

Cours POO M1: python

03 Octobre 2023

- **TP noté le 16 Octobre**
- Tous les documents autorisés + internet.
- CM 5 et TP4 = derniers cours avant évaluation (\Rightarrow anticipez vos questions).

Les classes en Python (suite)

Déjà vu jusqu'à présent

- constructeur de classe: `__init__`

Déjà vu jusqu'à présent

- constructeur de classe: `__init__`
- attributs publics et privés

Déjà vu jusqu'à présent

- constructeur de classe: `__init__`
- attributs publics et privés
- définition (surcharge) d'opérateurs: `__add__`, `__sub__`, ..., `__eq__`, ..., `__str__`

Héritage de classe

- Considérons la classe *Vehicule*. Un véhicule est défini par ses *attributs* : nombre de roues, sa vitesse maximale, le nombre de passagers qu'il peut transporter. La classe peut aussi posséder des méthodes.

Héritage de classe

- Considérons la classe *Vehicule*. Un véhicule est défini par ses *attributs* : nombre de roues, sa vitesse maximale, le nombre de passagers qu'il peut transporter. La classe peut aussi posséder des méthodes.

Héritage de classe

- Considérons la classe *Vehicule*. Un véhicule est défini par ses *attributs* : nombre de roues, sa vitesse maximale, le nombre de passagers qu'il peut transporter. La classe peut aussi posséder des méthodes.
- Une voiture *est* un *Vehicule* (et donc caractérisé par les points précédents) et peut être caractérisé aussi par son nombre de portes.

Héritage de classe

- Considérons la classe *Vehicule*. Un véhicule est défini par ses *attributs* : nombre de roues, sa vitesse maximale, le nombre de passagers qu'il peut transporter. La classe peut aussi posséder des méthodes.
- Une voiture *est* un *Vehicule* (et donc caractérisé par les points précédents) et peut être caractérisé aussi par son nombre de portes.
- On peut donc définir une classe *Voiture* qui **hérite** de la classe *Vehicule* (attributs et méthodes) et qui aura l'attribut supplémentaire "nombre de portes"

Héritage (exemple)

✓ signifie "peut être défini pour tout objet de la classe"

✗ signifie "ne peut pas être défini pour tout objet"

Classe	Vehicule	Voiture	Velo
nombre de roues	✓	✓	✓
vitesse max	✓	✓	✓
nombre de passagers	✓	✓	✓
nombre de portes	✗	✓	✗
type de dérailleur	✗	✗	✓

- Toute instance de la classe *Vehicule* possède les attributs "nombre de roues", "vitesse max" et "nombre de passagers"
- Toute instance de la classe *Voiture* possède les attributs de la classe *Vehicule* ainsi que "nombre de portes"
- Toute instance de la classe *Velo* possède les attributs de la classe *Vehicule* ainsi que "type de derailleur"
- Les classes *Voiture* et *Velo* **héritent** de la classe *Vehicule*.

Héritage

- Considérons deux classes: *ClasseP* (P pour parent) et *ClasseF* (F pour fille)
- Considérons que *ClasseF* hérite de *ClasseP*
- La déclaration de l'héritage se fait comme suit

```
class ClasseF(ClasseP):  
    """ ClasseF hérite de ClasseP """  
    def __init__(self, ...):      # constructeur  
        de ClasseF  
        ClasseP.__init__(self, ...)  
        ..
```

- L'appel au constructeur de *ClasseP* permet d'initialiser les attributs communs (*ClassP.__init__* fonctionnerait même sans héritage).
- Toute instance de *ClasseF* possède les méthodes de *ClasseP*.

Exemple d'héritage

Dans le même fichier python d'extension .py,

```
class ClasseP:
    def __init__(self, i):
        self.a = i
    def function_P(self):
        print('Classe parent: ClasseP')

class ClasseF(ClasseP):
    def __init__(self, i, j):
        ClasseP.__init__(self, i) #Applique le
                                #constructeur de ClasseP à l'instance de
                                #ClasseF
        self.b = j
    def function_F(self):
        print('Classe fille: ClasseF')
```

Exemple d'héritage

```
op = ClasseP(4)
print("op.a=", op.a)
op.function_P()
```

op = ClasseP(4); op.a 4
Classe parent: ClasseP

```
of = ClasseF(2, 1)
print("of.a=", of.a, "of.b=", of.b))
of.function_P()
of.function_F()
```

of.a= 2 of.b= 1
Classe parent: ClasseP
Classe fille: ClasseF

⇒ l'instance *of* de *ClasseF* possède la méthode *function_P* !

Héritage multiple en python

Possible d'avoir de l'héritage multiple en python

```
class ClasseC(ClasseA, ClasseB):  
    def __init__(self, ...):  
        ClasseA.__init__(self, ...)  
        ClasseB.__init__(self, ...)
```

Attention aux éventuels conflits (méthodes avec le même nom dans les classes parentes).

Remarque : une méthode de la classe mère peut être redéfinie (avec le même nom) dans la classe fille (voir jupyter notebook).

Attributs protégés

- Parfois, on veut indiquer à l'utilisateur qu'un attribut devrait être considéré comme privé, sans pour autant en empêcher l'accès.
- Pour cela, on définit des attributs protégés.
- En python, on préfixe par "_" les attributs protégés (c'est seulement une convention et ça n'entraîne pas d'erreur au contraire des attributs privés) : les données protégées sont accessibles normalement et le préfixe a pour seul objectif d'informer sur leur nature.

Attributs protégés

```
class ClasseP2: # classe Parent - 2ème exemple
    def __init__(self, i):
        self.a = i # attribut public
        self._beta = 2*i-3 # attribut protégé
        self.__gamma = 4*i
```

```
>>> p=ClasseP2(2)
>>> print("Attributs publics et protégés : ", p.a, p._beta)
Attributs publics et protégés : 2 1
>>> print("Attribut privé ",p.__gamma)
AttributeError: 'ClasseP2' object has no attribute '__gamma'
```

scipy

- *Scipy* contient *numpy* (certaines fonctions sont redéfinies pour les rendre plus souples)
- *Scipy* contient des sous-packages qui contiennent des algorithmes couramment utilisés en calcul scientifique pour l'algèbre linéaire, les statistiques, la résolution numérique d'EDO, l'intégration numérique, ...

- *Scipy* contient *numpy* (certaines fonctions sont redéfinies pour les rendre plus souples)
- *Scipy* contient des sous-packages qui contiennent des algorithmes couramment utilisés en calcul scientifique pour l'algèbre linéaire, les statistiques, la résolution numérique d'EDO, l'intégration numérique, ...
- *Numpy* est écrit en *C* et est plus rapide que *Scipy*, écrit en python. *Numpy* est préférable pour des opérations simples sur des tableaux.

Un exemple de différence scipy/numpy

- Les fonctions mathématiques *sqrt*, *log* dans *scipy* peuvent s'appliquer à des nombres complexes.

```
import scipy as sp
print(sp.sqrt(-1)) # renvoie 1j
print(sp.sqrt(144)) # renvoie 12.0

a = sp.log(-2); print(a) #
    (0.6931471805599453+3.141592653589793j)

print(sp.log(2)) # 0.6931471805599453
print(sp.exp(a)) # (-2+2.44929359829470e-16j)
```

scipy

<code>scipy.linalg</code>	Algèbre linéaire
<code>scipy.stats</code>	Statistiques
<code>scipy.integrate</code>	Résolution d'EDO et intégration numérique
<code>scipy.interpolate</code>	Interpolation numérique
<code>scipy.fftpack</code>	Transformées de Fourier
<code>scipy.optimize</code>	Optimisation mathématique
<code>scipy.sparse</code>	Gestion des matrices creuses
<code>scipy.special</code>	Fonctions spéciales
<code>scipy.signal</code>	Traitement du signal
<code>scipy.ndimage</code>	Traitement d'images
<code>scipy.io</code>	Entrées-sorties

Table: Principaux sous-packages de *scipy*

scipy.linalg

scipy.linalg contient beaucoup de fonctions faisant les opérations élémentaires sur des matrices (plus de fonctions et plus d'options que *numpy.linalg*)

- `scipy.linalg.norm` → normes de vecteurs ou de matrices
- `scipy.linalg.det` → calcul du déterminant
- `scipy.linalg.lstsq` → résolution par moindres carrés
- `scipy.linalg.chol` → factorisation de Cholesky
- `scipy.linalg.qr` → factorisation QR
- `scipy.linalg.eig` → calcul des valeurs et vecteurs propres
- `scipy.linalg.eigvals` → calcul des valeurs propres uniquement
- `scipy.linalg.solve` → résolution de systèmes linéaires
- `scipy.linalg.pinv` → calcul du pseudo-inverse
- `scipy.linalg.inv` → calcul de l'inverse
- `scipy.linalg.expm` → exponentielle de matrice

scipy.stats

- Permet la simulation d'un certain nombre de variables aléatoires discrètes et continues
- Contient des fonctions pour les Statistiques (tests, coefficients de corrélation, ...)
- Plus riche que *numpy.random*. Les noms de fonctions changent entre *numpy.random* et *scipy.stats*!

Variables aléatoires et *scipy.stats*

scipy.stats fournit un certain nombre de fonctions. Pour chaque loi de probabilité, on retrouve les mêmes noms de fonctions

- **pdf** = *probability density function* = densité de probabilité f (pour les lois continues)
- **pmf** = *probability mass function* = fonction de masse (pour les lois discrètes)
- **cdf** = *cumulative distribution function* = fonction de répartition F
- **ppf** = *percent point function* = fonction quantile (réciproque de la fonction de répartition)
- **rvs** = tirage pour une variable aléatoire

Rappels

Soit X une variable aléatoire à densité.

La fonction de densité f vérifie

$$\forall a < b \in \mathbb{R}, \mathbb{P}(a \leq X \leq b) = \int_a^b f(x) dx$$

La fonction de répartition vérifie

$$\begin{aligned} F : \mathbb{R} &\rightarrow [0, 1] \\ F(x) &= \int_{-\infty}^x f(s) ds \end{aligned}$$

et la fonction quantile

$$\begin{aligned} Q : [0, 1] &\rightarrow \mathbb{R} \\ Q(x) &= \inf \{y \in \mathbb{R} \mid x \leq F_X(y)\}, \quad 0 \leq x \leq 1 \end{aligned}$$

En particulier, $Q = F^{-1}$ si F^{-1} existe.

Variables aléatoires et *scipy.stats*

Avec *scipy*, paramétrer un loi de probabilité n'est pas toujours instinctif...

- Pour les loi continues, on a par défaut les arguments *loc*= 0, *scale*= 1.
- Cette paramétrisation par défaut correspond, pour chaque famille de lois, à une loi de référence ($\mathcal{N}(0, 1)$ pour normale, $\mathcal{U}_{[0,1]}$ pour uniforme, $\mathcal{E}(1)$ pour exponentielle...)
- Notons \hat{f} la densité d'une loi de référence, en modifiant les arguments *scale* et *loc*, on obtient la loi de densité

$$f(x) = \frac{1}{scale} \hat{f}\left(\frac{x - loc}{scale}\right)$$

Il faut donc connaître la loi de référence puis réaliser un petit calcul
Sinon : consulter l'aide de *scipy.stats.nom_de_la_loi* !

Loi normale

On s'intéresse à X , de loi $\mathcal{N}(m, \sigma^2)$. La densité f de X est:

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-m)^2}{2\sigma^2}}$$

D'après l'aide (? `scipy.stats.norm`), la loi de référence est $\mathcal{N}(0, 1)$, *loc*=moyenne et *scale*=écart-type. Pour la loi $\mathcal{N}(m, \sigma^2)$ on fera donc :

```
from scipy.stats import norm
norm.pdf(x, loc=m, scale=sigma)           # f(x)
norm.cdf(x, loc=m, scale=sigma)           # F(x)
norm.ppf(x, loc=m, scale=sigma)           # F^{-1}(x)
norm.rvs(size=n, loc=m, scale=sigma)      # n tirages
```

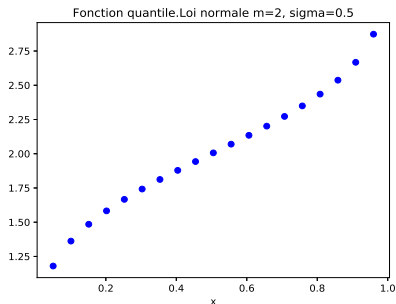
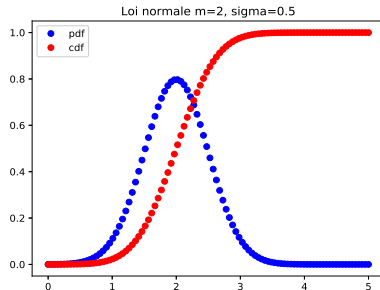
On aurait aussi pu trouver *loc* et *scale* pour que

$$f(x) = \frac{1}{scale} \hat{f}\left(\frac{x - loc}{scale}\right) \quad \text{avec} \quad \hat{f}(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Loi normale

```
import scipy as sp
from scipy.stats import norm
import matplotlib.pyplot as plt

m = 2; sigma = 0.5
x = sp.linspace(0, 5, 100)
plt.plot(x, norm.pdf(x, m, sigma),
         'ob', label='pdf')
plt.plot(x, norm.cdf(x, m, sigma),
         'or', label='cdf')
plt.title('Loi normale m=2, '+
         ' sigma=0.5')
plt.xlabel('x'); plt.legend()
plt.show()
plt.figure()
plt.plot(x, norm.ppf(x, m, sigma),
         'ob', label='ppf')
plt.title('Fonction quantile.' +
         'Loi normale m=2, sigma=0.5')
plt.xlabel('x')
plt.show()
```



Loi uniforme

Supposons que X suit $\mathcal{U}(a, b)$. La densité de X est:

$$f(x) = \frac{1}{(b-a)} \mathbb{1}_{[a,b]}(x)$$

Les arguments à saisir sont $loc=a$ et $scale=b-a$ (on peut remarquer que $loc \neq$ moyenne et $scale \neq$ écart-type cette fois).

```
from scipy.stats import uniform
uniform.pdf(x, loc=a, scale=(b-a)) # f(x)
uniform.cdf(x, loc=a, scale=(b-a)) # F(x)
uniform.ppf(x, loc=a, scale=(b-a)) # (quantile)
uniform.rvs(size=n, loc=a, scale=(b-a)) # n
tirages
```

Loi exponentielle

Supposons que X suit $\mathcal{E}(\theta)$. La densité de X est:

$$f(x) = \theta e^{-\theta x} \mathbb{1}_{[0, +\infty[}(x)$$

Les arguments à saisir sont $loc=0$ et $scale=\frac{1}{\theta}$.

```
from scipy.stats import expon
expon.pdf(x, loc=0, scale=1/theta)      # f(x)
expon.cdf(x, loc=0, scale=1/theta)      # F(x)
expon.ppf(x, loc=0, scale=1/theta)      # quantile
expon.rvs(size=n, loc=0, scale=1/theta) # tirages
```

Le $loc=0$ n'est pas nécessaire.

Loi du χ^2

Supposons que X suit une loi du χ^2 à k degrés de liberté. La densité de X est:

$$f(x) = \frac{1}{2^{k/2} \Gamma(k/2)} x^{k/2-1} e^{-x/2}, \quad \forall x > 0$$

```
from scipy.stats import chi2
chi2.pdf(x, df=k)
chi2.cdf(x, df=k) #loc=0, scale=1
chi2.rvs(x, df=k, size=n) #loc=0, scale=1
chi2.ppf(x, df=k) #loc=0, scale=1
```

Inutile de renseigner `loc` et `scale` (cela donnerait une loi différente, qui n'a pas de nom). `df` n'a pas de valeur par défaut.

Loi de Poisson

Soit X qui suit $\mathcal{P}(\lambda)$.

$$\mathbb{P}(X = k) = \frac{\mu^k}{k!} e^{-\lambda}$$

Pas de densité (loi discrète).

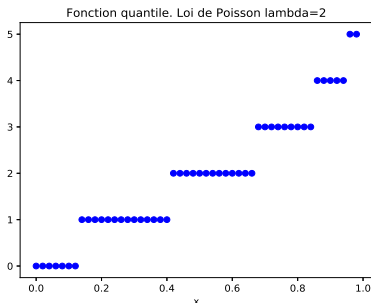
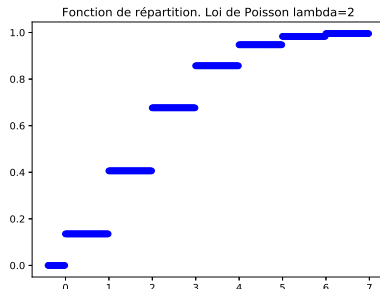
```
from scipy.stats import poisson
poisson.cdf(x, lmbda)
poisson.rvs(mu, size=n)
poisson.ppf(x, lmbda)
```

Ici encore, le paramètre `loc` existe mais produirait une loi qui n'est plus la loi de Poisson.

Loi de Poisson

```
import scipy as sp
from scipy.stats import poisson
import matplotlib.pyplot as plt

lbda = 2.0
x = sp.arange(-0.4, 7, 0.02)
plt.plot(x, poisson.cdf(x, lbda),
         'ob')
plt.title('Poisson lambda=2')
plt.title('Fonction de repartition.'
         + ' Loi de Poisson lambda=2')
plt.xlabel('x')
plt.show()
plt.figure()
plt.plot(x, poisson.ppf(x, lbda),
         'ob')
plt.title('Fonction quantile.'
         + ' Loi de Poisson lambda=2')
plt.xlabel('x')
plt.show()
```



Loi Binômiale

X suit une loi binômiale $\mathcal{B}(n, p)$

$$\mathbb{P}(X = k) = C_k^n p^k (1 - p)^{n-k}, \quad k \text{ dans } \{0, \dots, n\}$$

```
from scipy.stats import binom
binom.cdf(x, n, p)
binom.rvs(n, p, size=m)
binom.ppf(x, n, p)
```

Tests statistiques

scipy.stats fournit de nombreuses fonctions de tests statistiques:

```
from scipy.stats import ks_2samp, kstest  
from scipy.stats import shapiro
```

Présentation de *scipy.integrate*

- intégration numérique avec plusieurs types de méthodes
 - *scipy.integrate.quad* (approximation numérique de l'intégrale d'une fonction)
 - *scipy.integrate.trapz* (méthodes des trapèzes, avec tableaux numpy)
 - *scipy.integrate.simps* (méthodes de Simpson, avec tableaux numpy)
 - et d'autres: *scipy.integrate.quadrature*, *scipy.integrate.fixed_quad*, *scipy.integrate.dblquad*, ...

- résolution d'Equations Différentielles Ordinaires
 - *scipy.integrate.odeint* ou *scipy.integrate.solve_ivp*
 - il y a aussi *scipy.integrate.ode*

Intégration numérique avec *scipy.integrate*

Exemple sur la méthode de Simpson (interpolation polyn d'ordre 2 sur les bords et au centre de la maille)

```
import scipy.integrate as spi
import numpy as np
f = lambda x: 4/(1+x**2)

n = 6; x = np.linspace(0, 1, n); y = f(x)
I = spi.simps(y,x) # x= intervalle, y = image de
    x par la fonction à intégrer,
print("n=6", I, abs(I-spi.pi))

n = 21; x = np.linspace(0, 1, n); y = f(x)
I = spi.simps(y,x)
print("n=21", I, abs(I-spi.pi))
```

La solution exacte est $4(\arctan(1) - \arctan(0)) = \pi$

n=6 3.139679084417619 0.0019135691721743342

n=21 3.141592652969785 6.200080449048073e-10

Résolution d'EDO avec *scipy.integrate*

scipy.integrate.odeint permet de résoudre des EDO du 1er ordre, vérifiant:

$$\begin{cases} \frac{dy(t)}{dt} = f(t, y(t)) \\ y(t_0) = y_0 \end{cases}$$

Par exemple, on veut résoudre

$$\begin{cases} \frac{dy(t)}{dt} = -2y(t) \\ y(t_0) = y_0 = 1 \end{cases}$$

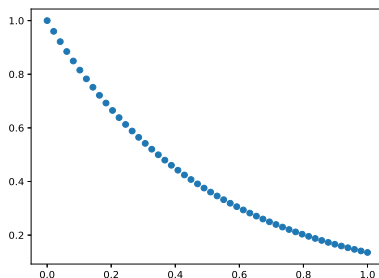
Résolution d'EDO avec *scipy.integrate*

Pour l'exemple précédent,

```
import scipy.integrate as spi
t = np.linspace(0, 1, 50)
yini = np.array([1])

def f(y, t): # ordre !!
    return -2*y

x = spi.odeint(f, y0, t)
plt.plot(t,x, 'o')
```



Résolution d'EDOs du second ordre avec *scipy.integrate*

- On veut résoudre, par exemple:

$$\begin{cases} \frac{d^2 y(t)}{dt^2} = -4 y(t) \\ y(0) = 0, \quad \frac{dy}{dt}(0) = 2 \end{cases}$$

- On se ramène à une EDO du 1er ordre en posant:

$$Y(t) = \begin{pmatrix} y(t) \\ \frac{dy}{dt}(t) \end{pmatrix} \quad \frac{dY}{dt}(t) = \begin{pmatrix} \frac{dy}{dt}(t) = Y_1(t) \\ \frac{d^2 y}{dt^2}(t) = -4 y(t) = -4 Y_0(t) \end{pmatrix}$$

et donc

$$Y(0) = [0, 2]$$

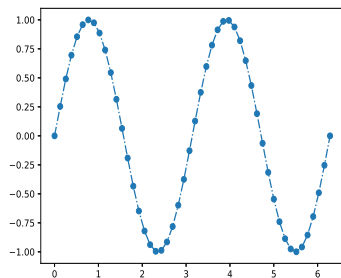
- On peut généraliser pour les ordres supérieurs.

Résolution d'EDOs du second ordre avec *scipy.integrate*

```
import scipy.integrate as spi
t = np.linspace(0, 2*np.pi, 50)
yini = np.array([0, 2])

def f(Y, t): # ordre !!
    return np.array([Y[1], -4*Y[0]])

Y = spi.odeint(f, yini, t)
plt.plot(t, Y[:,0], 'o', ls='-.')
```



Présentation de *scipy.interpolate*

Méthodes d'interpolation numérique

- *scipy.interpolate.interp1d* - interpolation 1D
- et d'autres: *scipy.interpolate.interp2d* - interpolation 2D

renvoie une fonction polynomiale par morceaux passant par les points de coordonnées (x_i, y_i) passés en arguments.

Interpolation

A partir d'un nuage de points, *scipy* calcule la fonction d'interpolation.

```
import scipy.interpolate as spi
n = 20; f0 = lambda x: x**3 + 1
x = np.linspace(0, 1, n)
y = f0(x)

eps = np.random.rand(n)
eps = eps/2. # pour se ramener à [0, 0.5]
y = y+ eps
f1 = spi.interpolate.interp1d(x, y)

x2 = np.linspace(0, 1, 2*n-1)
plt.plot(x, y, 'o')
#plt.plot(x2, f1(x2))
plt.plot(x2, f1(x2), 'r.')
```

