Python for Scientists
Part 7-2 – Deeper into classes

Cyril Desjouy

June, 2016
Updated : September 20, 2019

Back To Basics

Photography by Sprout

Syntax : class

```
class Guitar:                           # A class and ...
    """ My Great Guitar Class """       # ... its DocString

    def __init__(self):                 # Class initialization
        self.brand = 'Schecter'         # An instance attribute...
        self.color = 'Cherry'           # ...a second...
        self.owner = 'Me'               # ... and a third

    def description(self):              # A method and ...
        """ My Great method """         # ... its DocString

        print('A {} {}'.format(self.color, self.brand)
```

- A `class` is like an **object** factory ! It provides the plans to build **objects**
- **Objects** created with a class are called **instances** of this class
- **Instances** may be initialized using the special method `__init__()`
- The `__init__()` method sets the initial value of **instance attributes**
- **Instance attributes** are defined using the object `self`
- **Methods** are like functions but declared in a class with `self` as 1st argument
- Classes are named using the *CapWords* convention

Syntax : class

```
class Guitar:                           # A class and ...
    """ My Great Guitar Class """       # ... its DocString

    def __init__(self):                 # Class initialization
        self.brand = 'Schecter'         # An instance attribute...
        self.color = 'Cherry'           # ...a second...
        self.owner = 'Me'               # ... and a third

    def description(self):              # A method and ...
        """ My Great method """         # ... its DocString

        print('A {} {}'.format(self.color, self.brand))
```

How to instantiate a class ?

```
» gtr = Guitar()       # Create a new Guitar object...
» gtr.brand            # ... called instance of the class Guitar
Schecter               # ... inheriting all attributes
» gtr.description()    # ... and methods of its class !
A Cherry Schecter
```

What is self exactly ?

```python
class WhatIsSelf:

    def __init__(self):
        self.identity = id(self)    # identity of the object self
```

```
» instance = WhatIsSelf()
» instance.identity == id(instance)
True
```

Actually, the object self is the instance of the class

```
class ClassPerson:

    name = 'Jo'
```

```
» type(ClassPerson)
type
» cp = ClassPerson()
» cp.name
Jo
```

```
class ClassPerson:

    name = 'Jo'
```

```
» TypePerson = type('Person',
                    (),
                    {'name':'Jo'})
```

```
» type(ClassPerson)
type
» cp = ClassPerson()
» cp.name
Jo
```

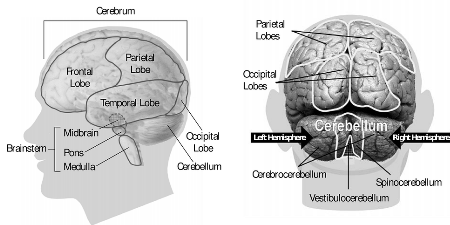
```
» type(TypePerson)
type
» tp = TypePerson()
» tp.name
Jo
```

Function `type` takes two different forms :

- `type(object)` : returns the type of the given object
- `type(name, bases, dict)` : returns a new type object (a class !) with :
  - `name` - Class name (`__name__` attribute)
  - `bases` - Tuple containing the base classes (`__bases__` attribute)
  - `dict` - Dictionary which is the namespace containing definitions for class body (`__dict__ attribute`)

Defining a new class is like defining a new type of object!

Instance & class attributes



From pixabay

Illustration : Instance attribute

```
class Guitar:

    def __init__(self, brand):
        self.brand = brand          # An instance attribute...
        self.strings = 6            # ... and another
```

. . . from an instance

```
» instance = Guitar('Fender')
» instance.brand
Fender
» instance.strings
6
» instance.strings = 7
» instance.strings
7
```

. . . from the class itself

```
» Guitar.brand
AttributeError: type object
    'Guitar' has no attribute
    'brand'
» Guitar.strings
AttributeError: type object
    'Guitar' has no attribute
    'strings'
```

Every instance has its own copy of every instance attribute

Illustration : Class attribute

```
class Guitar:

    strings = 6                # A class attribute...

    def __init__(self, brand):
        self.brand = brand     # ... and an instance attribute
```

...from an instance

```
» instance = Guitar('Fender')
» instance.brand
Fender
» instance.strings
6
» instance.strings = 7
» instance.strings
7
```

...from the class itself

```
» Guitar.brand
AttributeError: ...
» Guitar.strings
6
» Guitar.strings = 7
» Guitar.strings
7
» Guitar('Gibson').strings
7
```

Class attributes belong to the class itself. They are shared by all instances

CA with immutable type

```
class A:
    ca = 1     # Declare a CA


» a, b = A(), A()
» a.ca = 2     # Create new IA a.ca !
» b.ca         # b.ca still refers CA
1              # and A.ca is still 1
```

IA with immutable type

```
class A:
    def __init__(self):
        self.ia = 1  # Declare an IA

» a, b = A(), A()
» a.ia = 2         # a.ia & b.ia are
» b.ia             # 2 different IA !
1
```

CA with mutable type

```
class A:
    ca = []        # Declare a CA


» a, b = A(), A()
» a.ca.append(5)  # Changed in place
» b.ca            # A.ca is now [5]
[5]
```

IA with mutable type

```
class A:
    def __init__(self):
        self.ia = [] # Declare an IA

» a, b = A(), A()
» a.ia.append(5)  # a.ia & b.ia are
» b.ia            # 2 different IA !
[]
```

The instance namespace takes supremacy over the class namespace

How to set class attribute from an instance ?

```
class A:
    foo = 1

» a, b = A(), A()
» a.__class__.foo = 2         # Equivalent to A.foo = 2
» b.foo
2
```

```
class A:
    foo = 1

    def set_foo(self, val):
        A.foo = val           # Equivalent to self.__class__.foo = val

» a, b = A(), A()
» a.set_foo(2)
» b.foo
2
```

Application : Keep track of instances of a class

```
class Guitar:

    lst = []

    def __init__(self, brand):
        self.brand = brand
        Guitar.lst.append(brand)
```

```
class Guitar:

    N = 0

    def __init__(self, brand):
        self.brand = brand
        Guitar.N += 1
```

```
» g1 = Guitar('Fender')
» g2 = Guitar('Gibson')
» Guitar.lst
['Fender', 'Gibson']
```

```
» g1 = Guitar('Fender')
» g2 = Guitar('Gibson')
» Guitar.N
2
```

- Access to instance attributes only from instances
- Access to class attributes
  - directly from the class (without any instantiation)
  - or from any instance of the class

List attributes of an instance/object

```
class Guitar:
    """ Great Guitar class """

    lst = []

    def __init__(self, brand):
        self.brand = brand
        Guitar.lst.append(brand)
```

```
» g1 = Guitar('Fender')
» g2 = Guitar('Gibson')
» Guitar.lst
['Fender', 'Gisbon']
```

vars() - Displays instance attributes in the form of a dictionary

```
» vars(Guitar)
mappingproxy({'__module__': '__main__',
            '__doc__': ' Great guitar class ',
            'lst': ['Fender', 'Gibson'],
            '__init__':<function __main__.Guitar.__init__(self, brand)>,
            '__dict__':<attribute '__dict__' of 'Guitar' objects>,
            '__weakref__': <attribute '__weakref__' of 'Guitar' objects>})
» vars(g1)
{'brand': 'Fender'}
```

List attributes of an instance/object

```python
class Guitar:
    """ Great Guitar class """

    lst = []

    def __init__(self, brand):
        self.brand = brand
        Guitar.lst.append(brand)
```

```python
» g1 = Guitar('Fender')
» g2 = Guitar('Gibson')
» Guitar.lst
['Fender', 'Gisbon']
```

dir() - Displays instance and class attributes in form of a list

```python
» dir(Guitar)
['__class__', '__delattr__', (...), 'lst']

» dir(g1)
['__class__', '__delattr__', (...), 'brand', 'lst']
```

JAVA-like code

```
class Guitar:

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

    def get_brand(self):
        return self.brand

    def set_brand(self, brand):
        self.brand = brand
```

```
» g = Guitar('Fender')
» g.set_brand('Gibson')
» print(g.get_brand())
'Gibson'
```

Mandatory Encapsulation :
Access to data with getters/setters

Python-like code

```
class Guitar:

    def __init__(self, brand):
        self.brand = brand
```

```
» g = Guitar('Fender')
» g.brand = 'Gibson'
» print(g.brand)
'Gibson'
```

No Encapsulation :
Direct access to data

How to control object properties in Python ?

Syntax of property

```
class Guitar():
    # __ makes attribute private
    def __init__(self, brand):
        self.__brand = brand

    @property
    def brand(self):
        return self.__brand

    @brand.setter
    def brand(self, brand):
        self.__brand = brand
```

```
» g = Guitar('Fender')
» print(g.brand)
'Fender'
» g.brand = 'Gibson'
» print(g.brand)
'Gibson'
```

The point is ...

```
class Guitar():

    def __init__(self, brand):
        self.__brand = brand.upper()

    @property
    def brand(self):
        return self.__brand + '(c)'

    @brand.setter
    def brand(self, brand):
        self.__brand = brand.upper()
```

```
» g = Guitar('Fender')
» print(g.brand)
'FENDER (c)'
» g.brand = 'Gibson'
» print(g.brand)
'GIBSON (c)'
```

Properties make methods behave like attributes, without ()

Method Bestiary

From pixabay

Illustration : Instance methods

```python
class Guitar:

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

    def get_brand(self):
        return self.brand
```

```
» instance = Guitar('Fender') # Instantiate Guitar class
» Guitar.get_brand            # Display method directly from the class
<function __main__.Guitar.get_brand(self)>
» instance.get_brand          # Display method from the instance
<bound method Guitar.get_brand of <__main__.Guitar object at 0x7fc88> >
```

- When not attached to an instance
  - Cannot be called : `Guitar.get_brand()` $\Rightarrow$ `TypeError`
- When attached to an instance
  - Can be called : `instance.get_brand()` $\Rightarrow$ `Fender`

Illustration : Static Method

```
class Character:

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

    @staticmethod                    # Declaration with a decorator...
    def scream(n=2):                 # ... and no self argument !
        print('A{}h'.format('a'*n))
```

```
» john = Character('John Doe')
» john.scream(10)          # Static method can be called from an instance
Aaaaaaaaaaah
» Character.scream(3)      # ... or the class itself
Aaaah
```

- **Static methods :**
  - don't use the object `self`
  - are bounded to the class, not to the instance
  - can't access or modify class state
  - are used to create utility functions that belong to a class

Illustration : Class Method

```
class Character:

    screaming = 'Aou{}ouah'

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

    @classmethod                    # Declaration with a decorator...
    def scream(cls, n=2):           # ... and cls argument !
        print(cls.screaming.format('a'*n))
```

```
» john = Character('John Doe')
» john.scream(10)          # Class method can be called from an instance
Aouaaaaaaaaaaouah
» Character.scream(3)      # ... or the class itself
Aouaaaouah
```

- **Class methods :**
  - don't use the object `self` but have the class itself (`cls`) as parameter
  - are bounded to the class, not to the instance
  - can access and modify class state
  - are often used like factory methods to create new instances

```
class Character:   # with : from random import randint, choice

    syllable = ['di', 'mor', 'fam', 'dar', 'kil', 'glar', 'tres', 'gis']

    def __init__(self, name=None, force=None, life=None):
        self.name = name
        self.force = force
        self.life =life

    def summary(self):
        print('{} : {}/{}'.format(self.name, self.force, self.life))

    @classmethod
    def random_character(cls):      # Construct an instance of Character !
        name = cls.randname(cls.syllable)
        return cls(name=name, force=randint(5, 20), life=randint(5, 20))

    @staticmethod
    def randname(lst, length=2):    # Utility to generate random name
        name = [random.choice(lst) for i in range(length)]
        return ''.join(name).capitalize()
```

```
» boss = Character.random_character()
» boss.summary()                          # returns -> 'Morglar : 12/18'
```

Special methods ? What is it ?

- Special methods are instance methods known by Python
- Special methods names are surrounded by a double underscore `__`
- Special methods are used to :
  - Manage classes
  - Represent classes
  - Control instance attributes
  - Control properties
  - Overload operators
  - Overload indexation
  - ...

We've already seen one special method : `__init__()`

Illustration

```
class Room:

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

    def __str__(self):                  # Called by print function
        return '{} at floor {}'.format(self.name.capitalize(), self.floor)

    def __contains__(self, person):  # Called by membership operator 'in'
        return person in self.persons

    def __lt__(self, other):            # Called by comparison operator '<'
        return self.floor < other.floor
```

```
» bathroom = Room('bathroom', 1, ['Batman', 'Superman'])
» bedroom = Room('bedroom', 2, [])
» print(bathroom)          # Eq. to bathroom.__str__()
Bathroom at floor 1
» 'Batman' in bathroom     # True, Eq. to bathroom.__contains__('Batman')
» bathroom < bedroom       # True, Eq. to bathroom.__lt__(bedroom)
```

Class management		
self = MyClass(args)	`__new__(cls, args)`	(Constructor)
self = MyClass(args)	`__init__(self, args)`	(Init. args)
del self	`__del__(self)`	(Destructor)
self(args)	`__call__(self, args)`	(Make self callable)
✗	`__slots__(self)`	(To save memory)

Class representation		
repr(self)	`__repr__(self)`	
str(self)/print(self)	`__str__(self)`	
unicode(self)	`__unicode__(self)`	(Python 2)
bytes(self)	`__bytes__(self)`	(Python 3)
format(self, spec)	`__format__(self, spec)`	
hash(self)	`__hash__(self)`	
dir(self)	`__dir__(self)`	(List attributes)

Reflection		
isinstance(inst, class)	`__instancecheck__(self, inst)`	
issubclass(sub, class)	`__subclasscheck__(self, sub)`	
issubclass(sub, class)	`__subclasshook__(self, sub)`	(For abstract class)

Context manager	
with self as x:	`__enter__(self)`
with self as x:	`__exit__(self, exc, val, trace)`

Attributes		
self.name	__getattribute__(self, name)	(Unconditionally)
self.name	__getattr__(self, name)	(If name doesn't exist)
self.name = val	__setattr__(self, name, val)	
del self.name	__delattr__(self, name)	

Properties	
self.property	__get__(self, property)
self.property = val	__set__(self, property, value)
del self.property	__delete__(self, property)

Sequences		
self[key]	__getitem__(self, key)	
self[key] = value	__setitem__(self, key, value)	
del self[key]	__delitem__(self, key)	
self[key]	__missing__(self, key)	(If key doesn't exist)
len(self)	__len__(self)	
value in self	__contains__(self, value)	

Iterators		
iter(self)	__iter__(self)	
next(self)	__next__(self)	
reversed(self)	__reversed__(self)	(Reversed iterator)

Binary operators *(i.e. between 2)*		
Operator	Method Reverse Method	
+	`__add__(self, other)`	`__radd__()`
-	`__sub__(self, other)`	`__rsub__()`
*	`__mul__(self, other)`	`__rmul__()`
//	`__floordiv__(self, other)`	`__rfloordiv__()`
/	`__div__(self, other)` (Python 2)	`__rdiv__()`
/	`__truediv__(self, other)` (Python 3)	`__rtruediv__()`
%	`__mod__(self, other)`	`__rmod__()`
`divmod()`	`__divmod__(self, other)`	`__rdivmod__()`
**	`__pow__(self, other[, modulo])`	`__rpow__()`
<<	`__lshift__(self, other)`	`__rlshift__()`
»	`__rshift__(self, other)`	`__rrshift__()`
&	`__and__(self, other)`	`__rand__()`
^	`__xor__(self, other)`	`__rxor__()`
|	`__or__(self, other)`	`__ror__()`

Assignment operators	
+=	`__iadd__(self, other)`
-=	`__isub__(self, other)`
*=	`__imul__(self, other)`
/=	`__idiv__(self, other)`
//=	`__ifloordiv__(self, other)`
%=	`__imod__(self, other)`
**=	`__ipow__(self, other[, modulo])`
<<=	`__ilshift__(self, other)`
»=	`__irshift__(self, other)`
&=	`__iand__(self, other)`
^=	`__ixor__(self, other)`
|=	`__ior__(self, other)`

Comparison operators		
all operators	`__cmp__(self, other)`	(Only in Python 2)
<	`__lt__(self, other)`	
<=	`__le__(self, other)`	
==	`__eq__(self, other)`	
!=	`__ne__(self, other)`	
>=	`__ge__(self, other)`	
>	`__gt__(self, other)`	

Unary operators and functions *(i.e. applied to 1)*		
-self	__neg__(self)	
+self	__pos__(self)	
abs(self)	__abs__(self)	
~self	__invert__(self)	
complex(self)	__complex__(self)	
int(self)	__int__(self)	
round(self, n)	__round__(self, [n])	
float(self)	__float__(self)	
oct(self)	__oct__(self)	
hex(self)	__hex__(self)	
bool(self)	__bool__(self)	(__nonzero__() in Python 2)
x[self]	__index__(self)	

Module copy	
copy.copy(self)	__copy__(self)
copy.deepcopy(self)	__deepcopy__(self)
Module pickle	
pickle.dump(file, self)	__getstate__(self)
pickle.dump(file, self)	__reduce__(self)
pickle.dump(file, self)	__reduce_ex__(self)
x = pickle.load(file)	__getnewargs__(self)
x = pickle.load(file)	__getinitargs__(self)
x = pickle.load(file)	__setstate__(self)
Module math	
math.floor(self)	__floor__(self)
math.ceil(self)	__ceil__(self)
math.trunc(self)	__trunc__(self)
Module sys	
sys.sizeof(self)	__sizeof__(self)

```python
class Summary:

    def meth1(*args, **kwargs):        # Note: no "self" argument
        return 'Can be called on class but not on instance'

    def meth2(self, *args, **kwargs): # Note: "self" argument present
        return 'Can be called on instance but not on class'

    @staticmethod
    def meth3(*args, **kwargs):        # Note: no "self" argument
        return 'Can be called on class or instance'

    @classmethod
    def meth4(cls, *args, **kwargs):  # Note: "cls" argument
        return 'Can be called on class or instance'
```

The concepts of Delegation

From pixabay

Definitions

Delegation[1] **:** Evaluating a member of one object (the receiver) in the context of another, original object (the sender). In Oriented-Object Programming (OOP), there are two main delegation mechanisms :

- **Inheritance :** Mechanism of basing an object upon another object retaining similar implementation
  - Inheritance is an implicit delegation method
  - Implicitly establishes a sub-object from an existing object
- **Composition :** Mechanism of holding a reference to an object in another object.
  - Composition is an explicit delegation method
  - Explicitly delegates some tasks to other specialized objects

Delegation is an efficient way to factorize code and to facilitate code reuse

[1] From wikipedia.

```python
class Box:

    def __init__(self):
        self.status = 'closed'

    def open(self):
        self.status = 'open'
        print('Box open')

    def close(self):
        self.status = 'closed'
        print('Box closed')
```

```python
class LockedBox(Box):
    pass
```

```
» pandora = LockedBox()      # Instance of LockedBox
» pandora.status             # that inherits instance attribute status
closed
» pandora.open()             # and instance method open
Box open
» pandora.status
open
```

```python
class Box:

    def __init__(self):
        self.status = 'closed'

    def open(self):
        self.status = 'open'
        print('Box open')

    def close(self):
        self.status = 'closed'
        print('Box closed')
```

```python
class LockedBox(Box):

    pin = '0000'
    color = 'black'

    def open(self, pin):
        if LockedBox.pin == pin:
            print('Unlocked')
            self.status = 'open'
            print('Box open')
        else:
            print('Bad pin')
```

```
» pandora = LockedBox()     # Instance of LockedBox
» pandora.status            # that inherits instance attribute status
closed
» pandora.open('0000')      # and instance method open
Unlocked
Box open
» pandora.status
open
```

```python
class Box:

    def __init__(self):
        self.status = 'closed'

    def open(self):
        self.status = 'open'
        print('Box open')

    def close(self):
        self.status = 'closed'
        print('Box closed')
```

```python
class LockedBox(Box):

    pin = '0000'
    color = 'black'

    def open(self, pin):
        if LockedBox.pin == pin:
            print('Unlocked')
            super().open()

        else:
            print('Bad pin')
```

The `super()` function

- Syntax : `super().method(args)`[2]
- Provides access to inherited methods that have been override
- Often used on `__init__()` to recover instance attributes of the parent class
- In the case of multiple inheritance, search order determined by the `MRO`

[2] In Python 2, the syntax is `super(subClass, instance).method(args)`.

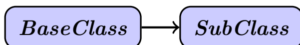
Multiple Resolution Order (MRO) : Order in which methods should be inherited.
The MRO can be displayed by using `__mro__` attribute or `mro` method

```
class SuperClass:                  # Equivalent to class SuperClass(object)

    def __init__(self):
        self.name = 'SuperClass' # Also named BaseClass or ParentClass

    def hello(self):
        print('Hello from the {}'.format(self.name))

class SubClass(SuperClass):

    def __init__(self):
        self.name = "SubClass"   # Also named DerivedClass or ChildClass
```

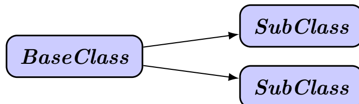
```
» SubClass.mro()
[__main__.SubClass, __main__.SuperClass, object]
» sc = SubClass()
» sc.name
'SubClass'
» sc.hello()
Hello from the SubClass
```

In OOP, there are 5 types of inheritance

- **Single inheritance :** A single class inherits from another.

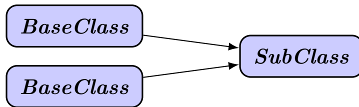


- **Hierarchical inheritance :** More than one class inherits from a class



- **Multilevel inheritance :** One class inherits from another, which in turn inherits from another...

- **Multiple inheritance :** One class inherits from multiple classes

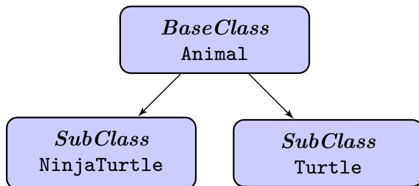


- **Hybrid inheritance :** A combination of at least two kinds of inheritance

Hierarchical Inheritance

More than one class inherits from a class

```python
class Animal:
    legs = 4

    @classmethod
    def count_legs(cls):
        print(cls.legs, 'legs')

class Turtle(Animal):
    attribute = 'shell'

class NinjaTurtle(Animal):
    legs = 2
```

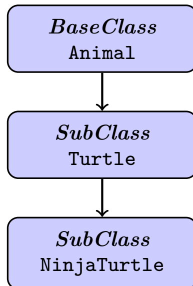



```
» Animal.count_legs
4 legs
» Turtle.count_legs
4 legs
» NinjaTurtle.count_legs()
2 legs
```

Multilevel Inheritance

One class inherits from another, which in turn inherits from another

```
class Animal:
    legs = 4

    @classmethod
    def count_legs(cls):
        print(cls.legs, 'legs')

class Turtle(Animal):
    attribute = 'shell'

class NinjaTurtle(Turtle):
    legs = 2
```

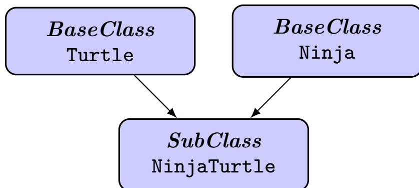



```
» NinjaTurtle.mro()
[__main__.NinjaTurtle, __main__.Turtle, __main__.Animal,  object]
» NinjaTurtle.attribute
'shell'
» NinjaTurtle.count_legs()
2 legs
```

Multiple Inheritance

One class inherits from multiple classes

```python
class Ninja:

    legs = 2
    battlecry = 'Yaka'

    @classmethod
    def attack(cls):
        print(cls.battlecry)

class Turtle:
    legs = 4
    attribute = 'shell'

class NinjaTurtle(Ninja, Turtle):
    battlecry = 'Cowabunga'
```

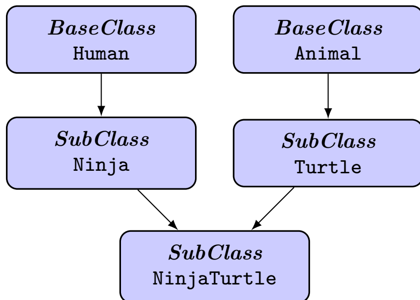



```
» NinjaTurtle.mro()
[__main__.NinjaTurtle, __main__.Ninja, __main__.Turtle, object]
» NinjaTurtle.legs
2
» NinjaTurtle.attack()
'Cowabunga'
```

Hybrid Inheritance

Combination of at least two kinds of inheritance

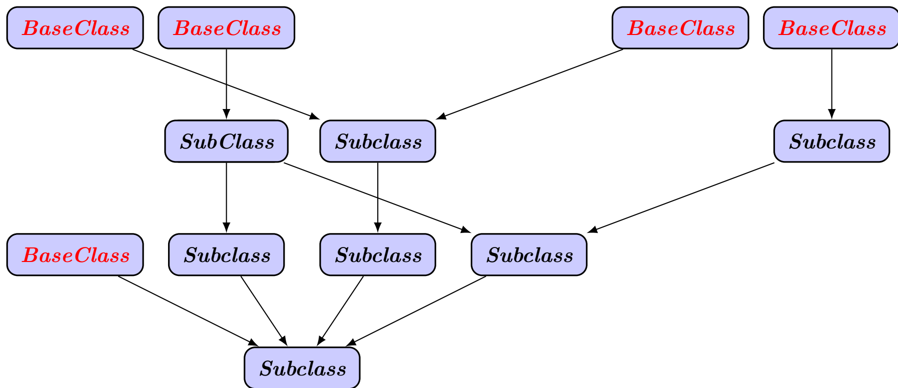
```
class Human:
    weapon = "intelligence"

class Animal:
    weapon = "instinct"

class Ninja(Human):
    weapon = "nunchaku"

class Turtle(Animal):
    armor = "shell"

class NinjaTurtle(Turtle, Ninja):
    pass
```




```
» NinjaTurtle.mro()
[__main__.NinjaTurtle,
 __main__.Turtle,
 __main__.Animal,
 __main__.Ninja,
 __main__.Human,
 object]
```

- `NinjaTurtle` class inherits attributes :
  - `armor = "shell"`
  - `weapon = "instinct"`
- Specified attribute/method searched in the current class, then into parent classes following the `MRO`.

Hybrid Inheritance
Combination of at least two kinds of inheritance



Multilevel inheritance

```
class Animal:
    def __init__(self):
        self.weapon = "instinct"

class Turtle(Animal):
    def __init__(self):
        super().__init__()
        self.armor = "shell"

class NinjaTurtle(Turtle):
    def __init__(self):
        super().__init__()
        self.battlecry = "cowabunga"
```

```
» Donatello = NinjaTurtle()
» vars(Donatello)
{'weapon':'instinct', 'armor':'shell',
    'battlecry':'cowabunga'}
```

`super().__init__()` can be used in all subclasses

Multiple inheritance

```
class Ninja:
    def __init__(self):
        self.weapon = "nunchaku"

class Turtle:
    def __init__(self):
        self.armor = "shell"

class NinjaTurtle(Ninja, Turtle):
    def __init__(self):
        Ninja.__init__(self)
        Turtle.__init__(self)
        self.battlecry = "cowabunga"
```

```
» Michelangelo = NinjaTurtle()
» vars(Michelangelo)
{'weapon':'nunchaku', 'armor':'shell',
    'battlecry':'cowabunga'}
```

`super().__init__()` can be used, but only provides access to `Turtle.__init__()` here

Composition

Holding a reference to an object in another object

```
class Character:

    def __init__(self, name, life=20, weapon=None):
        self.name = name      # instance attribute
        self.life = life      # instance attribute
        self.weapon = weapon  # instance attribute which is a class instance

    def attack(self, target):
        if self.weapon:
            self.weapon.attack(target)  # ... providing attack method

    def __repr__(self):
        if self.weapon:
            return '{} has {}'.format(self.name, self.weapon.name)
        else:
            return '{} is unarmed'.format(self.name)
```

- Definition of the class `Character` holding a reference to a `weapon` object
- Thus, `Character` class is *composed* of a `weapon` object and possibly other objects
- `Character` delegates `attack` method to `weapon` object

Composition

Holding a reference to an object in another object

```python
class Weapon:                               # Let's define the Weapon class...

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

    def attack(self, target):               # ... that provides an attack method
        target.life -= self.degat
        print('{} has {} life'.format(target.name, target.life))
```

```
» shredder = Character('Shredder', life=100)
» leonardo = Character('Leonardo', weapon=Weapon('Katanas', 10))
» leonardo
Leonardo has Katanas
» shredder
Shredder is unarmed
» leonardo.attack(shredder)
Shredder has 90 life
```

Which one to use ? *Both !*
Inheritance and composition allow both code factorisation and reuse but not the same way.
There are no rules that determine which one to use, but generally :

- If two objects have the same nature and one is a specialization of the other, such as
  - a Ninja Turtle is a specialisation of Turtle
  - a Bacon Burger is a specialisation of Hamburger

Go for Inheritance !

- If two objects are associated or exchange data, such as
  - a Ninja Turtle uses a weapon
  - a Fastfood restaurant serves Hamburgers

Go for Composition !

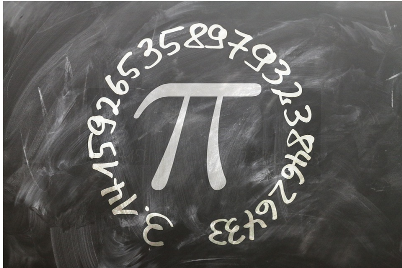
Most of the time a mix of inheritance and composition is the best way to go !

Mixing inheritance and composition

```
class NinjaTurtle(Character):

    subtype = 'Ninja Turle'

    def __init__(self, name, life=20, weapon=None):
        super().__init__(name, life, weapon)
        self.armor = 'shell'

    def double_strike(self, target):
        if self.weapon:
            self.weapon.attack(target)
            self.weapon.attack(target)
```

```
» donatello = NinjaTurtle('Donatello', weapon=Weapon('a Bow staff', 5))
» donatello
Donatello has a Bow staff
» donatello.subtype
'Ninja Turle'
» donatello.double_strike(leonardo)
Leonardo has 10 life
Leonardo has 5 life
```

Data Classes



From pixabay

What is a data class ?

- Data class is a feature coming with Python 3.7
- Data classes are classes containing mainly data
- There are no restrictions on the use of the features of regular classes in data classes
- Each dataclass implements the following special methods by default :
  - `__init__` with *kwargs* of the same names as those specified in the class
  - `__repr__` and `__str__` for representation
  - `__eq__` that compares all dataclass attributes in **order**

Illustration of dataclass

```
from dataclasses import dataclass

@dataclass
class Inventory:
    name: str
    quantity: str
```

```
» item = Inventory('Healing potion', 3)
» item
Inventory(name='Healing potion', quantity=3)
```

Our Example of Data Class...

```
from dataclasses import dataclass

@dataclass
class Inventory:
    name: str
    quantity: str
```

... and the form it would take with a regular class

```
class RegularInventory:

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

    def __repr__(self):
        return '{}(name={}, quantity={})'.format(self.__class__.__name__,
                                                  self.name,
                                                  self.quantity)

    def __eq__(self, other):
        return (self.name, self.quantity) == (other.name, other.quantity)
```

Description

- Default values can easily be set
- As for regular classes, *args* come first and *kwargs* at the end
- Each field is defined with a type hint :
  - If no type hint is provided, the field will not be a part of the dataclass
  - If you don't want to specify the type, use `typing.Any`

Illustration : Default values

```
from dataclasses import dataclass
from typing import Any

@dataclass
class Inventory:
    name: str
    quantity: int = 1
    misc: Any = None
```

```
» item = Inventory('Treasure Map')
» item
Inventory(name='Treasure Map', quantity=1, misc=None)
```

The `@dataclass` decorator can take the following arguments :

```
@dataclass (init=True, repr=True, eq=True,
            order=False, unsafe_hash=False,
            frozen=False)
```

Description of the arguments :

- `init` : if True, `__init__` method is generated.
- `repr` : if True, `__repr__` method is generated.
- `eq` : if True, `__eq__` method is generated.
- `order` : if True, `__lt__`, `__le__`, `__gt__`, and `__ge__` are generated.
  If order is true and eq is false, a ValueError is raised.
- `unsafe_hash` : if False, `__hash__` method is genetated *(advanced feature)*
- `frozen` : if True, fields are read-only

Description:

- The `__init__` method generated by the dataclass calls a `__post_init__` method
- The `__post_init__` method is useful to work on the fields declared in the dataclass

Illustration : `__post_init__`

```
@dataclass
class Inventory:
    name: str
    quantity: int

    def __post_init__(self):
        self.name = self.name.upper()
```

```
» item = Inventory('Healing potion', 3)
» item
Inventory(name='HEALING POTION', quantity=3)
```

Description:

- The `__init__` method generated by the dataclass calls a `__post_init__` method
- The `__post_init__` method is useful to work on the fields declared in the dataclass

Illustration : `__post_init__`

```
@dataclass
class Inventory:
    name: str
    quantity: int

    def __post_init__(self):
        self.name = self.name.upper()
```

```
» item = Inventory('Healing potion', 3)
» item
Inventory(name='HEALING POTION', quantity=3)
```

Summary of dataclass

- Data classes are a simple way to declare data structures in Python
- Aside from the basics presented here, data classes also provide advanced features
- See **realpython.com** for a very good guide to dataclass

Advanced concepts

Abstract Base Class & Metaclasses

From pixabay

Metaclasses

From pixabay

What is a Metaclass ?

Coming soon !

Abstract Base Classes

From pixabay

What is an Abstract Base Class ?

- An ABC is a class, just like any other class, but that is meant to be a base class
- An ABC contains one or more abstract methods that are methods without implementation
- Abstract methods have to be defined in subclasses of the ABC before instantiation

Declaration of an ABC

```
import abc

class SuperHero(abc.ABC):

    @abc.abstractmethod
    def super_power():
        """ To override """
```

```
» logan = SuperHero()
TypeError: Can't instantiate
    abstract class SuperHero with
    abstractmethods super_power
```

What is an Abstract Base Class ?

- An ABC is a class, just like any other class, but that is meant to be a base class
- An ABC contains one or more abstract methods that are methods without implementation
- Abstract methods have to be defined in subclasses of the ABC before instantiation

Declaration of an ABC

```
import abc

class SuperHero(abc.ABC):

    @abc.abstractmethod
    def super_power():
        """ To override """
```

... and a subclass

```
class Wolverine(SuperHero):

    def __init__(self):
        pass

    def super_power(self):
        print('Claws')
```

```
» logan = SuperHero()
TypeError: Can't instantiate
    abstract class SuperHero with
    abstractmethods super_power
```

```
» logan = Wolverine()
» logan.super_power()
Claws
```

Bibliography

From pixabay

Bibliography

- python.org
- thedigitalcatonline.com
- Sam & Max
- openclassroom.com
- programiz.com
- codefellows.github.io
- rszalski.github.io
- realpython.com
- zestedesavoir.com