Object-Oriented Programming in Python vs. Java

The magic of Python's OOPs

Jordan Williams Following
Aug 24 · 10 min read ★

Photo by Bruce Warrington on Unsplash

Python's OOP (Object Oriented Programming) can be a little different than the others you might be used to. Coming from Java it might seem like the two barely correlate.

This article will look at Python's OOP from a Java perspective. With Java being one of the most popular object-oriented programming languages, it will make this applicable to all these Python examples.

Let's look at a standard Java class first with the following code:

```
public class Student {

    private String school = "Jordan High";

    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getSchool() {
        return school;
    }
    public void setSchool(String school) {
        this.school = school;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age){
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student [age=" + age + ", name=" + name + ", school=" + school + "]";
    }


}
```

Student.java hosted with ♡ by GitHub view raw

From the code above, you can see that this is your standard Java class. This is a student class. It has three attributes: `school` , `name` and `age` . These are all private attributes. As we should know, the purpose of private attributes is for encapsulation. This is the principle of only allowing the modification of an attribute through methods (normally getter and setters).

```
class Student:
    __school = "Jordan High"

    def __init__(self,name,age):
        self.__name = name
        self.__age = age

    def get_school(self):
        return self.__school

    def set_school(self,school):
        self.__school = school

    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

    def __repr__(self):
        return "Student [age=" + str(self.__age) + ", name=" + self.__name + ", school=" + sel
```

Student.py hosted with ♡ by GitHub view raw

Here comes Python.

I have created an equivalent version of the Java class in Python. You might notice some differences in the syntax, but it's all the same.

Constructor

Java

```java
public Student(String name, int age) {
    this.name = name;
    this.age = age;
}
```

For Java, you would use the name of the class to create your constructor `public Student(String name, int age)`. `this` refers to the current instance of the class. Hence using `this.<variable name>` refers to the attributes declared at the top of our Java code.

Python

```python
def __init__(self,name,age):
    self.__name = name
    self.__age = age
```

For Python, we use the `__init__` method for our constructors. You will later notice a trend with methods with a double underscore at both ends.

You will notice the `self` parameter placed at the beginning of the constructor. `self` tells Python that this is a method. It gives the rest of the method access to instance variables and methods of the class. Python automatically passes the first parameter as `self` when you call the method.

Inside the constructor, we assigned two instance variables. `__name` and `__age` . Note that we had to use `self` to specify that it is an instance variable and unlike Java where we do not have to have previously declared the variables because Python is a dynamically typed language.

Attributes

Java

```java
public class Student {

    private String school = "Jordan High";

    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

For Java, you place the specific type of access you want an attribute to have at the beginning of declaring the variable. There are three types of access modifiers: `public` , `private` and `protected` .

- public —Can be accessed from anywhere.

- private — Can only be accessed within the current class.
- protected — Can only be accessed by the current class and its subclasses.

In the above class all the attributes(`age` , `name` , `school`) are private.

Python

```python
class Student:
    __school = "Jordan High"

    def __init__(self,name,age):
        self.__name = name
        self.__age = age
```

For Python, there is no explicit way to enforce private or protected access modifiers. Everything will always be public. However, Python has general conventions, whenever seen you can instantly know the specific access modifier.

Python uses the underscore `_` to represent the different types of access. In the above code, the access is currently `private` because of the double underscore at the beginning.

- public — No underscore.
- protected — single underscore.
- private — double underscore.

Let's look at the below code that shows the different access types.

```python
class Player:

    # public
    health = None
    #protected
    _magic = None
    #private
    __name = None

    def __init__(self,name,health,magic):
        self.health = health
        self._magic = magic
        self.__name = name
```

For ease, I initialized these variables to `None`. This was completely optional because as I said previously, Python is a dynamic language and variables could have just been defined in the constructor.

Let’s instantiate our object and see these access conventions.

```python
>>> p1 = Player("Jordan",120,300)
>>> p1.health
120
>>> p1._magic
300
>>>
```


We see both `health` and `_magic` easily accessed by just calling their respective variables. `health` being public is normal for it to be accessed but `_magic` being protected normally should not be easily accessed because only the class or its subclasses should access it. The single underscore '_' tells whoever is trying to access that this should only be accessed by the class and its subclasses.

For private access (double underscore), Python handles it a little differently.

```
>>> p1.__name
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    p1.__name
AttributeError: 'Player' object has no attribute '__name'
>>>
```

private_name.py hosted with ♡ by GitHub view raw

Notice that when I attempt to access `__name` I get an error. This is because Python made an attempt to hide this variable from being accessed. However, as I stated before, all variables in Python are public and can be accessed.

Let's see why.

```
>>> p1._Player__name
'Jordan'
>>>
```

show_private.py hosted with ♡ by GitHub view raw

Here we can see we were able to access our value. But for some reason, the variable name changed. Well, this is Python's way of showing that this value should not be accessed outside of the class (*being private and all*). It does this by changing the name to `_<class name>__<variable name>` .

The same principle for assigning access to variables can also be done to methods as well.

String Representation

Java

```
public String toString() {
        return "Student [age=" + age + ", name=" +
name + ", school=" + school + "]";
    }
```

For Java, `toString` is inherited from the `Object` class that every Java class by default is inherited from. By default, the `toString` method returns the name of the object's class plus its hash code. This is often not very useful, therefore we use its override to produce more valuable output.

```
Student student = new Student("Jordan", 22);

System.out.println(student);
```

Output:

```
Student [age=22, name=Jordan, school=Jordan High]
```

Python

```
def __repr__(self):
        return "Student [age=" + str(self.__age) +
", name=" + self.__name + ", school=" +
str(self.__school) + "]"
```

To produce the same effect, Python uses `__repr__` (another method with a double underscore at both ends). `__repr__` is also inherited from the default class all Python objects inherit from.

```
>>> st = Student("Jordan",22)
>>> st
Student [age=22, name=Jordan, school=Jordan High]
>>>
```


Getter and Setter Functions

Although we created getters and setters, Python has its own Pythonic way of creating them.

The current implementation:


```
>>> a = Student("Jordan",22)
>>> st = Student("Jordan",22)
>>> st.get_age()
22
>>> st.set_age(30)
>>> st.get_age()
30
```

getter_setter.py hosted with ♡ by GitHub view raw

The Pythonic way:

```python
@property
def age(self):
    return self.__age

@age.setter
def age(self,age):
    self.__age = age
```

By adding the `@property` decorator to a method, it will be designated as a getter. Following that designation, you will use the name of that method(which became a decorator) to assign a corresponding setter. This is done by adding the `<name>.setter` .

To make our setter more meaningful, let’s add a constraint to it. Whenever the age is going to be set below 0, we will raise an exception.

```python
@property
def age(self):
    return self.__age

@age.setter
def age(self,age):
    if age < 0:
        raise Exception("Less than 0")
    self.__age = age
```

The code above now achieves this. Let’s try it out just to make sure:

```
>>> std = Student("Jordan",25)
>>> std.age
25
>>>
```


You now can now access it like a normal property. By calling the `age` property, we just created our objects by accessing the private variable `__age` .

Let’s try our setter next:

```
>>> std = Student("Jordan",25)
>>> std.age
25
>>> std = Student("Jordan",25)
>>> std.age = 30
>>> std.age
30
>>> std.age = -1
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    std.age = -1
  File "<pyshell#0>", line 31, in age
    raise Exception("Less than 0")
```

```
Exception: Less than 0
>>>
```

student_setter.py hosted with ♡ by GitHub view raw

We can set our `age` by directly assigning a variable to it similar to a normal variable where we can also keep its constraints as well.

Notice whenever we try to assign -1 to `age` the property it goes through is an exception.

Remember that we can still access our private variables `__age`, `__name` and `__school` as stated before by using `_<class name>__<variable name>`.

Keep in mind that the reason we have getters and setters in OOP (*Object Oriented Programming*) is encapsulation. We keep our variables private to ensure that the only way one can access and modify them is through getters and setters. With this, we can control how they are seen and modified.

Imagine you had a `Card` class. It would have a `cardnumber` property because card numbers are very sensitive data. Whenever we call our getter, it would always mask all but the last four (4) digits of the card number. We would only allow our setter to allow card numbers of size 16 digits because that is the normal length of a card number.

Now, let's turn all our getters to the Pythonic way:

```
class Student:
    __school = "Jordan High"

    def __init__(self,name,age):
        self.__name = name
        self.__age = age

    @property
    def school(self):
        return self.__school

    @school.setter
    def school(self,school):
        self.__school = school

    @property
    def name(self):
```

```
        return self.__name

    @name.setter
    def name(self,new_name):
        self.__name = new_name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self,age):
        if age < 0:
            raise Exception("Less than 0")
        self.__age = age

    def __repr__(self):
        return "Student [age=" + str(self.__age) + ", name=" + self.__name + ", school=" + self
```

Student.py hosted with ♡ by GitHub view raw

For consistency, let's modify the Java class with the changes made to the age setter:

```
public class Student {

    private String school = "Jordan High";

    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getSchool() {
        return school;
    }
    public void setSchool(String school) {
        this.school = school;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
```

```java
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) throws Exception {
        if (age < 0){
            throw new Exception("Less than 0");
        }
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student [age=" + age + ", name=" + name + ", school=" + school + "]";
    }


}
```

Student.java hosted with ♡ by GitHub view raw

Static Methods

Yes, static methods. As for static variables, Python does not support them (as far as I know).

Static methods are methods that can be called without instantiating a class. They are called directly from the class name itself. Static methods are often used for methods whose purpose does not require an instantiation like a square root function for example.

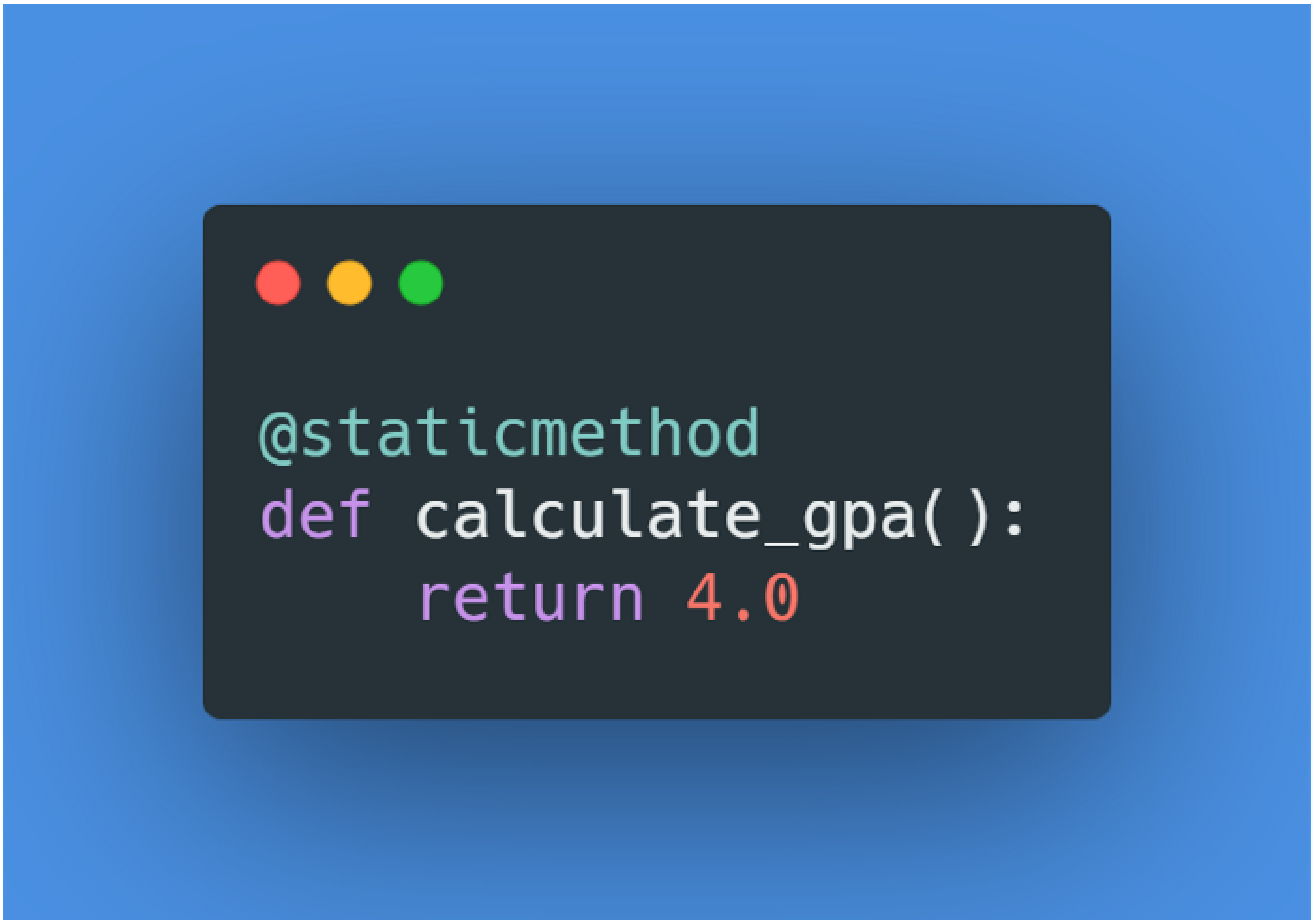
These methods are accessed by `<name of class>.<static method>` .

Java

```java
public static double calculateGPA(){
        return 4.0;
    }
```

For Java, the `static` modifier needs to be added to the method definition.

Python



For Python, the `staticmethod` decoration needs to be added to the method. A `self` parameter is not needed. This is because `self` ties the method to the current instance of the class and gives access to instance variables and methods. These are all things we do not need.

Reminder: Throughout this article, I have been using camel case(`calculateGPA`) for Java methods and variables and snake case(`calculate_gpa`) for Python functions and variables. These are the standard naming conventions for both languages.

Please click the link below to learn more about Python's naming convention through its documentation.

Python documentation on naming conventions.

Magic Methods

Python magic methods are special internal methods that abstract common Python operations. These methods happen behind the scenes.

```
>>> num1 = 10
>>> num2 = 5
>>> num1 + num2
15
>>>
```

normal_add.py hosted with ♡ by GitHub view raw

The above snippet shows the simple addition between two (2) numbers. This is a simple operation that internally is being modified by a magic method.

```
>>> num1 = 10
>>> num2 = 5
>>> num1.__add__(num2)
15
>>>
```

magic_methods_add.py hosted with ♡ by GitHub view raw

The snippet above shows what is really happening internally. Whenever an addition is performed Python uses the `__add__` method. Notice the double underscore score. These are called **Magic methods**. Alternatively, they are also called **Dunder** methods(**D**ouble **Under**score).

We have used quite a few of these magic methods throughout this article. `__init__` and `__repr__` are some of those methods. Notice that we never actually specifically called these methods yet they were used internally.

In going back to our addition example, `num1` and `num2` are both integers and both inherit from the `int` class.

Let’s look at the `int` class.

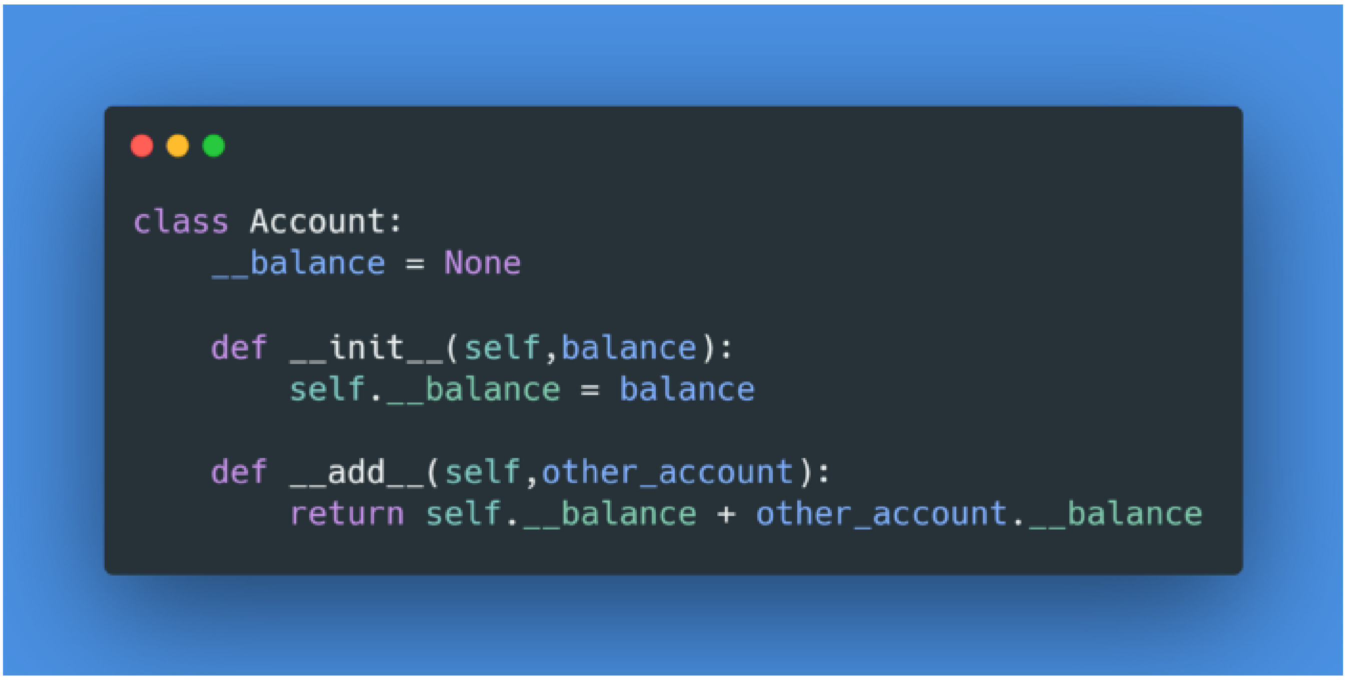
```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
```

```
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__',
'__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
'__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>>
```

int_class.py hosted with ♡ by GitHub view raw

The `dir` function shows all methods and variables for an object or class. You can see many other magic methods here as well. All these magic methods abstract some operations that are happening in the background. Every operation from multiplication, division, and powers is done by magic methods.

We can even add these methods to our custom classes.



This is an `Account` class that stores the balance of an account. We added the magic method `__add__` which will allow us to perform addition on the `Account` objects based on the balance.

```
>>> a = Account(100)
>>> b = Account(200)
>>> a + b
300
```

account_addition.py hosted with ♡ by GitHub view raw

Here we were able to add our two (2) Account objects with the plus (+) operator. Which is equivalent to `a.__add__(b)` .

We are not limited to addition. There are many different magic operators you can try in your spare time. Some of these include but are not limited to the following:

- `__mul__`
- `__sub__`
- `__pow__`

There is a magic method for every operation. Magic methods are even used by many Python built-in functions.

```
>>> name = 'Jordan Williams'
>>> len(name)
15
>>> name.__len__()
15
>>>
```

len.py hosted with ♡ by GitHub view raw

In the above snippet, we used the Python inbuilt function `len` . This function is just an abstraction on calling the `.__len__()` method. Once again, all these magic methods work behind the scenes.

The name variable in the above example inherits from the string (`str`) class. Look at the contents of the `str` class below.

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
```

```
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
```

str.py hosted with ♡ by GitHub   view raw

You can see the `__len__` method along with many other string methods you are used to. Running `dir` on the variable name would produce a similar result.

Another one of my personal favorite magic methods is the `__call__` method. This allows an object to behave like a function. This means you can call your objects exactly how you would call a function.

```python
class Adder:
    def __call__(self,a,b):
        print("inside call magic")
        return a + b
```

We implemented a class with the call method above.

```
>>> add = Adder()
>>> value = add(10,20)
inside call magic
>>> value
30
>>>
```

call_magic.py hosted with ♡ by GitHub   view raw

We first instantiated our object to the `add` variable. No constructor or parameters were set in the class definition, hence no parameters were placed in the constructor. After instantiation, we use that object as a function.

Calling `add(10,20)` performs addition on 10 and 20 and stores it in the variable `value` . This behavior is exactly like a function.

Let's compare it to a function.

```python
def adder(a,b):
    print("inside call magic")
    return a + b
```

We implemented a functional version of our previous class example above.

```
>>> value = adder(10,20)
inside call magic
>>> value
30
>>>
```

func_adder.py hosted with ♡ by GitHub view raw

As you can see, they both work the same.

Many of Python's built-in classes behave like this. We often confuse them for functions but they are not. The `int` and `str` both follow this principle.

```
>>> int
<class 'int'>
>>> int("3")
3
```

```
>>> str
<class 'str'>
>>> str(3)
'3'
>>>
```

example_call.py hosted with ♡ by GitHub view raw

As you can see `str` and `int` are both classes. They both have the ability to convert a value to its type acting as a function.

Summary

Here you see all the nuances of Python classes and see how it compares to that of a Java implementation. Python magic methods can give you a better understanding of how Python works behind the scenes. What I showed barely scratched the surface when it comes on to magic methods and their true abilities.

Thanks to Anupam Chugh.

Programming Python Software Development Software Engineering Data Science