Cours POO M1: python

26 Septembre 2023

import et docstring

Commande *import*

On peut importer un module, des fonctions ou des variables (ou des classes).

- On a déjà vu la commande *import* pour importer un module ou un sous-package afin d'avoir accès à des fonctions ou des variables

```
import numpy
import numpy as np
import numpy.linalg
import numpy.linalg as nl
from numpy import random
```

- On peut aussi importer des fonctions, des variables d'un fichier (local)

```
import monfichier #import du contenu de monfichier.py
                  #on n'écrit pas l'extension .py
monfichier.f      #fonction f dans monfichier.py
from monfichier import f
```

Le fichier doit se trouver dans le répertoire courant.

Fichier python - *import*

- Quand on importe un fichier, celui-ci s'exécute. Pour éviter cela, on utilise la syntaxe suivante dans le fichier sur la partie de code dont on ne veut pas qu'elle soit exécutée à l'import:

```
# contenu de hello.py
def f(t):
    .... # corps de la fonction f

if __name__ == "__main__": # Cette partie n'est
    pas exécutée à l'importation
        print("Hello world!")
```

```
import hello # La fonction f est importée (on l'
    utilisera par hello.f), pas d'exécution de
    print("Hello world!")
```

Docstring (documentation string)

En Python, on peut documenter un fichier, une fonction ou une classe.

- Les délimiteurs de la documentation sont trois guillemets (""""), au début et à la fin de la portion de documentation.
- Exemple avec le fichier hello.py contenant le code suivant et enregistré dans le répertoire courant :

```
#Contenu de hello.py
""" Fichier hello.py affichant Hello world! """
print("Hello world!")
```

On accède à la documentation par la commande help()

```
import hello
help(hello) # ? hello sous ipython
```

```
Help on module hello:
NAME
    hello - Fichier hello.py affichant Hello world!
FILE
    /tmp/hello.py
```

Docstring (documentation string)

- Possibilité de mettre de la documentation sur différentes lignes
- Contenu du fichier hello.py contenant une documentation pour le fichier lui-même et une pour la fonction f (sur la 1ère ligne suivant l'entête) :

```
""" Fichier hello.py affichant Hello world! """
def f():
    """
      Fonction f affichant Hello world!
    """
    print("Hello world!")
```

Docstring (documentation string)

```
import hello
help(hello)
```

```
Help on module hello:
NAME
    hello - Fichier hello.py affichant Hello world!
FILE
    /tmp/hello.py
FUNCTIONS
    f()
        Fonction f affichant Hello world!
```

```
help(hello.f)
```

```
Help on function f in module hello:
f()
    Fonction f affichant Hello world!
```

Nombres complexes

Les nombres complexes avec python

- Le nombre complexe $i = \sqrt{-1}$ se note en python *1j*
- $2i$ se note en python *2j*
- $1.5i$ se note *1.5j*

Les nombres complexes avec python

- Le nombre complexe $i = \sqrt{-1}$ se note en python *1j*
- $2i$ se note en python *2j*
- $1.5i$ se note *1.5j*
- Opérations élémentaires directement possibles
- Un nombre complexe a le type *complex*

```
a = 1 + 2j
b = -2 + 3j
print(a, a+b, a*b)
```

```
(1+2j) , (-1+5j) , (-8-1j)
```

Si x est un objet pour lequel $x * i$ a un sens (float, array de floats,...), **la syntaxe est** `x*1j` et non `x*j` (ni `xj` bien sûr).

Exemples avec les nombres complexes

- Parties réelle et imaginaire s'obtiennent avec *.real*, *.imag*
- Le module s'obtient avec *abs*.

```
a = 1 + 2j
print(a.real, a.imag, abs(a))
```

```
1.0
2.0
2.23606797749979
```

Les classes en Python

Intérêt d'utiliser les classes

L'utilisation de classes est très courante en programmation orientée objet.
Plusieurs avantages:

- Facilité de programmation de notions complexes
- Lecture et transmission du code
- Encapsulation (attributs et méthodes privés)
- Surcharge des opérateurs
- Héritage de classes (sous-classes)
- ...

Les classes en Python

Classe Une *classe* est un nouveau *type* d'objets, créé par l'utilisateur.

Instance Un objet créé à partir de cette classe, portant donc ce nouveau type, s'appelle une *instance* de la classe.

En Python, pour une classe donnée, il est possible de définir

- un **constructeur** de classe (méthode *__init__*) appelé à la création d'une instance de la classe.

Les classes en Python

Classe Une *classe* est un nouveau *type* d'objets, créé par l'utilisateur.

Instance Un objet créé à partir de cette classe, portant donc ce nouveau type, s'appelle une *instance* de la classe.

En Python, pour une classe donnée, il est possible de définir

- un **constructeur** de classe (méthode *__init__*) appelé à la création d'une instance de la classe.
- des **attributs** (*i.e* des propriétés) de classe

Les classes en Python

Classe Une *classe* est un nouveau *type* d'objets, créé par l'utilisateur.

Instance Un objet créé à partir de cette classe, portant donc ce nouveau type, s'appelle une *instance* de la classe.

En Python, pour une classe donnée, il est possible de définir

- un **constructeur** de classe (méthode *__init__*) appelé à la création d'une instance de la classe.
- des **attributs** (*i.e* des propriétés) de classe
- des **méthodes**, c'est-à-dire fonctions qui sont propres à la classe.

Les classes en Python

Classe Une *classe* est un nouveau *type* d'objets, créé par l'utilisateur.

Instance Un objet créé à partir de cette classe, portant donc ce nouveau type, s'appelle une *instance* de la classe.

En Python, pour une classe donnée, il est possible de définir

- un **constructeur** de classe (méthode *__init__*) appelé à la création d'une instance de la classe.
- des **attributs** (*i.e* des propriétés) de classe
- des **méthodes**, c'est-à-dire fonctions qui sont propres à la classe.
- des **surcharges** (*i.e.* des redéfinitions) des opérateurs $+, *, ==, (), ...$

Comment définir une classe en Python?

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- *class* est le mot-clé permettant de définir une classe
- il faut respecter l'indentation
- Les différents points ou <statement> correspondre à des déclarations d'*attributs* ou à des définitions de *méthodes*.

Constructeur de classe *__init__*

La méthode *__init__* correspond au constructeur de classe. Elle est appelée à chaque création d'une instance de la classe pour définir ses attributs.

- À la création d'une instance, ses attributs sont saisis par l'utilisateur. Le constructeur *__init__* peut aussi définir des attributs par défauts.
- Le constructeur *__init__* possède au moins un argument, conventionnellement nommé **self**, qui se réfère à l'instance elle-même.
- *__init__* est une fonction qui ne renvoie rien.
- S'il n'est pas définit, Python le considère par défaut comme une fonction vide.

1er exemple de classe

On va créer la classe *MyComplex* afin de manipuler des nombres complexes. Ils existent déjà en python, mais c'est un exemple simple pour découvrir les classes.

1er exemple de classe

On va créer la classe *MyComplex* afin de manipuler des nombres complexes. Ils existent déjà en python, mais c'est un exemple simple pour découvrir les classes.

- Toute instance de la classe *MyComplex* doit posséder les attributs **r** et **i** (parties réelle et imaginaire). On souhaite aussi définir des méthodes pour la classe (**module** d'un nombre complexe) et redéfinir les **opérateurs** habituels.

Classe *MyComplex* et constructeur *__init__*

```
class MyComplex:
    """ Classe pour les nombres complexes """
    def __init__(self, reel=0, imag=0):
        self.r = reel
        self.i = imag
```

Pour créer une instance de *MyComplex*, saisir les arguments demandés par *__init__* (sauf le *self*). Sinon, valeurs par défaut :

```
a0 = MyComplex()
a1 = MyComplex(1)
a2 = MyComplex(imag=-3)
a3 = MyComplex(5, 7)
a4=MyComplex(imag=7,reel=5)    # équivalent
```

Classe *MyComplex* et constructeur *__init__*

La fonction "*print*" n'a pas été configurée pour afficher le contenu d'un objet de classe MyComplex !

- `print(a0)` renvoie `<__main__.MyComplex object at 0x000002030A43D760>` (où le code correspond à `id(a0)` en hexadécimal.
- On peut faire `print(a0.r, a0.i)` puisque *print* est bien définie pour ces types (float,int,...)

Classe *MyComplex* et constructeur *__init__*

Un exemple sans valeur par défaut pour *reel* : l'utilisateur doit la saisir.
Syntaxe: d'abord les arguments sans valeur par défaut, puis les autres.

```
class MyComplexBis:
    """ Classe pour les nombres complexes """
    def __init__(self, reel, imag=0): # /!\ reel
        devra être saisi lors de l'instanciation.
        self.r = reel
        self.i = imag
```

```
a0 = MyComplexBis(1,2); print(a0.r, a0.i)
# affiche  1 2
a1 = MyComplexBis(1); print(a1.r, a1.i)
# affiche  1 0
a2 = MyComplexBis(imag=-3)
# TypeError: __init__() missing 1 required
  positional argument: 'reel'
```

Classe *MyComplex*: définition d'une méthode

- Création de la méthode *module*

```
from math import sqrt
class MyComplex:
    """ Classe pour les nombres complexes """
    def __init__(self, reel=0, imag=0):
        self.r = reel
        self.i = imag
    def module(self):
        return sqrt(self.r**2 + self.i**2)
```

```
>>> a = MyComplex(2, 4) ;  print(a.module())
4.47213595499958
```

Classe *MyComplex*: méthode *rotation*

Une méthode ne **renvoie** par toujours un résultat (via *return*). Elle peut par exemple **afficher** ou **tracer** quelque chose, ou encore **modifier l'instance**.

```
from math import , cos, sin
class MyComplex:
    def __init__(self, reel=0, imag=0):
        self.r = reel
        self.i = imag
    def rotation(self,theta):
        p=self.r
        self.r=self.r*cos(theta)-self.i*sin(theta)
        self.i=self.i*cos(theta)+p*sin(theta)
```

```
>>> from math import pi
>>> a = MyComplex(3, 0); print(a.r,a.i)
>>> a.rotation(pi); print(a.i,a.r)
-3.6739403974420594e-16 -3.0
```

Classe *MyComplex*: méthode *rotation*

Noter la différence avec la version méthode *rotation2*, qui ne *modifie pas l'instance* et *renvoie* un nouvel objet.

```
class MyComplex:
        ...
    def rotation(self,theta):
        self.r=self.r*cos(theta)-self.i*sin(theta)
        self.i=self.i*cos(theta)+self.r*sin(theta)
    def rotation2(self,theta):
        x=self.r*cos(theta)-self.i*sin(theta)
        y=self.i*cos(theta)+self.r*sin(theta)
        return MyComplex(x,y)
```

```
>>> a = MyComplex(3, 0) ;  a.rotation(pi)
>>> b=a.rotation2(pi)  # a n'est pas modifié au passage
 print(a.i,a.r) ; print(b.i,b.r)
-3.6739403974420594e-16 -3.0 # dans les deux cas
```

2ème exemple de classe

- Le fichier MaClasse.py contient

```
class MaClasse:
    s = "Hello world!"
    def f(self):
        return "Bonjour"
```

- *s* est un **attribut constant** de la classe. Toute instance de la classe possède l'attribut *s* dont la valeur ne dépend pas de l'instance.
- *f* est une méthode de la classe. Toute instance de la classe possède la méthode *f*

2ème exemple de classe

```
o = MaClasse()
print(o.s)   # renvoie "Hello world!"
o.f()        # execute f, donc renvoie "Bonjour"
```

- La variable *o* est une **instance** (ou un objet) de la classe *MaClasse*
- *MaClasse()* correspond à l'appel au **constructeur** de la classe MaClasse (qui n'est pas défini). C'est donc ici le constructeur par défaut (qui ne fait rien)
- L'objet *o*, qui instancie la classe *MaClasse* possède l'**attribut** *s* et la **méthode** *f*.

2ème exemple de classe

- Comme on l'a vu, si la classe possède un attribut *att* et une méthode *meth*, on crée une instance de cette classe par

```
a=MaClasse()
```

- On accède à l'attribut *att* de *a* par

```
a.att
```

- et à la méthode *meth* par

```
a.meth()
```

en complétant si besoin les parenthèses de *MaClasse* et *meth*, si un argument est nécessaire. Même vides, il **faut** saisir les parenthèses pour l'instanciation et pour l'appel à une méthode !

2ème exemple de classe

- Pour mieux organiser son code, on peut définir la classe *MyClass* dans un fichier auxiliaire (*MyClassFile.py*). Pour utiliser cette classe dans le fichier principal on l'importe par

```
import MyClassFile

o=MaClasse()
```

ou, si on souhaite importer uniquement la classe MaClasse

```
from MaClasseFile import MaClasse


o=MaClasse()
```

Attributs - bilan

- Pour une classe, il peut exister des **attributs de classe** qui sont communs à toutes les instances de la classe
- Exemple d'attribut de classe: la chaîne de caractères *MaClasse.s* de la classe *MaClasse*
- Pour une classe, il peut exister des **attributs d'instance**.
- Exemple d'attribut d'instance: *r* et *i* sont les attributs de l'instance *a* de la classe *MyComplex*

Attributs privés

- On peut définir des attributs privés d'une classe. Les valeurs de ces attributs ne sont pas alors accessibles directement
- Exemple dans la classe *MyComplex2*

```
class MyComplex2:
    """ Classe python pour représenter un nombre
        complexe avec attributs privés """
    def __init__(self, reel=0, imag=0):
        self.__r = reel
        self.__i = imag
```

```
>>> b = MyComplex2(5, 7); print(b.__r, b.__i)
AttributeError: 'MyComplex2' object has no attribute '__r'
```

Attributs privés

Les méthodes de la classe peuvent accéder aux attributs privés.

- On peut définir des accesseurs (getters) et mutateurs (setters)

```
class MyComplex2:
    """ Classe python pour représenter un nombre
        complexe avec attributs privés"""
    def __init__(self, reel=0, imag=0):
        self.__r = reel
        self.__i = imag
    def get_imag(self):
        return self.__i
    def get_real(self):
        return self.__r
    def set_imag(self, i):
        self.__i = i
    def set_real(self, r):
        self.__r = r
```

Attributs privés

```
>>> b = MyComplex2(5, 7); print(b.get_real(), b.get_imag())
5 7
>>> b.set_real(-2.1), b.set_imag(8);
>>> print(b.get_real(), b.get_imag())
-2.1 8
```

Surcharge des opérateurs

- On peut choisir quel sens donner au symbole + *au sein d'une classe* grâce à la méthode *__add__*, $p1 + p2$ sera alors interprété comme $p1.\_\_add\_\_(p2)$.
- On peut donner n'importe quel sens à un symbole, on essaie en général de garder une cohérence par rapport à ses usages classiques.

De même pour de nombreux opérateurs. Quelques exemples :

Nom	Symbole	méthode
addition	+	*__add__*, *__radd__*
soustraction	−	*__sub__*, *__rsub__*
multiplication	*	*__mul__*, *__rmul__*
division	(/)	*__div__*
comparaison	==, <=, <, >=, >	*__eq__*, *__le__*, *__lt__*, *__ge__*, *__gt__*
format affichage	via *print*	*__str__*
instace exécutable	via ()	*__call__*

Classe *MyComplex* et opérateur d'addition (*__add__*)

```
class MyComplex:
...
  def __add__(self, c1):
    r0 = self.r + c1.r
    i0 = self.i + c1.i
    return MyComplex(r0, i0)
...
```

```
>>> a = MyComplex(2, 4)
>>> b = MyComplex(-1, 0.5)
>>> c = a + b  # Strictement équivalent à c = a.__add__(b);
>>> print(c.r, c.i) !
1 4.5
```

Remarque : en l'état, `a+c` renvoie une erreur si c n'est pas un *MyComplex* car *c.r* et *c.i* n'existent pas. Pour un float c, il faut faire $a + MyComplex(c, 0)$.

Classe *MyComplex* et opérateur de soustraction (*__sub__*)

```
class MyComplex:
...
  def __sub__(self, c1):
        r = self.r - c1.r
        i = self.i - c1.i
        return MyComplex(r, i)
...
```

```
>>> a = MyComplex(2, 4)
>>> b = MyComplex(-1, 0.5)
>>> c = a - b; print(c.r, c.i)
3 3.5
```

Classe *MyComplex* et opérateur de multiplication (*__mul__*)

```
class MyComplex:
...
    def __mul__(self, c1):
        r = self.r; i = self.i
        if (type(c1) == float) or (type(c1) ==
            int):
            r0 = c1 * r
            i0 = c1 * i
        else:
            r0 = r * c1.r - i * c1.i
            i0 = r * c1.i + i *  c1.r
        return MyComplex(r0, i0)
```

Cette fois on a défini $*$ pour une multiplication par un int ou float (on ferait de même pour + et −).

Classe *MyComplex* - Les opérateurs "à droite"

Dans l'exemple précédent, `z*a` est bien défini si *z* est *MyComplex* et *a* est un float/int.

```
>>> z = MyComplex(2, 4); z2 = MyComplex(-1, -2)
>>> z3 = z * z2; print(z3.r, z3.i) # 6 -8
>>> z4 = z2 * 5; print(z4.r, z4.i) # -5 -10 # bien défini ici
```

Classe *MyComplex* - Les opérateurs "à droite"

Dans l'exemple précédent, `z*a` est bien défini si z est *MyComplex* et a est un float/int.

```
>>> z = MyComplex(2, 4); z2 = MyComplex(-1, -2)
>>> z3 = z * z2; print(z3.r, z3.i) # 6 -8
>>> z4 = z2 * 5; print(z4.r, z4.i) # -5 -10 # bien défini ici
```

Mais ce n'est pas le cas de $a * z$ car les float/int ne savent pas comment interpréter la multiplication à droite par un *MyComplex* !

- Il existe l'opérateur __rmul__ (multiplication à droite) $\Rightarrow$ voir Notebook jupyter
- De même, il existe __radd__ et __rsub__ permettant de donner un sens à $a + z$ et $a - z$ (lorsque a n'est pas un *MyComplex*).

Classe *MyComplex* et opérateur de test d'égalité (*__eq__*)

```
class MyComplex:
...
    def __eq__(self, c1):
        eps = 1e-8 # pour erreur numerique
        return (abs(self.r - c1.r)< eps) \
            & (abs(self.i - c1.i)< eps)
...
```

```
>>> a = MyComplex(2, 4); b = MyComplex(-1, -2)
>>> c = a * b
>>> d = MyComplex(6.0, -8)
>>> c == d
True
```

Classe *MyComplex* et opérateur d'affichage (*__str__*)

Comment afficher une instance ? Selon l'usage, on peut vouloir n'afficher que certains attribut etc.
Par défaut, Python affiche simplement la classe de l'instance et son adresse :

```
>>> a = MyComplex(2, 4); print(a)
<__main__.MyComplex object at 0x00000257C3158D30>
```

Classe *MyComplex* et opérateur d'affichage (*__str__*)

```
class MyComplex:
...
  def __str__(self):
    return "Nombre complexe ({0:f}, {1:f})"
        .format(self.r, self.i)
...
```

```
>>> a = MyComplex(2, 4); print(a)
Nombre complexe (2.000000, 4.000000)

>>> b = MyComplex(-1, 0.5); print(b)
Nombre complexe (-1.000000, 0.500000)
```

Remarque : `__str__` s'applique également au résultat de la conversion `str(a)`

Classe *MyComplex* et instances exécutables (*__call__*)

Permet d'exécuter une méthode à partir de la syntaxe `objet(args)`, comme si l'instance de classe était une fonction.

```
class MyComplex:
...
    def __call__(self, theta=0,mar="*",col="b"):
        """ self(theta,mar,col) trace le complexe
            self après rotation d'angle theta
           dans le style voulu"""
        u=self.rotation2(theta)
        plt.plot([u.r],[u.i],marker=mar,color=col
            )
        return
```

Cette méthode est définie ici à titre d'exemple. Pour ce genre de tache il aurait probablement été plus naturel de définir ici méthode nommée *plot*, par exemple.

Classe *MyComplex* et attribut de classe

- Une méthode de classe s'exécute sur le nom de la classe si elle n'a pas d'argument (pas même *self*)
- Une attribut de classe est accessible depuis un objet ou depuis le nom de la classe

```
class MyComplex:
...
    goal = "To simulate complex numbers"
    def printClass():
        print("Inside MyComplex class")
...
```

Classe *MyComplex* et attribut de classe

```
>>> MyComplex.printClass()
Inside MyComplex class
>>> a = MyComplex(2,4)
>>> MyComplex.goal
To simulate complex numbers
>>> a.goal
To simulate complex numbers


# erreur si on exécute a.printClass()
```

vars

La commande *vars* permet de voir la liste des attributs d'une variable qui est l'instanciation d'une classe et leur contenu. Le retour est sous forme de dictionnaire.

```
>>> a = MyComplex(2,3)
>>> vars(a)
{'r': 2, 'i': 3}
```