('le produit a*b est donné par',a*b)
print('.....................................................................')
print('la partie réelle de a est ',a.real)
print('.....................................................................')
print('la partie réelle de a est ',a.imag)
print('.....................................................................')
print('le module de a est',abs(a))
```

```
.....................................................................
le produit a*b est donné par (19+9j)
.....................................................................
la partie réelle de a est  2.0
.....................................................................
la partie réelle de a est  3.0
.....................................................................
le module de a est 3.605551275463989
```

Résolution approchée d'équations

- Pour résoudre une équation du type $f(x) = 0$ où f est une fonction d'une variable réelle, on peut utiliser la fonction fsolve du module scipy.optimize. Il faut préciser la valeur initiale $x0$ de l'algorithme employé par la fonction fsolve. Le résultat peut dépendre de cette condition initiale.

```
def f(x) :
    return x**2 - 2
print("Pour la condition initiale x=-2,la solution de l'équation f(x)=0 est", resol.fsolve(f, -2.))
print('.....................................................................')
print("Pour la condition initiale x=2,la solution de l'équation f(x)=0 est", resol.fsolve(f, 2.))
```

```
Pour la condition initiale x=-2,la solution de l'équation f(x)=0 est [-1.41421356]
.....................................................................
Pour la condition initiale x=2,la solution de l'équation f(x)=0 est [1.41421356]
```

Résolution d'un système

- Dans le cas d'une fonction f à valeurs vectorielles, on utilise la fonction root. Par exemple, pour résoudre le système non linéaire. $\begin{cases} x^2 - y^2 = 1 \\ x + 2y - 3 = 0 \end{cases}$

```
def f(v) :
    return v[0]**2-v[1]**2-1, v[0]+2*v[1]-3
print('.....................................................................')
print("Pour la condition initiale (0,0),la solution du système S est",resol.root(f, [0,0]))
print('.....................................................................')
print("Pour la condition initiale (-5,5),la solution du système S  est", resol.root(f, [-5,5]))
```

```
     fun: array([2.22044605e-16, 0.00000000e+00])
 message: 'The solution converged.'
    nfev: 11
     qtf: array([ 3.64074679e-11, -1.43012733e-11])
       r: array([ 2.73511307, -0.97830981,  2.53305931])
  status: 1
 success: True
       x: array([1.30940108, 0.84529946])
.....................................................................
Pour la condition initiale (-5,5),la solution du système S  est     fjac: array([[-0.99354255,  0.11346009],
       [-0.11346009, -0.99354255]])
     fun: array([ 8.8817842e-15, -4.4408921e-16])
 message: 'The solution converged.'
    nfev: 9
     qtf: array([5.14182101e-09, 5.87182754e-10])
       r: array([ 8.81367149, 10.74388946, -0.78607331])
  status: 1
 success: True
       x: array([-3.30940108,  3.15470054])
```

Calcul approché d'intégrales

- La fonction quad du module scipy.integrate permet de calculer des valeurs approchées d'intégrales. Elle renvoie une valeur approchée de l'intégrale ainsi qu'un majorant de l'erreur commise. Cette fonction peut aussi s'employer avec des bornes d'intégration égales à $+\infty$ ou $-\infty$.

```
f=lambda x: np.exp(-x)
print('.....................................................................')
print("l'integrale de f entre 0 et 1 est égale à et son erreur sont le couple",integr.quad(f,0,1))
```

```
.....................................................................
l'integrale de f entre 0 et 1 est égale à et son erreur sont le couple (0.6321205588285578, 7.017947987503856e-15)
```

Intégrale dépendante d'un paramètre

- La fonction integr.quad peut être employée pour la définition d'intégrales à paramètres. Ainsi si on veut obtenir des valeurs $\Gamma(x) = \int_0^{+\infty} e^{-t} t^{x-1}$ approchées de pour x réel strictement positif on pourra procéder ainsi :

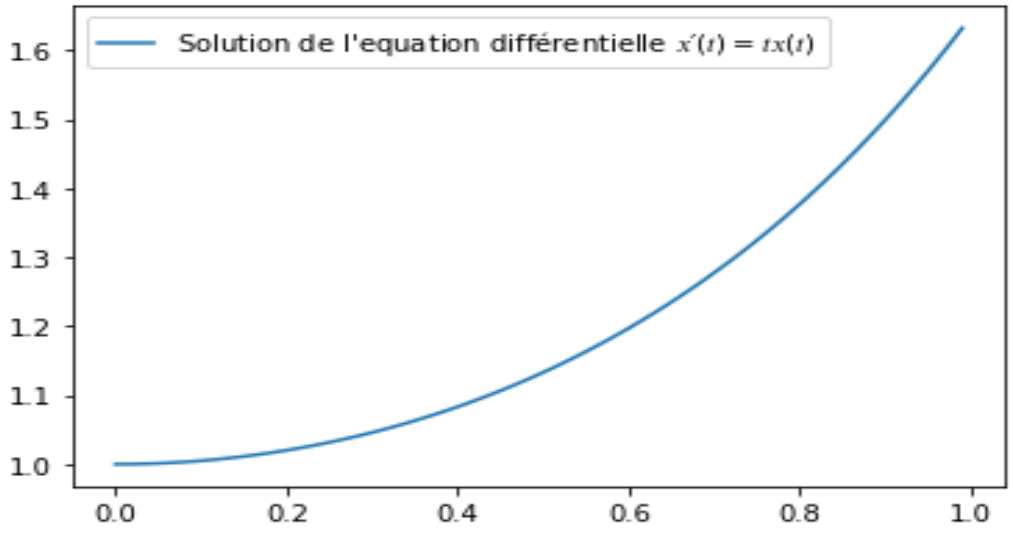
```
def g(x) :
    def f(t) :
        return np.exp(-t)*t**(x-1)
    return integr.quad(f,0,np.inf)[0]
print("la valeur de l'intégrale pour x=2 est",g(2))
```

```
.....................................................
la valeur de l'intégrale pour x=2 est 0.9999999999999998
```

Résolution approchées d'équations différentielles

- Pour résoudre une équation différentielle $x' = f(x, t)$, on peut utiliser la fonction odeint du module scipy.integrate. Cette fonction nécessite une liste de valeurs de t, commençant en $t0$, et une condition initiale $x0$. La fonction renvoie des valeurs approchées (aux points contenus dans la liste des valeurs de t de la solution x de l'équation différentielle qui vérifie $x(t0) = x0$. Pour trouver des valeurs approchées sur $[0, 1]$ de la solution $x'(t) = tx(t)$ qui vérifie $x(0) = 1$, on peut employer le code suivant.

```
f=lambda x,t:t*x
T=np.arange(0,1,0.01)
X=integr.odeint(f,1,T)
plt.plot(T,X)
plt.legend(("Solution de l'equation différentielle $x'(t) = tx(t)$",), loc = 0)
plt.show()
```

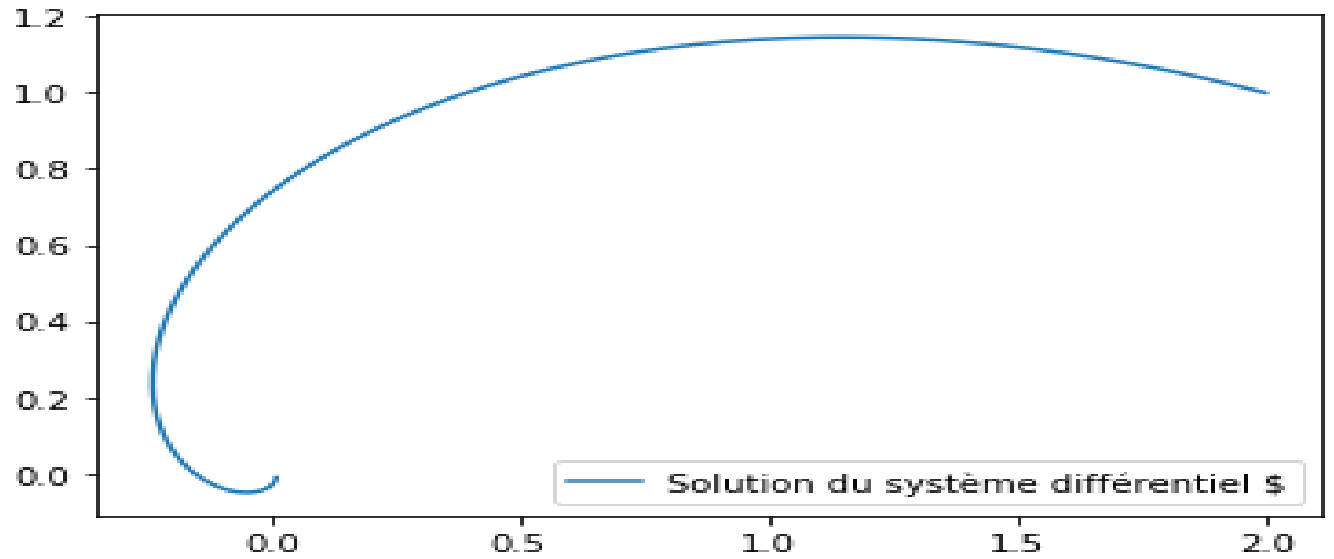


Système différentiel

- Si on veut résoudre, sur [0, 1], le système différentiel (S) $\begin{cases} x'(t) = -x(t) - y(t) \\ y'(t) = x(t) - y(t) \end{cases}$

avec la condition initiale $x(0) = 2$, $y(0) = 1$ le code devient le suivant

```
def f(x, t) :
    return np.array([-x[0]-x[1], x[0]-x[1]])
T = np.arange(0, 5.01, 0.01)
X = integr.odeint(f, np.array([2.,1.]), T)
plt.plot(X[ :,0], X[ :,1])
plt.legend(("Solution du système différentiel $",), loc = 0)
plt.show()
```



Polynômes

Polynômes

La classe Polynomial du module numpy.polynomial.polynomial permet de travailler avec des polynômes.

- Pur créer un polynôme, il faut lister ses coefficients par ordre de degré croissant. Par exemple, pour le polynôme $X^3 + 2X - 3$. l'instruction à exécuter est p = Polynomial([-3, 2, 0, 1])
- On peut alors utiliser la variable p comme une fonction pour calculer, en un point quelconque, la valeur de la fonction polynôme associée. Cette fonction peut agir également sur un tableau de valeurs, elle calcule alors la valeur de la fonction polynôme en chacun des points indiqués.

```
from numpy.polynomial import Polynomial
p = Polynomial([-3, 2, 0, 1])
print("l'image de 3 par le polynome p est",p(3))
```

```
l'image de 3 par le polynome p est 30.0
```

Polynômes

- L'attribut coef donne accès aux coefficients ordonnés par degré croissant ; ainsi p.coef[i] correspond au coefficient du terme de degré i. La méthode degré renvoie le degré du polynôme alors que roots calcule ses racines.

```
print("les coefficients de p sont",p.coef)
print('.....................................................................')
print("le cofficients du monome d'ordre 1 de p est",p.coef[1])
print('.....................................................................')
print("le degré de p est",p.degree())
print('.....................................................................')
print("les racines de  p sont",p.roots())
```

```
les coefficients de p sont [-3.  2.  0.  1.]
.....................................................................
le cofficients du monome d'ordre 1 de p est 2.0
.....................................................................
le degré de p est 3
.....................................................................
les racines de  p sont [-0.5-1.6583124j -0.5+1.6583124j  1. +0.j       ]
```

Polynômes

- La méthode deriv renvoie un nouveau polynôme, dérivé du polynôme initial. Cette méthode prend en argument facultatif un entier positif indiquant le nombre de dérivations à effectuer. De la même manière la méthode integ intègre le polynôme, elle prend un paramètre optionnel supplémentaire donnant la constante d'intégration à utiliser, ce paramètres peut être une liste en cas d'intégration multiple ; les constantes d'intégration non précisées sont prises égales à zéro.

```
print("les coefficients du polynome dérivé de p est",p.deriv().coef)
print('.....................................................................')
print("les coefficients du polynome dérivé deuxième de p est",p.deriv(2).coef)
print('.....................................................................')
print("les coefficients du polynome primitive  de p est",p.integ().coef)
```

```
les coefficients du polynome dérivé de p est [2. 0. 3.]
.....................................................................
les coefficients du polynome dérivé deuxième de p est [0. 6.]
.....................................................................
les coefficients du polynome primitive  de p est [ 0.   -3.    1.    0.    0.25]
```

Polynômes

- Les opérateurs +, -, * permettent d'additionner, soustraire et multiplier des polynômes. Ils fonctionnent également entre un polynôme et un scalaire. L'opérateur ** permet d'élever un polynôme à une puissance entière positive. Enfin, on peut composer deux polynômes (p(q) remplace l'indéterminée X par le polynôme q dans le polynôme p)

```
a=Polynomial([1, 2, 1])
b=Polynomial([5, 3])
c=Polynomial([-7, 2])
d=2*a * b + Polynomial([-7,2])
e=(p**2).coef
f=a(b).coef
print("le polynôme $(2a+b+c)(x)$ ",d)
print("les coffecients de $2a+b+c$", p.coef)
print("les coefficients de polynome carré de p est",e)
print("les coefficients de polynome carré de $a(b)$ est",f)
```

```
le polynôme $(2a+b+c)(x)$  poly([ 3. 28. 22.  6.])
les coffecients de $2a+b+c$ [ 3. 28. 22.  6.]
les coefficients de polynome carré de p est [   9.  168.  916. 1268.  820.  264.   36.]
les coefficients de polynome carré de $a(b)$ est [36. 36.  9.]
```

Polynômes

L'opérateur / permet de diviser un polynôme par un scalaire. Pour diviser deux polynômes il faut utiliser l'opérateur // qui renvoie le quotient ; l'opérateur % calcule le reste.

```
(p / 2).coef
>>>array([ 1.5, 14. , 11. , 3. ])
 q = p // a
 r = p % a
 q.coef
>>>array([ 10., 6.])
 r.coef
>>>array([-7., 2.])
 (q * a + r).coef
array([ 3., 28., 22., 6.])
```

Probabilité

Probabilité

- Les fonctions d'échantillonnage et de génération de valeurs pseudo-aléatoires sont regroupées dans la bibliothèque numpy.random.
- L'expression randint(a, b) permet de choisir un entier au hasard dans l'intervalle $[\![a, b[\![$. La fonction randint prend un troisième paramètre optionnel permettant d'effectuer plusieurs tirages et de renvoyer les résultat sous forme de tableau ou de matrice.

```
import numpy.random as rd
print(" une réalisation possible d'une expérience aléatoire de lancer d'un dé est", rd.randint(1, 7))
print("une réalisation possible d'une expérience de lancer  de 20 dés à la fois ",rd.randint(1, 7, 20))
print("une réalisation possible d'une expérience de lancer de 20 dés à la fois sous forme matricielle est "
      ,rd.randint(1, 7,(4,5)))
```

```
 une réalisation possible d'une expérience aléatoire de lancer d'un dé est 3
une réalisation possible d'une expérience de lancer  de 20 dés à la fois  [4 6 3 2 1 1 5 2 3 3 3 4 5 1 4 4 5 1 6 1]
une réalisation possible d'une expérience de lancer de 20 dés à la fois sous forme matricielle est  [[2 2 2 1 6]
 [4 5 3 1 6]
 [5 4 1 3 1]
 [4 2 6 5 3]]
```

Probabilité

- La fonction random renvoie un réel compris dans l'intervalle [0, 1[. Si X désigne la variable aléatoire correspondant au résultat de la fonction random, alors pour tout a et b dans [0, 1] avec $a \leqslant b$, on a $P\ (a \leqslant X < b) = b - a$.
- Cette fonction accepte un paramètres optionnel permettant de réaliser plusieurs tirages et de les renvoyer sous forme de tableau ou de matrice.

```
print(" un nombre aléatoire entre 0 et 1 est", rd.random())
print(" un un couple de nombres aléatoires entre 0 et 1 est", rd.random(2))
print(" une matrice de  nombres aléatoires entre 0 et 1 est", rd.random((3,2)))
```

```
un nombre aléatoire entre 0 et 1 est 0.9788675274377165
un un couple de nombres aléatoires entre 0 et 1 est [0.69844132 0.25223252]
une matrice de  nombres aléatoires entre 0 et 1 est [[0.82730044 0.91547874]
[0.49402908 0.51465053]
[0.24730793 0.0664916 ]]
```

Probabilité

- La fonction binomial permet de simuler une variable aléatoire suivant une loi binomiale de paramètres n et p.
- Elle permet donc également de simuler une variable aléatoire suivant une loi de Bernoulli de paramètres p en prenant simplement $n = 1$. Cette fonction prend un troisième paramètre optionnel qui correspond, comme pour les fonctions précédentes, au nombre de valeurs à obtenir.

```
print(" une réalisation possible d'une expérience aléatoire binomiale de paramètres $n=10$  et $p=0.3$ est",
      rd.binomial(10, 0.3))
print("un ensemble de 5 réalisations possibles d'une expérience aléatoire binomiale de paramètres $n=10$  et $p=0.3$ ",
      rd.binomial(10, 0.3,5))
print("un ensemble de 20 réalisations possibles d'une expérience aléatoire binomiale de paramètres $n=10$
      et $p=0.3$  sous forme matricielle est " ,rd.binomial(10, 0.3,(4,5)))
```

```
 une réalisation possible d'une expérience aléatoire binomiale de paramètres $n=10$  et $p=0.3$ est 4
un ensemble de 5 réalisations possibles d'une expérience aléatoire binomiale de paramètres $n=10$  et $p=0.3$  [2 0 2 2 2]
un ensemble de 20 réalisations possibles d'une expérience aléatoire binomiale de paramètres $n=10$  et $p=0.3$  sous forme matr
icielle est  [[3 4 4 3 3]
 [4 2 6 7 2]
 [2 2 4 4 5]
 [1 1 1 4 3]]
```

Probabilité

- Les fonctions geometric et poisson fonctionnement de la même manière pour les lois géométrique ou de Poisson.

```
rd.geometric(0.5, 8)
```

```
array([1, 1, 1, 2, 1, 3, 1, 1])
```

```
rd.poisson(0.5, 8)
```

```
array([0, 1, 2, 0, 3, 1, 1, 0])
```