Python OOP

<https://www.python-course.eu/python3_properties.php>

[https://www.learnbyexample.org/**python-properties**/](https://www.learnbyexample.org/python-properties/)

<https://pythonguide.readthedocs.io/en/latest/python/property.html>

**Properties in Python**

Some object-oriented languages such as Java and C# support private object attributes; which cannot be directly accessed from outside. Programmers often have to write getter and setter methods to access such private attributes.

**However in Python, all the attributes and methods are public, so it is useless to write getters or setters**.

**If you want to prevent direct access to an attribute, you should define it as a property**.

It is a simple way to customize access to an attribute.

Python programming provides us with a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

# Basic method of setting and getting attributes in Python  
class Celsius:  
 def \_\_init\_\_(self, temperature=0):  
 self.temperature = temperature  
  
 def to\_fahrenheit(self):  
 return (self.temperature \* 1.8) + 32  
  
  
# Create a new object  
human = Celsius()  
  
# Set the temperature  
human.temperature = 37  
  
# Get the temperature attribute  
print(human.temperature)  
  
# Get the to\_fahrenheit method  
print(human.to\_fahrenheit())

Whenever we assign or retrieve any object attribute like temperature as shown above, Python searches it in the object's built-in \_\_dict\_\_ dictionary attribute.

>>> human.\_\_dict\_\_

{'temperature': 37}

Therefore, man.temperature internally becomes man.\_\_dict\_\_['temperature'].

# Using @property decorator  
class Celsius:  
 def \_\_init\_\_(self, temperature=0):  
 self.temperature = temperature  
  
 def to\_fahrenheit(self):  
 return (self.temperature \* 1.8) + 32  
  
 @property  
 def temperature(self):  
 print("Getting value...")  
 return self.\_temperature  
  
 @temperature.setter  
 def temperature(self, value):  
 print("Setting value...")  
 if value < -273.15:  
 raise ValueError("Temperature below -273 is not possible")  
 self.\_temperature = value  
  
  
# create an object  
human = Celsius(37)  
  
print(human.temperature)  
  
print(human.to\_fahrenheit())  
  
coldest\_thing = Celsius(-300)

class Person(object):  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @property  
 def full\_name(self):  
 return self.first\_name + ' ' + self.last\_name  
  
 @full\_name.setter  
 def full\_name(self, value):  
 first\_name, last\_name = value.split(' ')  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @full\_name.deleter  
 def full\_name(self):  
 del self.first\_name  
 del self.last\_name  
This is Python's way of creating getters, setters, and deleters (or mutator methods) for a property in a class.

## When do you use **class method**?

**https://www.programiz.com/python-programming/methods/built-in/classmethod**

### 1. Factory methods

Factory methods are those methods that return a class object (like constructor) for different use cases.

It is similar to function overloading in C++. Since, Python doesn't have anything as such, class methods and static methods are used.

### Example 2: Create factory method using class method

from datetime import date  
  
# random Person  
class **Person**:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 @classmethod  
 def fromBirthYear(cls, name, birthYear):  
 return cls(name, date.today().year - birthYear)  
  
 def display(self):  
 print(self.name + "'s age is: " + str(self.age))  
  
person = Person('Adam', 19)  
person.display()  
  
person1 = **Person**.fromBirthYear('John', 1985)  
person1.display()

# Adam's age is: 19  
# John's age is: 31

Here, **we have two class instance creator**, a constructor and a fromBirthYear method.

The constructor takes normal parameters name and age. While, fromBirthYear takes class, name and birthYear, calculates the current age by subtracting it with the current year and returns the class instance.

The fromBirthYear method takes Person class (not Person object) as the first parameter cls and returns the constructor by calling cls(name, date.today().year - birthYear), which is equivalent to Person(name, date.today().year - birthYear)

Before the method, we see @classmethod. This is called a [decorator](https://www.programiz.com/python-programming/decorator) for converting fromBirthYear to a class method as classmethod().

class Pizza:

def \_\_init\_\_(self, ingredients):

self.ingredients = ingredients

def \_\_repr\_\_(self):

return f'Pizza({self.ingredients!r})'

**@classmethod**

def margherita(cls):

return **cls(['mozzarella', 'tomatoes'])**

@classmethod

def prosciutto(cls):

return cls(['mozzarella', 'tomatoes', 'ham'])

Note how I’m using the cls argument in the margherita and prosciutto factory methods instead of calling the Pizza constructor directly.

This is a trick you can use to follow the [Don’t Repeat Yourself (DRY)](https://en.wikipedia.org/wiki/Don't_repeat_yourself) principle. If we decide to rename this class at some point we won’t have to remember updating the constructor name in all of the classmethod factory functions.

Now, what can we do with these factory methods? Let’s try them out:

>>>

>>> Pizza.margherita()

Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()

Pizza(['mozzarella', 'tomatoes', 'ham'])

As you can see, we can use the factory functions to create new Pizza objects that are configured the way we want them. They all use the same \_\_init\_\_ constructor internally and simply provide a shortcut for remembering all of the various ingredients.

**Another way to look at this use of class methods is that they allow you to define alternative constructors for your classes**.

Python only allows one \_\_init\_\_ method per class. Using class methods it’s possible to add as many alternative constructors as necessary. This can make the interface for your classes self-documenting (to a certain degree) and simplify their usage.

class Student(object):  
  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
scott = Student('Scott', 'Robinson')

class Student(object):  
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @classmethod  
 def from\_string(cls, name\_str):  
 first\_name, last\_name = map(str, name\_str.split(' '))  
 student = cls(first\_name, last\_name)  
 return student  
  
scott = Student.from\_string('Scott Robinson')

class Student(object):  
  
 @classmethod  
 def from\_string(cls, name\_str):  
 first\_name, last\_name = map(str, name\_str.split(' '))  
 student = cls(first\_name, last\_name)  
 return student  
  
 @classmethod  
 def from\_json(cls, json\_obj):  
 # parse json...  
 return student  
  
 @classmethod  
 def from\_pickle(cls, pickle\_file):  
 # load pickle file...  
 return student

class **Student**(object):  
   
 def \_\_init\_\_(self, first\_name, last\_name):  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 @staticmethod  
 def is\_full\_name(name\_str):  
 names = name\_str.split(' ')  
 return len(names) > 1  
  
**Student**.is\_full\_name('Scott Robinson') # True  
**Student**.is\_full\_name('Scott')

## **What is a static method?**

Static methods, much like [class methods](https://www.programiz.com/python-programming/methods/built-in/classmethod), are methods that are bound to a class rather than its object.

## When do you use static methods?

### 1. **Grouping utility function to a class**

Static methods have a limited use case because, like class methods or any other methods within a class, they cannot access the properties of the class itself.

However, when you need a utility function that doesn't access any properties of a class but makes sense that it belongs to the class, we use static functions.

class **Dates**:  
 def \_\_init\_\_(self, date):  
 self.date = date  
  
 def getDate(self):  
 return self.date  
  
 **@staticmethod**  
 def **toDashDate**(date):  
 return date.replace("/", "-")  
  
  
date = Dates("15-12-2016")  
dateFromDB = "15/12/2016"  
dateWithDash = **Dates**.**toDashDate**(dateFromDB)  
  
if (date.getDate() == dateWithDash):  
 print("Equal")  
else:  
 print("Unequal")

**Output**

Equal

Here, we have a Dates class that only works with dates with dashes. However, in our previous database, all dates were present in slashes.

In order to convert the slash-dates to dash-dates, we have created a utility function toDashDate within Dates.

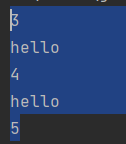
It is a static method because it doesn't need to access any properties of Dates itself and only requires the parameters.

We can also create toDashDate outside the class, but since it works only for dates, it's logical to keep it inside the Dates class.

* Global vs local



global a  
a = 3  
def Fuc():  
 global a  
 print(a)  
 a = a+1  
if \_\_name\_\_ == "\_\_main\_\_":  
 # global a  
 for i in range(2):  
 Fuc()  
 print('hello')  
 # global a  
 print(a)



NAME = ['ab','cd']  
def test1():  
 NAME = "erbi"  
 def test():  
 nonlocal NAME  
 NAME = "sb"  
 print(NAME)  
 test()  
 print(NAME)  
print(NAME)  
test1()  
print(NAME)  
  
# ['ab', 'cd']  
# sb  
# sb  
# ['ab', 'cd']

NAME = ['ab','cd']  
def test1():  
 NAME = "erbi"  
 def test():  
 nonlocal NAME  
 NAME = "sb"  
 print(NAME)  
 test()  
 print(NAME)  
print(NAME)  
test1()  
#   
# ['ab', 'cd']  
# sb  
# sb

