Programmation 3
CM séance 2 : Modules

Manon Scholivet, adapté des cours de Florian Bridoux et Jeremy Auguste

16 septembre 2020

Plan

Surcharge : str

Chaque objet a une valeur str par défaut :

1. Fichier "angle.py" :

```
class Angle :
    def __init__(self, degre) :
        self.degre = degre

if __name__ == '__main__' :
    a = Angle(180)
    print(a) # équivalent à : print(str(a))
```

2. Exécution (dans un terminal) :

$ python3 angle.py

<__main__.Angle object at 0x7f44d38a2a58>

Surcharge : str

Chaque objet a une valeur str par défaut :

1. Fichier "angle.py" :

```
class Angle :
    def __init__(self, degre) :
        self.degre = degre
    def __str__(self) :
        return str(self.degre)+'°'
if __name__ == '__main__' :
    a = Angle(180)
    print(a) # équivalent à : print(str(a))
```

2. Exécution (dans un terminal) :

   $ python3 angle.py

   180°

Surcharge : str

1. Fichier "angle.py" :

```
class Angle :
    def __init__(self, degre) :
        self.degre = degre
    def __str__(self) :
        return str(self.degre)+'°'
if __name__ == '__main__' :
    a = Angle(180)
    print(a) # équivalent à : print(str(a))
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> a = Angle(120)
>>> a
<angle.Angle object at 0x7f32c3228080>
```

Surcharge : repr

1. Fichier "angle.py" :

```
class Angle :
    def __init__(self, degre) :
        self.degre = degre
    def __str__(self) :
        return str(self.degre)+'°'
    def __repr__(self) :
        return "Instance d'Angle valant "+str(self)
if __name__ == '__main__' :
    a = Angle(180)
    print(a) # équivalent à : print(str(a))
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> a = Angle(120)
>>> a
Instance d'Angle valant 120°
```

Surcharge : add

1. Fichier "angle.py" :

```
class Angle :
...
    def __add__(self, other) :
        res = (self.degre + other.degre) % 360
        return Angle(res)
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> a = Angle(120)
>>> b = Angle(30)
>>> print(a+b)
80°
```

Surcharge : add

1. Fichier "angle.py" :

```
class Angle :
...
    def __add__(self, other) :
        res = (self.degre + other.degre) % 360
        return Angle(res)
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> a = Angle(120)
>>> print(a+30)
AttributeError: 'int' object has no attribute 'degre'
```

Surcharge : add

1. Fichier "angle.py" :

```
class Angle :
...
    def __add__(self, other) :
        try:
            res = (self.degre + other.degre) % 360
            return Angle(res)
        except:
            raise TypeError("Addition Angle et " + str(type(other))
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> a = Angle(120)
>>> print(a+30)
TypeError: Addition Angle et <class 'int'>
```

Surcharge : mul

1. Fichier "angle.py" :

```
class Angle :
...
    def __mul__(self, other) :
        try:
            res = (self.degre * other.degre) % 360
            return Angle(res)
        except:
            raise TypeError("Multiplication Angle et " + str(type(o
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> Angle(5)*3
Instance d'Angle valant 15°
>>> 3*Angle(5)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'int' and 'Angle'
```

Surcharge : rmul

1. Fichier "angle.py" :

```
class Angle :
...
    def __mul__(self, other) :
        try:
            res = (self.degre * other.degre) % 360
            return Angle(res)
        except:
            raise TypeError("Multiplication Angle et " + str(type(o
    def __rmul__(self, other) :
        return self * other
```

2. Exécution (Mode interactif python) :

```
>>> from angle import *
>>> Angle(5)*3
Instance d'Angle valant 15°
>>> 3*Angle(5)
Instance d'Angle valant 15°
```

Opérateurs de calculs

+	__add__(self,other)
+=	__iadd__(self,other)
-	__sub__(self,other)
-=	__isub__(self,other)
*	__mul__(self,other)
*=	__imul__(self,other)
**	__pow__(self,other)
**=	__ipow__(self,other)
/	__truediv__(self,other)
/=	__itruediv__(self,other)
//	__floordiv__(self,other)
//=	__ifloordiv__(self,other)

Opérateurs unaires et logiques

-	__neg__(self,other)
+	__pos__(self,other)
abs	__abs__(self,other)
not	__not__(self,other)
and	__and__(self,other)
or	__or__(self,other)

Opérateurs unaires de signes et comparaison

len(instance)	__len__(self)
other in instance	__contains__(self,other)
instance[clé] ou instance.get(clé)	__getitem__(self,clé)
instance[clé] = valeur ou instance.set(clé, valeur)	__setitem__(self,clé, valeur)
del instance[clé]	__delitem__(self,clé,valeur)
==	__eq__(self,other)
!=	__ne__(self,other)
<	__let__(self,other)
>	__get__(self,other)
<=	__le__(self,other)
>=	__ge__(self,other)

Surcharge : conclusion

On peut tout surcharger... sauf l'affectation.

Voir la liste complète ici :

`https://docs.python.org/3/reference/datamodel.html#specialmethod-names`

Plan

Où écrivez-vous votre code ?

Pour le moment, probablement dans un fichier unique.

Dans ce fichier il y a :

- Des définitions d'objets
- Des fonctions
- Des variables globales
- Le programme principal (__*main*__)
- Une analyse d'éventuels arguments de ligne de commande
- ...

Quel est le problème avec ça ?

Si votre script est court et que chaque partie ne sera utilisée que dans ce script : pas de soucis.

Cependant, si :

- Votre programme est désormais un projet avec de nombreuses lignes ;
- Vous avez des fonctions que vous pourriez réutiliser dans d'autre programmes (p.ex. un analyseur d'un certain type de fichier,...)
- Vous êtes plusieurs à travailler sur le même projet

Alors, travailler sur un unique fichier rendra tous les points précédents problématiques.

Utilisation de modules pour structurer votre programme

La solution à cela est d'utiliser des modules.

Qu'est-ce qu'un module ?

En Python, un module est un fichier ".py" contenant des instructions et des définitions de fonction. Un module peut être importé par d'autres modules.

Chaque module se concentrera uniquement sur un aspect du programme :

- Chaque objet sera définit dans un module (un fichier par objet)
- Les modules "utilitaires" : analyse des fichiers d'entrée, fonctions utilitaires, ...
- Le module principal avec l'éventuel analyse des arguments et quelques appels de fonction à partir d'autres modules.

Module principal

Le module principal est le module qui sera donné à python, le programme qui sera exécuté. Ce programme fait appel aux autres modules créés.

Il est possible de détecter dans un module s'il est utilisé comme module en utilisant la variable spéciale `__name__`. Il aura la valeur `"__main__"` lorsque le module est le script d'entrée.

Il permet d'écrire des instructions qui ne seront exécutées que si le module est le script d'entrée :

```
if __name__ == '__main__':
    do_something()
```

Import des modules

Pour importer des modules situés dans le même répertoire, il suffit d'utiliser le mot-clé `import`, suivi du nom du fichier (sans l'extension ".py") :

```
import vector # Importe 'vector.py'
```

Les modules importé peuvent être renommé pour raccourcir leur nom ou pour éviter d'éventuel conflits. On utilise le mot clef `as`.

```
import vector as vec
```

Import des modules

Pour utiliser les fonctions et les variables globales d'un module importé, vous utilisez la syntaxe : `nom_module.nom_élément`.

Par exemple, pour créer une instance de Vector :

```
vector1 = vec.Vector((5,-2,1.5))
```

Import d'éléments spécifiques des modules

Il est possible d'importer des éléments spécifiques, sans importer tout le module. On utilise le mot clef `from` :

```
from vector import Vector
```

Cela vous permet d'utiliser les éléments sans spécifier le module.

```
vector1 = Vector((5,-2,1.5))
```

Import de tous les éléments d'un module

Il est possible d'importer tous les éléments en une fois à l'aide du symbol '*'.

```
from vector import *
```

Attention

Cette syntaxe peut conduire à des comportement imprévisible : cela peut importer éléments indésirables et écraser les éléments précédemment importés d'autres modules !

Exemples de modules : vector.py

```
class Vector :
    def __init__(self, coeffs):
        self.list_coeffs = coeffs
    def __str__(self):
        stringToPrint = ""
        for i in self.list_coeffs:
            stringToPrint += str(i) + " ; "
        if(stringToPrint == ""):
            stringToPrint = "[]"
        else:
            stringToPrint = "[ " + stringToPrint[:-3] + " ]"

        return stringToPrint

    def dimension(self):
        return len(self.list_coeffs)

    def get(self, index):
        try:
            return self.list_coeffs[index]
        except:
            return None
    def calculate_sum(self, vector2):
        if( not self.dimension() == vector2.dimension() ):
            raise ValueError("Different dimensions : " + str(self.dimension()) + " and " +
str(vecteur2.dimension()))
        else:
            new_list_coeffs = []
            for i in range(len(self.list_coeffs)):
                new_list_coeffs.append(self.get(i) + vector2.get(i))
            return Vector(new_list_coeffs)
```

Exemples de modules : polynomial.py

```python
from vector import Vector

class Polynomial(Vector):
    def __init__(self, coeffs):
        super().__init__(coeffs)
    def degree(self):
        return self.dimension()-1
    def __str__(self):
        stringToPrint = ""
        for i,v in enumerate(self.list_coeffs):
            stringToPrint+=str(v) + "x^" + str(i) + " + "
        if(stringToPrint == ""):
            stringToPrint = "0"
        else:
            stringToPrint = stringToPrint[:-3]

        return stringToPrint
    def evaluate(self, x):
        finalValue = 0
        for i,v in enumerate(self.list_coeffs):
            finalValue += v*x**i
        return finalValue
```

Exemples de modules : tests.py

```
from polynomial import Polynomial
from vector import Vector

if __name__ == '__main__' :
    vect1 = (5,-2,1.5)
    vect2 = ()
    vect3 = range(4,8)
    vect4 = range(1,5)
    vect5 = [10,20]
    vect6 = [-6,11,3,-2]

    v1 = Vector(vect1)
    v2 = Vector(vect2)
    v3 = Vector(vect3)
    v4 = Vector(vect4)
    v5 = Vector(vect5)

    print(v1, "\ndimension : ", v1.dimension(), "\n1er élément : ", v1.get(0))
    print(v2, "\ndimension : ", v2.dimension(), "\n1er élément : ", v2.get(0))
    print(v3, "\ndimension : ", v3.dimension(), "\n1er élément : ", v3.get(0))
    print(v5, "\ndimension : ", v5.dimension(), "\n1er élément : ", v5.get(0))
    print(v3, "\n", v4, "\nSomme des deux : ",v4.calculate_sum(v3))

    p1 = Polynomial(vect1)
    p2 = Polynomial(vect6)

    print(p1, "\n degrée :", p1.degree(), "\nvaleur du polynôme en 0 : ", p1.evaluate1(6))
    print(p2, "\n degrée :", p2.degree(), "\nvaleur du polynôme en -2 :", p2.evaluate2(-2))
```

Sous-modules

Dans de gros projets, les modules sont généralement stockés dans des sous répertoires.

```
my_project/
├── main.py
├── display/
│   ├── actions.py
│   └── graphics.py
├── loader/
│   ├── parser.py
│   ├── dumper.py
│   └── converter.py
└── engine/
    ├── structure.py
    └── operations.py
```

Sous-modules

Pour accéder aux modules qui se trouvent dans des sous-répertoires (les sous-modules), vous devez donner le chemin d'accès au module commençant au niveau du script d'entrée. Le délimiteur de chemin est le "." dans ce contexte.

Disons que "main.py" est le script d'entrée. Voilà quelque exemples :

```
import loader.parser # Then use with parser.item_name
import engine.operations as op
from loader.dumper import dump
```

Plan

Modules standards

Python est accompagné d'un ensemble de modules standard qui ne font pas partie du langage de base mais qui sont intégrés à l'interpréteur. De tels modules existent pour :

- Fournir des implémentations efficaces des opérations de base : nombres aléatoire, opérations mathématiques, collections supplémentaires, ...
- Fournir des primitives de système d'exploitation : arguments passés en ligne de commande, appels système, manipulation de chemin, ...

Liste de modules standards utiles

Module	Description
sys	Paramètres et fonctions spécifiques au système
os	Diverses interfaces du système d'exploitation
random	Génère des nombres pseudo-aléatoires
math	Fournit diverses fonctions mathématiques
argparse	Analyseur pour des options passées en ligne de commande
re	Opération d'expressions régulières
collections	Structures de données supplémentaires

Liste complète :

`https://docs.python.org/3/library/index.html`

Import de modules standards

Pour importer des modules standard, le mot-clé `import` est à nouveau utilisé.

Tous les autres mots-clés associés peuvent également être utilisés.

```
import sys
import random as rnd
from math import exp, cos, sin, tanh
from collections import defaultdict
```

Fonctions et variables utiles des modules liés au système

Les modules `sys` et `os` sont utilisés pour interagir avec le système et le système de fichiers.

Item	Description
sys.argv	Liste des arguments donnés en ligne de commande
sys.stdin	Objet gérant l'entrée standard
sys.stdout	Objet gérant la sortie standard
sys.stderr	Objet gérant le sortie d'erreur
os.listdir	Fonction listant le contenu d'un répertoire
os.mkdir	Fonction créant un nouveau répertoire
os.remove	Fonction pour supprimer un fichier
os.rmdir	Fonction pour supprimer un répertoire
os.rename	Fonction pour renommer un fichier ou un répertoire

Génération de nombres pseudo-aléatoires

Le module `random` est utilisé pour générer des séquences de nombres pseudo-aléatoires.

```
import random

a = random.randint(0, 100) # Un entier aléatoire entre 0 et 100
b = random.uniform(0, 100) # Un flottant aléatoire entre 0 et 100

l = [1,2,3]
random.shuffle(l) # Mélange les éléments de la liste en place
```

En savoir plus sur `random` à :

`https://docs.python.org/3/library/random.html`

Opérations mathématiques supplémentaires

Le module `math` est utilisé pour accéder à des fonctions et des constantes mathématiques plus avancées.

Item	description
math.exp	La fonction exponentielle
math.log	La fonction de logarithme
math.sqrt	La fonction racine carrée
math.cos	La fonction cosinus
math.sin	La fonction sinus
math.tan	La fonction tangente
math.pi	La constante mathématique π
math.inf	Utilisé pour représenter une valeur infinie

Plus d'informations sur :

`https://docs.python.org/3/library/math.html`

Plan

Pourquoi re-coder quand ça existe déjà ?

Python et les bibliothèques standard offrent de nombreuses fonctionnalités qui permettent de mettre en œuvre tout ce que vous voulez.

Cependant, la communauté Python a déjà implémenté de nombreux outils et structure de données. Ceux-ci sont disponibles via des packages qui incluent des modules Python et peuvent être utilisés comme des modules standard.

Où trouver ces formidables packages ?

La plupart des modules externes se trouvent sur le site Web `pypi.org` affilié à la Python Software Foundation.

L'utilisation d'un moteur de recherche (tel que Google) peut être utile pour voir si une fonctionnalité a déjà été incluse dans un package.

Installation d'un package (systèmes de type Unix uniquement)

La méthode recommandée pour installer un package est d'utiliser le programme d'installation de package pip. Comme l'interpréteur Python, pip a deux versions : une pour Python 2 et une pour Python 3.

Pour être sûr que le bon pip est utilisé, lancez pip avec la commande pip3.

Installation du système (droits d'administrateur requis)

```
sudo pip3 install package_name
```

Installation locale

```
pip3 install --user package_name
```

NumPy : Calculs sur des tableaux en Python

NumPy est un package fondamental pour faire du calcul scientifique avec Python.

Numpy fournit :

- un objet de tableau à N dimensions ;
- des opérations efficaces sur les vecteurs et les matrices ;
- une syntaxe adaptée à Python (slices, surcharges d'opérateur, ...).

Documentation : `https://numpy.org/doc/1.17/`
Tuto : `https://numpy.org/devdocs/user/quickstart.html`

NumPy : Création de tableaux

Les tableaux NumPy ne peuvent contenir que des objets qui sont tous du même type. Ils ont également une forme fixe (donc pas d'ajout ni de suppression possible).

À partir d'une liste existante :

```
>>> a = numpy.array([[1,2,3], [4,5,6]])
array([[1, 2, 3],
       [4, 5, 6]])
```

Avec des zéros uniquement :

```
>>> a = numpy.zeros((2,3,2), dtype=numpy.float32)
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]], dtype=float32)
```

NumPy : Basic Operations

La plupart des opérateurs arithmétiques peuvent être utilisés sur des tableaux numpy :

```
>>> a = numpy.array([1,2,3])
>>> b = numpy.array([2,4,6])
>>> a+b
array([3, 6, 9])
>>> a-b
array([-1, -2, -3])
>>> a+2
array([3, 4, 5])
>>> a**2
array([1, 4, 9])
>>> a*b  # produit élément par élément
array([ 2,  8, 18])
```

NumPy : Opérations plus avancées

Il existe des méthodes et fonctions supplémentaires qui implémentent des opérateurs avancés :

```
>>> a.dot(b)  # Produit scalaire
28

>>> c = numpy.array([[1,1],[0,1]])
>>> d = numpy.array([[2,0],[3,4]])
>>> c.dot(d)  # Produit de matrice
array([[5, 4],
       [3, 4]])
>>> d.transpose()
array([[2, 3],
       [0, 4]])
```

NumPy : Fonctions et méthodes utilitaires

Certaines fonctions utilitaires sont disponibles :

```
>>> d.sum()  # somme tous les éléments d'une matrice
9
>>> d.max()  # récupère la valeur max dans la matrice
4
>>> d.mean()  # calcule la valeur moyenne de la matrice
2.25
>>> d[1,0]  # accède à l'élément de la deuxième ligne et de la première
0
>>> d[:,1]  # accède à tous les éléments de la deuxième colonne
array([0, 4])
```

Matplotlib : Graphiques 2D de qualité

Matplotlib is a Python 2D plotting library that creates good quality plots that are easily customised.

It can create any type of plot :

Matplotlib est une bibliothèque permettant de créer des graphiques Python 2D de bonne qualité facilement personnalisables.

On peut créer tout type de tracé :

- graphique linéaire, nuage de points, graphique à barres, ... ;
- animé ou pas ;
- avec plusieurs types de graphiques combinées si nécessaire ;
- peut générer différents formats (png, pdf, svg, ...) ;
- fonctionne bien avec NumPy.

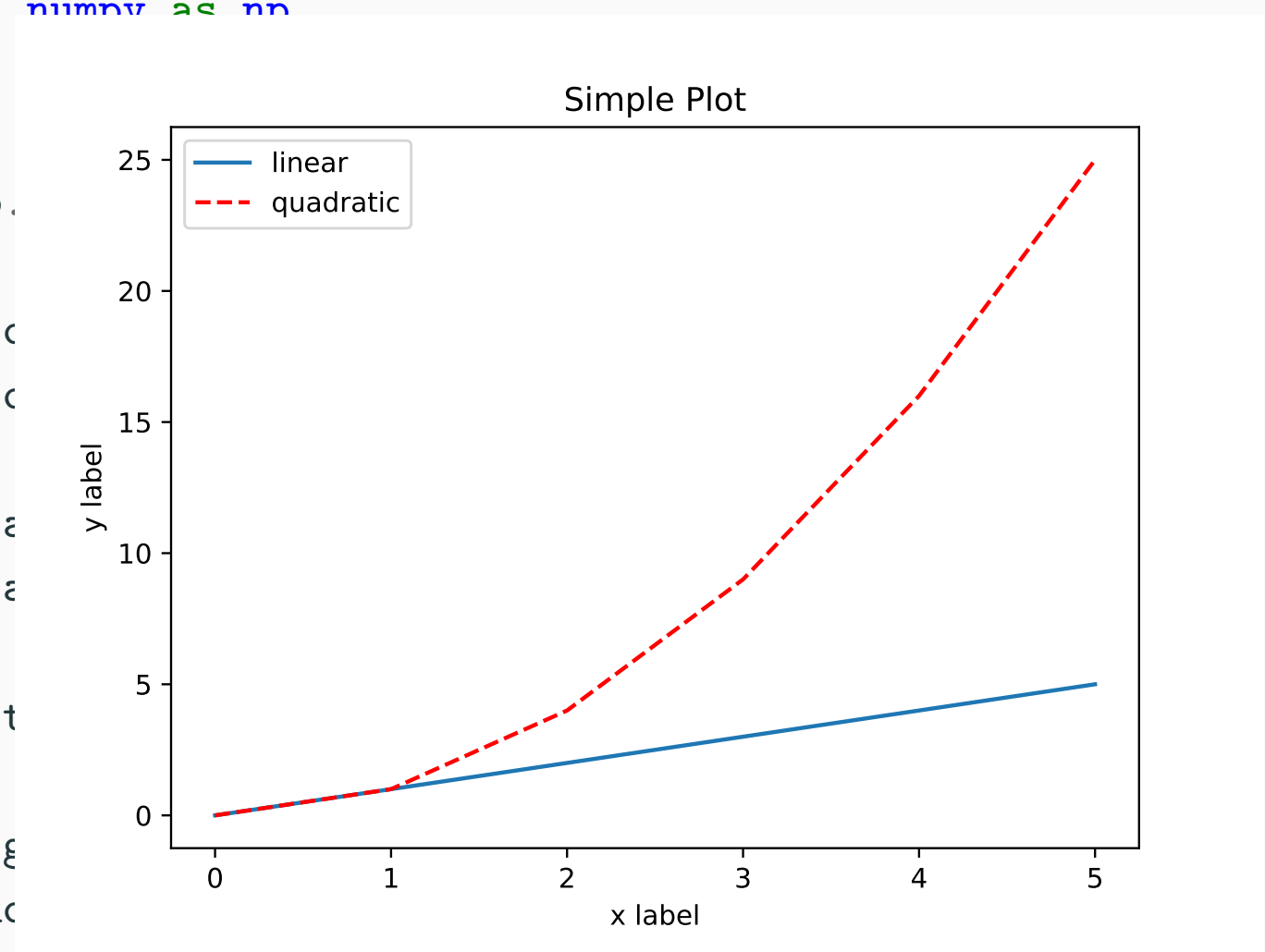
Documentation : `https://matplotlib.org/api/pyplot_summary.html`
Tutor : `https://matplotlib.org/tutorials/`

Matplotlib : Line Plots

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

x = np.array([0,1,2,3,4,5])

plt.plot(x, x, label='linear')
plt.plot(x, x**2, 'r--', label='quadratic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()
plt.show()
```

Matplotlib : Line Plots

```
import numpy as np
import

x = np.

plt.plo
plt.plo

plt.xla
plt.yla

plt.tit

plt.leg
plt.sho
```

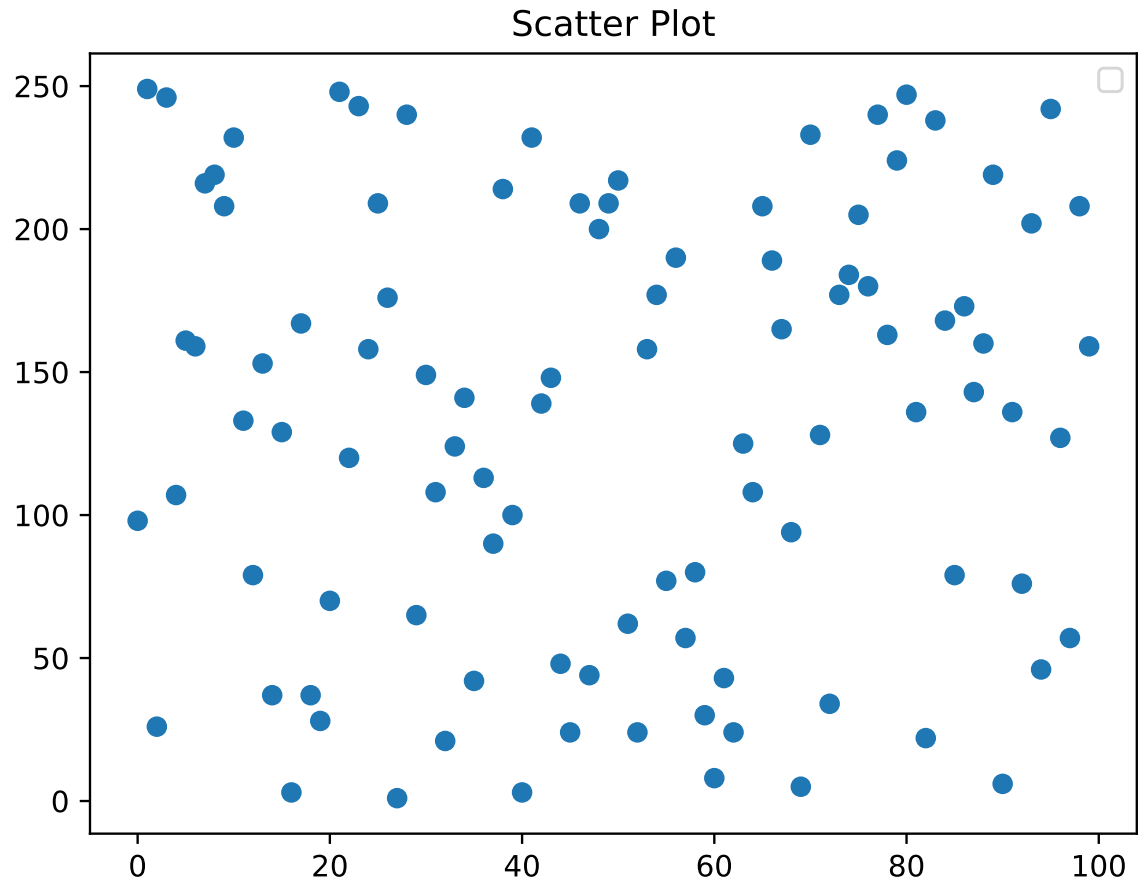


Matplotlib : Scatter Plots

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

x = np.arange(100)
y = np.random.randint(0, 250, 100)

plt.scatter(x, y)

plt.title("Scatter Plot")

plt.legend()
plt.show()
```

Matplotlib : Scatter Plots

```
import
import

x = np.
y = np.

plt.sca

plt.tit

plt.leg
plt.sho
```



Matplotlib : Bar Plots

```
import matplotlib.pyplot as plt

cities = ['Marseille', 'Paris', 'Lyon', 'Toulouse']
density = [3583, 20781, 10773, 4019]

plt.bar(cities, density)

plt.xlabel('City')
plt.ylabel('pop./km^2')

plt.title("Bar Plot")

plt.legend()
plt.show()
```

Matplotlib : Bar Plots

```
import

cities
density

plt.bar

plt.xla
plt.yla

plt.tit

plt.leg
plt.sho
```