Python Programming Language

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo | Machine Learning Researcher

spring 2019


sw.eng.MohamedAyman@gmail.com


facebook.com/cs.MohamedAyman


linkedin.com/in/cs-MohamedAyman


codeforces.com/profile/Mohamed_Ayman

Lecture

Data Encapsulation

Lecture Agenda

We will discuss in this lecture the following topics

1- Setters & Getters
2- Static Variables
3- Static Methods
4- Private Variables
5- Private Methods
6- Built-In Class Attributes


Let's
START UP

Lecture Agenda

Section 1: Setters & Getters

Section 2: Static Variables

Section 3: Static Methods

Section 4: Private Variables

Section 5: Private Methods

Section 6: Built-In Class Attributes

Setters & Getters

- Data member: A class variable or instance variable that holds data associated with a class and its objects.
- In Python, getters and setters are not the same as those in other object-oriented programming languages. Basically, the main purpose of using getters and setters in object-oriented programs is to ensure data encapsulation.
- Private variables in python are not actually hidden fields like in other object oriented languages. Getters and Setters in python are often used when:
- We use getters & setters to add validation logic around getting and setting a value. To avoid direct access of a class field i.e. private variables cannot be accessed directly or modified by external user.
- To achieve getters & setters property, if we define normal get() and set() methods it will not reflect any special implementation.

Setters & Getters

- In this code functions

get_name(), get_age(), set_name() and set_age() acts as normal functions and doesn't play any impact as getters and setters, to achieve such functionality Python has a special function property().

```
class Person:
    def __init__(self, name = '', age = 0):
        self.name = name
        self.age = age

    def get_name(self):
        return self.name
    def set_name(self, name):
        self.name = name

    def get_age(self):
        return self.age
    def set_age(self, age):
        self.age = age

x = Person()
x.set_age(24)
print(x.get_age())
print(x.age)
```

```
24
24
```

Setters & Getters

- We use getters & setters to add validation logic around getting and setting a value. To avoid direct access of a class field i.e. private variables cannot be accessed directly or modified by external user.

```
class Person:
    def __init__(self, name = '', age = 0):
        self.name = name
        self.age = age
    def get_name(self):
        return self.name
    def get_age(self):
        return self.age
    def set_name(self, name):
        self.name = name
    def set_age(self, age):
        if age >= 0:
            self.age = age

x = Person('Muller', 28)
print(x.name, x.age)                    Muller 28
x.set_age(30)
print(x.name, x.age)                    Muller 30
x.set_age(-32)
print(x.name, x.age)                    Muller 30
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- **Section 2: Static Variables**
- Section 3: Static Methods
- Section 4: Private Variables
- Section 5: Private Methods
- Section 6: Built-In Class Attributes

Static Variables

- Class variable: A variable that is shared by all instance of a class. Class variables are defined within a class but outside any of the class's method and belongs only to the current instance of a class.
- Class variables or static variables are shared by all objects. Instance or non-static variables are different for different objects (every object has a copy of it).
- For example, let an Employee represented by class Employee. The class may have a static variable whose value is "raise-amount" for all objects. And class may also have non-static members like first-name and last-name.
- The Python approach is simple, it doesn't require a static keyword. All variables which are assigned a value in class declaration are class variables. And variables which are assigned values inside class methods are instance variables.

Static Variables

```
class Employee:
    raise_amount = 1.15

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

x = Employee('Jack', 'Bill')
y = Employee('Mark', 'Peter')

print(x.raise_amount, x.first_name, x.last_name, sep='\n')
print('-----')
print(y.raise_amount, y.first_name, y.last_name, sep='\n')
print('-----')
Employee.raise_amount = 1.17
print(x.raise_amount, x.first_name, x.last_name, sep='\n')
print('-----')
print(y.raise_amount, y.first_name, y.last_name, sep='\n')
print('-----')
```

```
1.15
Jack
Bill
-----
1.15
Mark
Peter
-----
1.17
Jack
Bill
-----
1.17
Mark
Peter
-----
```

Static Variables

```
class Employee:
    company_name = 'yahoo'

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def get_email(self):
        return  self.first_name + '_' + self.last_name + \
                '@' + Employee.company_name + '.com'

x = Employee('Jack', 'Bill')
y = Employee('Mark', 'Peter')

print(x.get_email())
print(y.get_email())
print('-----')
Employee.company_name = 'google'
print(x.get_email())
print(y.get_email())
print('-----')
```

```
Jack_Bill@yahoo.com
Mark_Peter@yahoo.com
-----

Jack_Bill@google.com
Mark_Peter@google.com
-----
```

Static Variables

- For example, let an Employee represented by class Employee. The class may have a static variable whose value is “raise-amount” for all objects. And class may also have non-static members like first-name and last-name. you can access the static variable from class name or any object of the class.

```
class Employee:
    raise_amount = 1.15

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

x = Employee('Jack', 'Bill')
y = Employee('Mark', 'Peter')

print(x.raise_amount, y.raise_amount)          1.15  1.15
x.raise_amount = 1.17
print(x.raise_amount, y.raise_amount)          1.17  1.17
y.raise_amount = 1.19
print(x.raise_amount, y.raise_amount)          1.19  1.19
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- ✓ Section 2: Static Variables
- **Section 3: Static Methods**
- Section 4: Private Variables
- Section 5: Private Methods
- Section 6: Built-In Class Attributes

Static Methods

- Static methods are methods that are related to a class in some way, but don't need to access any class-specific data. You don't have to use self, and you don't even need to instantiate an instance.
- The @staticmethod decorator was used to tell Python that this method is a static method.
- Static methods are great for utility functions, which perform a task in isolation. They don't need to (and cannot) access class data. They should be completely self-contained, and only work with data passed in as arguments. You may use a static method to add two numbers together, or print a given string.
- Static Methods: Cannot access anything else in the class. Totally self-contained code. Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters.

Static Methods

```
class Employee:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @staticmethod
    def split_name(full_name):
        first, last = full_name.split()
        return first, last

x, y = Employee.split_name('Jack Bill')
print(x)
print(y)
z = Employee(x, y)
```

```
Jack
Bill
```

Static Methods

```
class Employee:
    company_name = 'amazon'
    @staticmethod
    def generate_email(first_name, last_name):
        return  first_name + '_' + last_name + '@' + \
                Employee.company_name + '.com'
    @staticmethod
    def split_name(full_name):
        return full_name.split()
    def __init__(self, full_name):
        self.first_name, self.last_name = \
            Employee.split_name(full_name)
        self.email = \
            Employee.generate_email(self.first_name, self.last_name)

x = Employee('Mark Mullar')
print(x.first_name)
print(x.last_name)
print(x.email)
```

```
Mark
Mullar
Mark_Mullar@amazon.com
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- ✓ Section 2: Static Variables
- ✓ Section 3: Static Methods
- **Section 4: Private Variables**
- Section 5: Private Methods
- Section 6: Built-In Class Attributes

Private Variables

- Python doesn't restrict us from accessing any variable or calling any member method in a python program. All python variables and methods are public by default in Python. So when we want to make any variable or method public, we just do nothing.
- When we declare our data member private we mean, that nobody should be able to access it from outside the class. Here Python supports a technique called name mangling. This feature turns every member name prefixed with at least two underscores. So to make our member private.
- In actual terms (practically), python doesn't have anything called private member variable in Python. However, adding two underlines(__) at the beginning makes a variable or a method private is the convention used by most python code.

Private Variables

```
class Employee:
    def __init__(self, first_name, last_name):
        self.__first_name = first_name
        self.__last_name = last_name
    def get_first_name(self):
        return self.__first_name
    def get_last_name(self):
        return self.__last_name

x = Employee('Jack', 'Bill')
print(x.get_first_name(), x.get_last_name())
print(x.__first_name)
```

```
Jack Bill
Traceback (most recent call last):
  File "main.py", line 16, in <module>
    print(x.__first_name)
AttributeError: 'Employee' object has
no attribute '__first_name'
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- ✓ Section 2: Static Variables
- ✓ Section 3: Static Methods
- ✓ Section 4: Private Variables
- **Section 5: Private Methods**
- Section 6: Built-In Class Attributes

Private Methods

- Methods can be made private in the same way, by naming them with two leading underscores and no trailing underscores. This is about syntax rather than a convention. double underscore will mangle the attribute names of a class to avoid conflicts of attribute names between classes. (so-called “mangling” that means that the compiler or interpreter modify the variables or function names with some rules, not use as it is)
- The mangling rule of Python is adding the “_ClassName” to front of attribute names are declared with double underscore. That is, if you write method named “__method” in a class, the name will be mangled in “_ClassName__method” form.
- Because of the attributes named with double underscore will be mangled like above, we can not access it with “ClassName.__method”. Sometimes, some people use it as like real private ones using these features, but it is not for private and not recommended for that. For more details, read Python Naming.

Private Methods

```
class Employee:
    def __init__(self):
        self.first_name = ''
        self.last_name = ''
    def __set_full_name(self, first, last):
        self.first_name = first
        self.last_name = last
    def set_full_name(self, full_name):
        first, last = full_name.split()
        self.__set_full_name(first, last)

x = Employee()
x.set_full_name('Jack Bill')
print(x.first_name, x.last_name)
print(x.__set_full_name('Jim', 'Mark'))
```

```
Jack Bill
Traceback (most recent call last):
  File "main.py", line 15, in <module>
    print(x.__set_full_name('Jim', 'Mark'))
AttributeError: 'Employee' object has
no attribute'__set_full_name'
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- ✓ Section 2: Static Variables
- ✓ Section 3: Static Methods
- ✓ Section 4: Private Variables
- ✓ Section 5: Private Methods
- **Section 6: Built-In Class Attributes**

Built-In Class Attributes

- We can access the built-in class attributes using the . operator. Following are the built-in class attributes.

Attribute	Description
`__dict__`	This is a dictionary holding the class namespace.
`__doc__`	This gives us the class documentation if documentation is present. `None` otherwise.
`__name__`	This gives us the class name.
`__module__`	This gives us the name of the module in which the class is defined. > In an interactive mode it will give us `__main__`.
`__bases__`	A possibly empty tuple containing the base classes in the order of their occurrence.

Built-In Class Attributes

```
class Employee:
    '''This is a sample class called Employee.'''
    def __init__(self, first_name = '', last_name = ''):
        self.first_name = first_name
        self.last_name = last_name
    def get_first_name(self):
        return self.first_name
    def set_first_name(self, first_name):
        self.first_name = first_name
    def get_last_name(self):
        return self.last_name
    def set_last_name(self, last_name):
        self.last_name = last_name
    def __del__(self):
        del self.first_name
        del self.last_name
```

Built-In Class Attributes

Example:

```
x = Employee('Peter', 'Jack')
print(x.__dict__)
print(x.__doc__)
print(Employee.__doc__)
print(Employee.__name__)
print(x.__module__)
print(Employee.__module__)
```

Output:

```
{'first_name': 'Peter', 'last_name': 'Jack'}
This is a sample class called Employee.
This is a sample class called Employee.
Employee
__main__
__main__
```

Built-In Class Attributes

- getattr() function is used to access the attribute value of an object and also give an option of executing the default value in case of unavailability of the key. This has greater application to check for available keys in web development and many other phases of day-to-day programming.
- setattr() is used to assign the object attribute its value. Apart from ways to assign values to class variables, through constructors and object functions, this method gives you an alternative way to assign value.
- hasattr() is an inbuilt utility function in Python which is used in many day-to-day programming applications. It's main task is to check if an object has the given named attribute and return true if present, else false.
- delattr() method is used to delete the named attribute from the object, with the prior permission of the object.

Built-In Class Attributes

Example:

```
x = Employee('Bill', 'Jack')
print(hasattr(x, 'first_name'))
print(hasattr(x, 'last_name'))
print(getattr(x, 'first_name'))
print(getattr(x, 'last_name'))
setattr(x, 'first_name', 'Mark')
setattr(x, 'last_name', 'Peter')
print(getattr(x, 'first_name'))
print(getattr(x, 'last_name'))
delattr(x, 'first_name')
delattr(x, 'last_name')
print(hasattr(x, 'first_name'))
print(hasattr(x, 'last_name'))
```

Output:

```
True
True
Bill
Jack


Mark
Peter


False
False
```

property() Method for Setters & Getters

- Using property() function to achieve getters and setters behavior:
- In Python property() is a built-in function that creates and returns a property object. A property object has three methods, getter(), setter(), and delete().
- property() function in Python has four arguments property(fget, fset, fdel, doc), fget is a function for retrieving an attribute value. fset is a function for setting an attribute value. fdel is a function for deleting an attribute value. doc creates a doc-string for attribute.
- A property object has three methods, getter(), setter(), and delete() to specify fget, fset and fdel individually.

property() Method for Setters & Getters

- In this code there is only one print statement at assignment line but output consists of three lines due to setter method set_age() called in assignment statement and getter method get_age() called in printing statement. Hence age is a property object that helps to keep the access of private variable safe.

```
class Person:
    def __init__(self, age = 0):
        self._age = age

    def get_age(self):
        print('Get Age Method')
        return self._age
    def set_age(self, age):
        print('Set Age Method')
        self._age = age
    def del_age(self):
        print('Del Age Method')
        del self._age

    age = property(get_age, set_age, del_age)

x = Person()
x.age = 24
print(x.age)
```

```
Set Age Method
Get Age Method
24
```

Lecture Agenda

- ✓ Section 1: Setters & Getters
- ✓ Section 2: Static Variables
- ✓ Section 3: Static Methods
- ✓ Section 4: Private Variables
- ✓ Section 5: Private Methods
- ✓ Section 6: Built-In Class Attributes


DO
MORE.