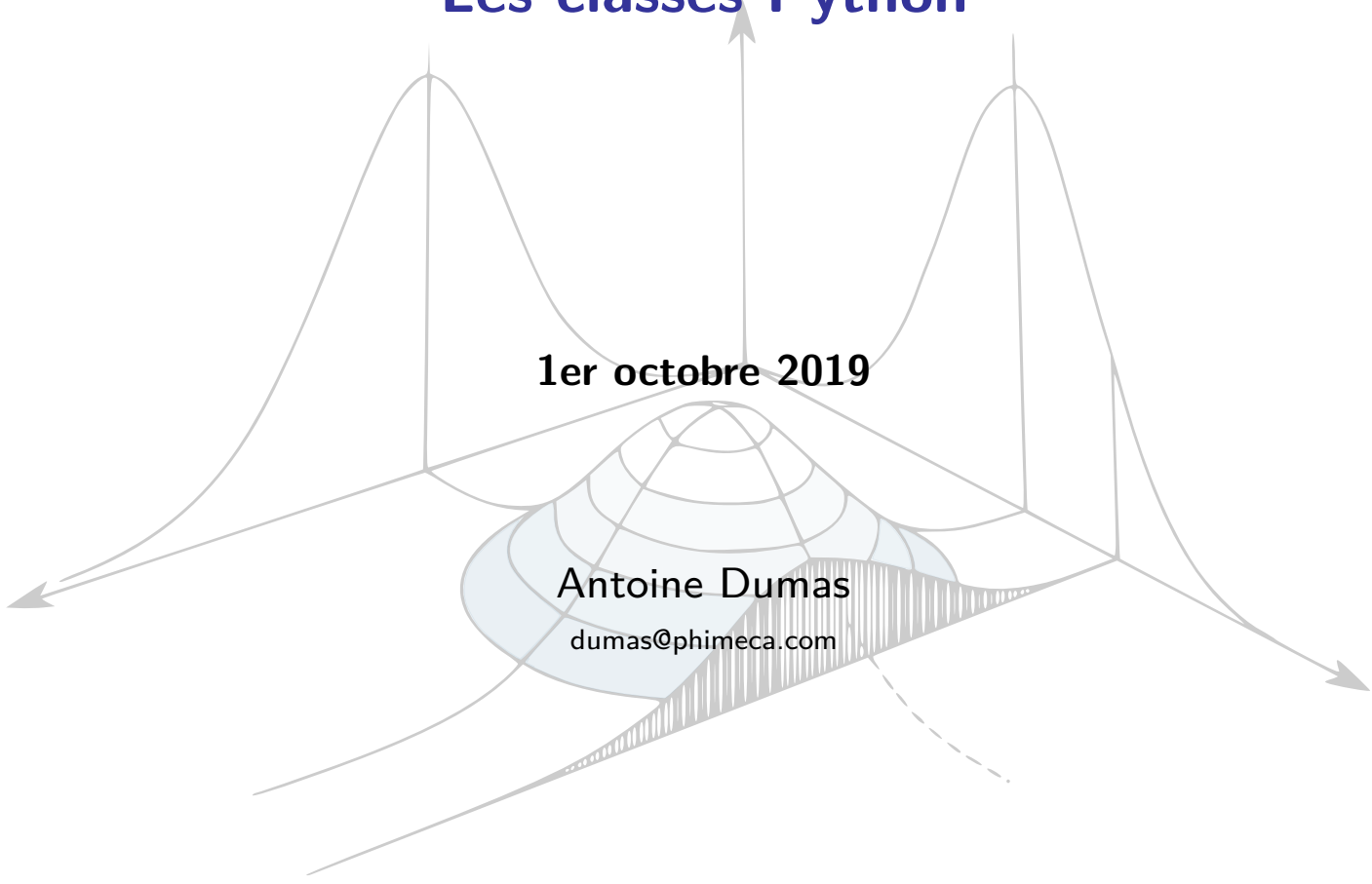
PHIMECA
Solutions for robust engineering

Les classes Python



1er octobre 2019

Antoine Dumas
dumas@phimeca.com

Sommaire

Sommaire

Pourquoi les classes ?

- Le langage python est orienté objet, tout est objet. Il est donc naturel de développer des classes permettant d'utiliser cette propriété : on fait alors de la POO.
- Cela permet de structurer le code et de le rendre beaucoup plus lisible lors du développement de gros projet. Il est également plus facile à maintenir dans le temps.
- Il est plus aisé de sauvegarder une instance d'une classe que plusieurs variables ou fonctions seules.
- La classe permet de différencier des paramètres "globaux" utilisés de façon récurrente au sein de celle-ci et des paramètres spécifiques liés à une méthode.
- L'implémentation des classes se rapproche du mécanisme en C++ : permet de définir des attributs, des méthodes, faire de l'héritage, surcharger des méthodes. Cependant en Python tout est public, et il n'y a pas besoin de déclarer le type des variables.

Sommaire

Implémenter une classe

La déclaration d'une classe a une forme similaire à la définition d'une fonction. Le mot-clé utilisé ici est :

- `class` : déclare une classe, par convention le nom est de type *CamelCase*.

```
# définit une classe vide
class MyClass:
    pass

>>> # on créer une instance de la classe MyClass
>>> instance = MyClass()
```

Implémenter une classe : méthode

Une classe peut contenir des méthodes qui sont accessibles partout à l'intérieur d'une classe et depuis une instance d'une classe. Par convention, les noms sont en minuscule séparés par des _. Pour les définir, il est nécessaire d'utiliser le mot-clé suivant :

- `self` : définit l'objet lui-même. Il est obligatoire de le mettre comme premier paramètre des méthodes de la classe. Cependant, lors de l'appelle à la méthode, il n'est pas considéré comme un argument de la méthode.

```
class MyClass:
    def my_method(self):
        print('Hello world !')

    def second_method(self):
        # appel à une méthode interne
        self.my_method()

>>> instance = MyClass()
>>> instance.my_method()
'Hello world !'
>>> instance.second_method()
'Hello world !'
```

La méthode `__init__` et les attributs

Lors de l'instanciation (création d'un objet à partir d'une classe), il est possible de définir un état initial, à l'image d'un constructeur en C++.

- `__init__` : lors de l'instanciation d'un objet, cette méthode est automatiquement appelée une seule fois. Il est possible de définir des attributs dans cette méthode, et même appeler des méthodes existantes.
- Les attributs peuvent être appelés et modifiés dans n'importe quelle méthode de la classe ainsi qu'à partir d'une instance de la classe. Par convention, il faut écrire `self.nom_de_mon_attribut`.
- Les attributs d'une classe sont contenus dans un dictionnaire accessible via la méthode `__dict__`.

```
class MyClass:
    def __init__(self):
        self.my_attribute = 3
        self.message = 'Hello world'

    def my_method(self):
        print(self.message)

>>> instance = MyClass()
>>> print(instance.__dict__)
{'message': 'Hello world', 'my_attribute':3}
```

```
>>> print(instance.my_attribute)
3
>>> instance.my_method()
'Hello world'
>>> # nouvelle valeur de l'attribut
>>> instance.message = 'New message'
>>> instance.my_method()
'New message'
```

Les arguments des méthodes

- Lors de la définition de la classe, des arguments peuvent être donnés, avec ou sans valeur par défaut. Il est souvent nécessaire de créer des attributs avec ces arguments pour y avoir accès à l'intérieur de la classe.
- Il en va de même pour les méthodes des classes.

```
class MyClass:
    def __init__(self, message):
        self.message = message

    def my_function(self, second_message):
        print(self.message)
        print(second_message)

>>> instance = MyClass('Hello World !')
>>> instance.my_function('Hello again !')
'Hello world !'
'Hello again !'
```

Les méthodes et attributs privés

- Par défaut dans Python, rien n'est privé mais par convention les attributs et méthodes privés commencent par _.
- Ils sont accessibles depuis une instance mais sa nature privée indique qu'ils ne devraient pas être modifiés ou appelés.

```
class ComputeSquare:
    def __init__(self):
        self._private_power = 2

    def _my_private_print(self, x, result):
        print('Square of {} = {}'.format(x, result))

    def eval(self, x):
        result =  x**self._private_power
        self._my_private_print(x, result)

>>> square = ComputeSquare()
>>> square.eval(3)
'Square of 3 = 9'
>>> # modification de l'attribut privé
>>> # qui modife la nature de la classe
>>> square._private_power = 3
>>> square.eval(3)
'Square of 3 = 27'
```

Les méthodes et attributs privés

- Des attributs demandant un traitement spéciale ou impactant d'autres attributs doivent être privés et être uniquement accessibles via des méthodes accesseurs.

```
class Data:
    def __init__(self, data_list):
        self.set_data(data_list) # définit self.dim et self._data_list

    def get_data(self):
        return self._data_list

    def set_data(self, new_data_list):
        if type(new_data_list) is not list:
            raise AttributeError('The given parameter is not a list.')
        else:
            self._data_list = new_data_list
            self.dim = len(new_data_list) # la dimension est mise à jour !
```

```
>>> the_data = [5, 6, 7]
>>> my_data = Data(the_data)
>>> print(my_data.dim)
3
>>> my_data.set_data(5) # mauvaise affectation
AttributeError: The given parameter is not a list.
```

```
>>> # bonne affectation
>>> my_data.set_data([9, 4])
>>> print(my_data.dim)
2
```

Les attributs de classes

- Les attributs peuvent être définis directement dans la classe. La différence est qu'il ne sont pas propres à l'objet créé. C'est notamment utile pour créer un compteur du nombre d'instances créées d'une même classe.
- De la même manière on peut créer des méthodes de classes (en utilisant `cls` et `classmethod`) et des méthodes statiques (avec `staticmethod`) mais leur utilisation est plus rare.

```
class MyClass:

    compteur = 0
    def __init__(self):
        MyClass.compteur += 1

>>> print(MyClass.compteur)
0
>>> instance1 = MyClass()
>>> print(MyClass.compteur)
1
>>> instance2 = MyClass()
>>> print(MyClass.compteur)
2
```

Exemple d'une mauvaise utilisation d'un attribut de classe

- Dans l'exemple ci-dessous, l'attribut de classe *tricks* est partagé par toutes les instances. Alors que l'attribut *tricks* est en réalité personnel au chien.
- Quelle est la bonne façon de faire ?

```
class Dog:

    tricks = []         # mauvaise utilisation de l'attribut de classe

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> print(d.tricks)           # partage non attendu par tous les chiens
['roll over', 'play dead']
```

Exemple d'une mauvaise utilisation d'un attribut de classe

- La bonne façon est d'utiliser un attribut d'instance à la place.

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # crée une nouvelle liste vide
                            # pour chaque chien
    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> print(d.tricks)
['roll over']
>>> print(e.tricks)
['play dead']
```

Documenter une classe

- Une bonne habitude est de commenter vos classes et méthodes avec des *docstring*.
- La docstring est accessible depuis l'instance avec la méthode spéciale `__doc__` ou en faisant appel à l'aide (*help*).
- Plusieurs conventions de format existent, notamment le format numpydoc. Des mots clés spécifiques sont utilisés (`Parameters`, `Return`, `Notes`, `Examples`, ...), ce qui permet de construire des documentations html automatiquement.

```
class MyClass:
    '''This is my first class.

    Parameters
    ----------
    message : str
        The message to display.
    '''
        def __init__(self, message):
            self.message = message
```

```
def my_function(self, second_message):
    '''Print 2 messages.

    Parameters
    ----------
    second_message : str
        The second message to display.
    '''
    print(self.message)
    print(second_message)
```

L'héritage

- L'héritage permet de créer des classes mères (ou de base) et où d'autres classes vont hériter de cette classe de base. L'héritage peut être simple (un seul héritage) ou multiple.
- Il est nécessaire d'initialiser la classe mère dans la classe fille.
- Les attributs et méthodes de la classe mère sont accessibles dans les classes enfants. La recherche des méthodes se fait dans l'ordre de la définition de la classe : d'abord dans la classe dont l'objet est directement issu puis récursivement dans les classes mères de gauche à droite.
- Les classes enfants peuvent surcharger les méthodes dans leur propre classe afin de les modifier.

L'héritage, exemple 1/3

- La classe `HumanBeing` hérite de `LivingLife`, elle hérite donc de la méthode `presentation` et de l'attribut `age`.
- Cependant dans la classe `HumanBeing`, l'attribut `age` est la valeur par défaut qui est 0.

```
class LivingLife:
    def __init__(self, age=0):
        self.age = age
    def presentation(self):
        print('I am a Living Life and I am {}.'.format(self.age))

class HumanBeing(LivingLife):
    pass # do nothing
```

```
>>> living_body = LivingLife(24)
>>> living_body.presentation()
'I am a Living Life and I am 24.'
>>> human = HumanBeing()
>>> human.presentation()
'I am a Living Life and I am 0.'
```

L'héritage, exemple 2/3

- Maintenant, la classe `HumanBeing` possède un paramètre d'entrée qui est un attribut age qui sera utilisé pour définir l'attribut de la classe mère.
- Le nom de l'attribut peut être différent mais s'il correspond exactement au même paramètre, il est conseillé de conserver le même nom pour plus de clarté.

```
class LivingLife:
    def __init__(self, age=0):
        self.age = age
    def presentation(self):
        print('I am a Living Life and I am {}.'.format(self.age))

class HumanBeing(LivingLife):
    def __init__(self, age):
        # initialisation de la classe mère
        LivingLife.__init__(self, age)

>>> living_body = LivingLife(24)
>>> living_body.presentation()
'I am a Living Life and I am 24.'
>>> human = HumanBeing(34)
>>> human.presentation()
'I am a Living Life and I am 34.'
```

L'héritage, exemple 3/3

- La classe HumanBeing a maintenant deux attributs et surcharge la méthode presentation.

```
class LivingLife():
    def __init__(self, age=0):
        self.age = age
    def presentation(self):
        print('I am a Living Life and I am {}.'.format(self.age))

class HumanBeing(LivingLife):
    def __init__(self, age, name):
        # initialisation de la classe mère
        LivingLife.__init__(self, age)
        self.name = name

    # surchage de presentation
    def presentation(self):
        print('I am a Human Being named {} and I am {}.'.
              format(self.name, self.age))

>>> human = HumanBeing(34, 'Paul')
>>> human.presentation()
'I am a Human Being named Paul and I am 34.'
```

Créer ma propre exception

- Créer votre propre erreur vous permet de les inclure dans des structures `try ... catch` en laissant libre les potentiels erreurs qui pourraient survenir pour d'autres causes.

```
class NumberOneError(Exception):
    def __init__(self, message):
        super(NumberOneError, self).__init__(message)

def test_erreur(value):
    try:
        result = 1. / value
        if result == 1.: # je ne veux pas que le résultat soit 1 !
            raise NumberOneError('Le résultat ne peut pas être 1.')
        print('Le résultat est {}.'.format(result))
    except NumberOneError as e
        print(e)
        # faire autre chose dans ce cas spécifique

>>> test_erreur(5)
'Le résultat est 0.2.'
>>> test_erreur(1) # l'erreur est bien récupérée.
'Le résultat ne peut pas être 1.'
>>> test_erreur(0) # le code retourne une erreur non prévue
ZeroDivisionError: float division by zero
```

Hériter de la classe `object` pour aller plus loin.

- L'héritage depuis la classe `object` est nécessaire en Python 2.x mais automatique en Python 3.x.
- Cela permet d'avoir accès à un grand nombre de méthodes spéciales (`__methodespeciale__`). Elles peuvent (ou doivent) être surchagées pour indiquer ce que Python doit faire dans différentes circonstances. Une liste exhaustive de ces méthodes est ici `https://rszalski.github.io/magicmethods/`
  - Construire et détruire une instance : `__new__`, `__init__`, `__del__`
  - Accéder au docstring : `__doc__`
  - Afficher des informations : `__repr__`, `__str__` avec `print`
  - Modifier ou effacer un attribut : `__setattr__` et `__delattr__`
  - Définir un itérateur (pour les boucles `for`) : `__iter__`
  - Additionner 2 objets (et soustraire ...) : `__add__`, ...
  - Comparer 2 objets : `__eq__`, ...
  - Définir les entrées et sorties du contexte manager `with:` : `__enter__`, `__exit__` (par exemple voir la classe otwrapy.TempWorkDir)
  - Sauvegarder et charger une instance avec pickle : `__getstate__`, `__setstate__`. (par exemple voir le code ReliaPipe)
  - ...

Hériter de la classe `object` pour aller plus loin.

Exemple avec les méthodes `__repr__` et `__str__`

```
class MyClass:
    def __init__(self):
        self.my_attribute = 3

    def __repr__(self):
        return "Fonction __repr__ de MyClass."

    def __str__(self):
        return "Fonction __str__ de MyClass."

>>> instance = MyClass()
>>> instance
"Fonction __repr__ de MyClass."
>>> print(instance)
"Fonction __str__ de MyClass."
```

Hériter de la classe `object` pour aller plus loin.

Exemple avec la méthode `__setattr__`. Cette technique est par exemple utilisée dans le code de ReliaPipe.

```
class MyClass:
    def __init__(self):
        self.my_attribute = 3
        self.something_has_changed = False

    def __setattr__(self, name, value):
        if name != "something_has_changed":
            self.__dict__["something_has_changed"] = True
            print("Modifying {} to {}".format(name, value))

        self.__dict__[name] = value
```

```
>>> instance = MyClass()
'Modifying my_attribute to 3'
>>> print(instance.something_has_changed)
False
>>> instance.my_attribute = 5
'Modifying my_attribute to 5'
>>> print(instance.something_has_changed)
True
```

Sommaire

Conclusion

- Les classes vous permettront de structurer le code de façon claire (à la fois pour vous et pour les autres):
  - N'hésitez pas à créer une classe par idée et d'avoir un fichier par classe.
  - Lorsqu'un code est utilisé plusieurs fois (même avec des changements mineurs), cela indique que vous pouvez probablement utiliser une méthode prenant un ou plusieurs arguments en entrée.
  - Dans la classe, découpez le code en plusieurs méthodes pour le rendre lisible. Les méthodes utiles uniquement au sein de la classe doivent être privée.
  - De même les attributs d'une classe nécessitant un traitement spéciale ou impactant d'autres attributs doivent être privés. Ils faut alors créer une méthode accesseur contenant le code spécifique pour ce traitement.
  - Respectez au maximum les conventions d'écritures des attributs, fonctions et méthodes !!
- Le développement d'une ou plusieurs classes est un premier pas vers la création de modules Python.
  - créer un package installable dans votre Python
  - créer une documentation html
  - créer des tests unitaires
  - valider votre code via l'intégration continue

Références

- Documentation Python officiel : `https://docs.python.org/fr/3.6/tutorial/classes.html`
- Le site open class room très complet : `https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/232721-apprehendez-les-classes`
- Un site avec de nombreux exemples avancées : `http://www.xavierdupre.fr/app/teachpyx/helpsphinx/c_classes/classes.html`
- Les méthodes spéciales : `https://rszalski.github.io/magicmethods/`

Sommaire

Travaux pratiques

- Créer une classe qui construit une suite arithmétique : $u_{n+1} = u_n + r$. Les paramètres de cette suite sont : l'état initial u_0, la raison r et le nombre d'éléments à calculer n. Créer la classe avec les éléments suivants :
  1. une fonction `__init__` qui prend comme argument la raison : ce sera un attribut de la classe.
  2. une méthode `build` qui construit la suite et qui prend comme argument l'état initial u_0 et le nombre n d'éléments à construire. Cette méthode doit stocker les éléments dans une liste qui sera un attribut de la classe.
  3. une méthode `get_values` qui est un accesseur à la liste créée avec la méthode `build`. Attention à la gestion de l'attribut de résultat !
- Créer une classe qui regroupe la fonction de calcul de la moyenne glissante et l'affichage.
- Construire la classe `DescenteGradient` qui regroupe l'ensemble des méthodes et paramètres dans la classe.
- Bonus : créer un paquet python !