Object Oriented Programming in Python

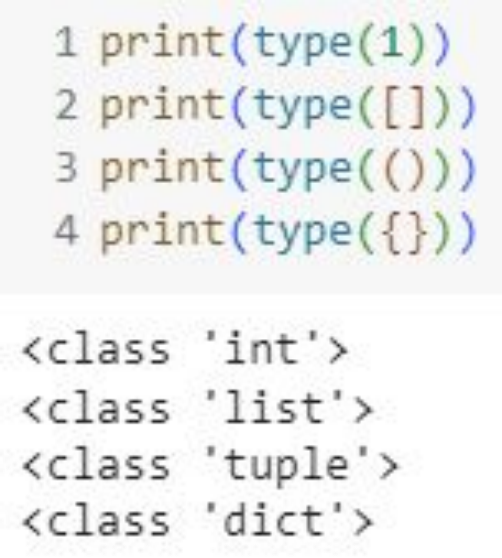
Instructor: Yiyang (Ian) Wang

Object Oriented Programming (OOP)

- Object oriented programming (OOP) allows programmers to create their own objects that have methods and attributes.
- **Recall that after defining a string, list, dictionary, or other objects, you were able to call methods off of them with the .method_name() syntax.**
- **For much larger scripts of Python code, functions by themselves aren’t enough for organization and repeatability**
- Commonly repeated tasks and objects can be defined with OOP to create code that is more usable

Objects

In Python, everything is an object. Remember we can use type() to check the type of object something is:



```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

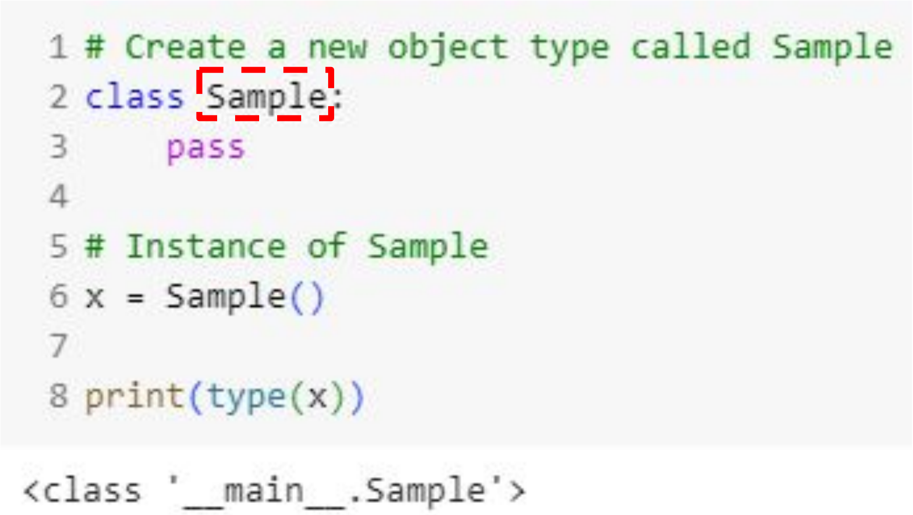
```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

We know all these things are objects, how can we create our own object types? That is where the *class* keyword comes in.

Class

The **class** is a blueprint that defines the nature of a future object.

From classes we can construct instances. An instance is a specific object created from a particular class.

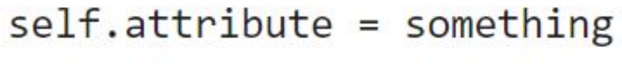


```
# Create a new object type called Sample
class Sample:
    pass

# Instance of Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

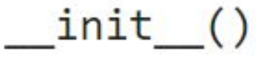
Attributes: Characteristics of an object

The syntax for creating an attribute is:



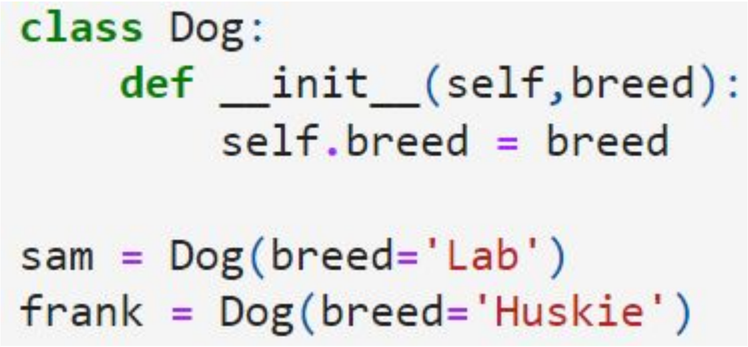
```
self.attribute = something
```

There is a special method called:



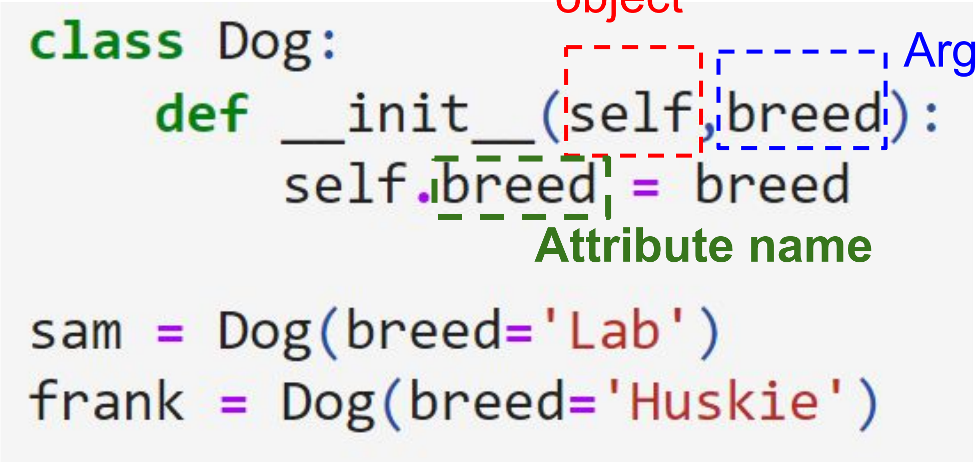
```
__init__()
```

constructor in Java



```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Attribute Example

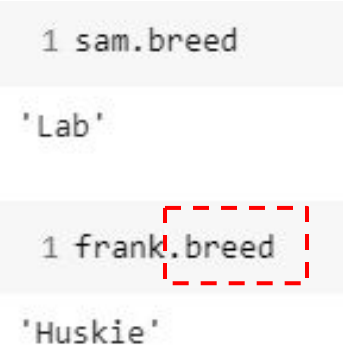


```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

a reference to the instance object

Argument name

Attribute name



```
sam.breed
```

```
'Lab'
```

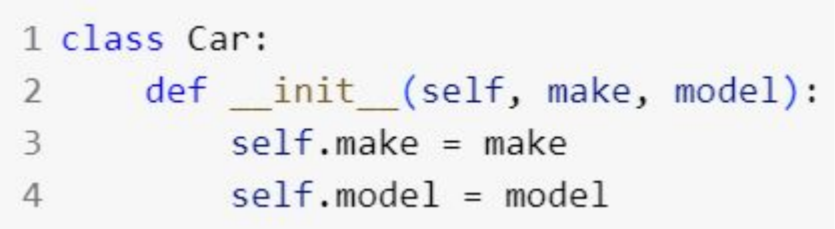
```
frank.breed
```

```
'Huskie'
```

No (): it is an attribute and doesn't take any arguments.

Question 1

Suppose you have the following Python class definition:

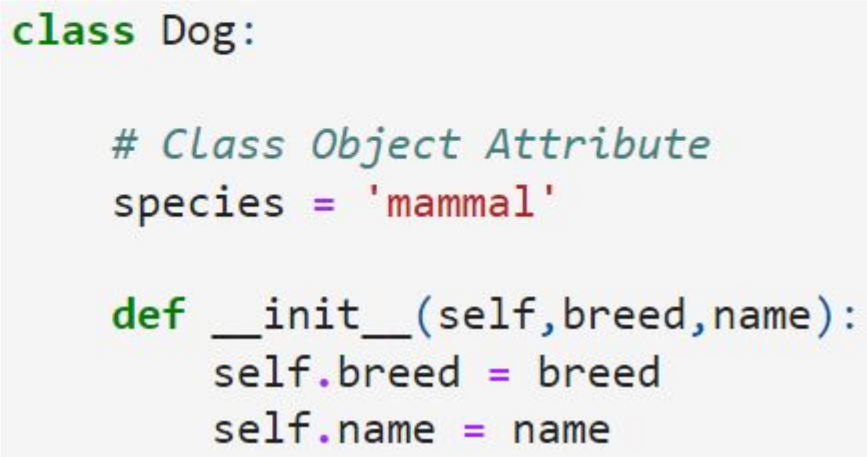


```python
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
```

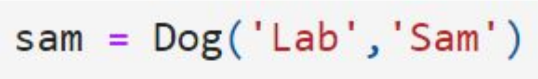
- What will happen if you try to create an instance of *Car* without providing values for *make* and *model* ?
- How can you modify the __init__ method so that make and model can be optional when creating an instance of Car?

Class Object Attributes (‘Static Variables’ in Java)

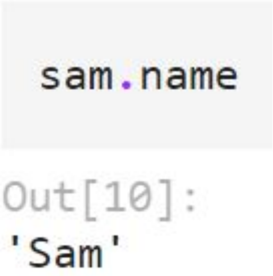
These Class Object Attributes are the same for any instance of the class.



```python
class Dog:

    # Class Object Attribute
    species = 'mammal'

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

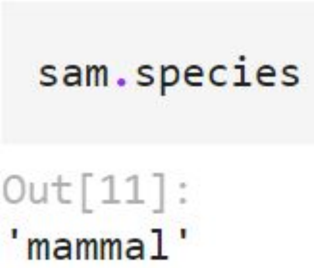


```python
sam = Dog('Lab','Sam')
```



```python
sam.name
```

```
Out[10]:
'Sam'
```



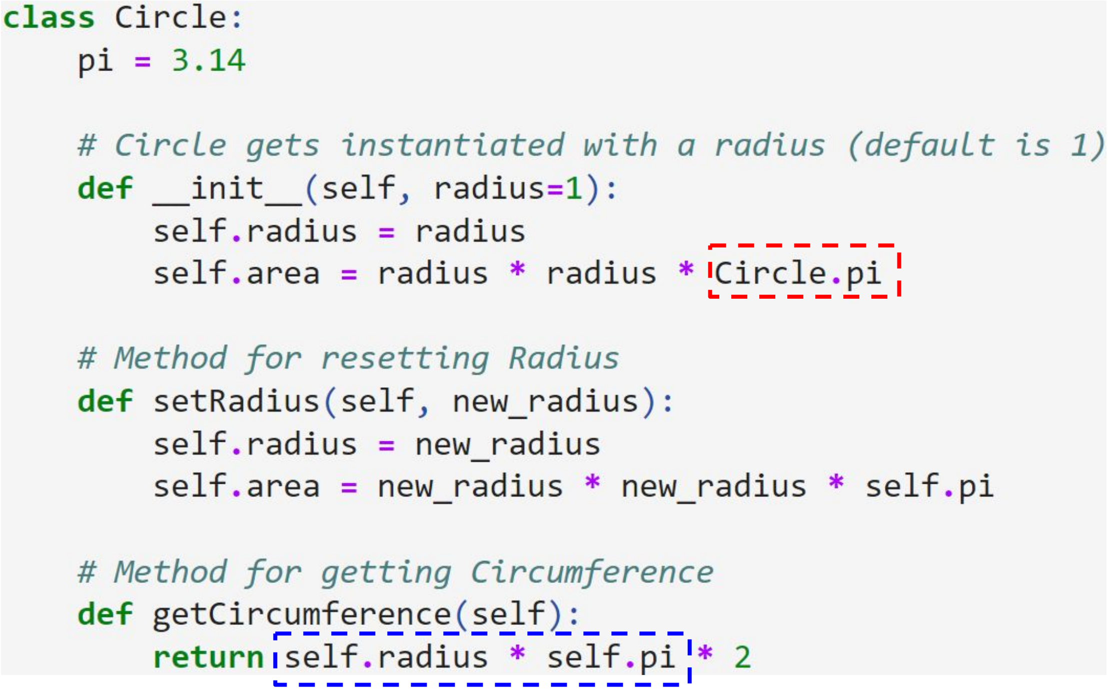
```python
sam.species
```

```
Out[11]:
'mammal'
```

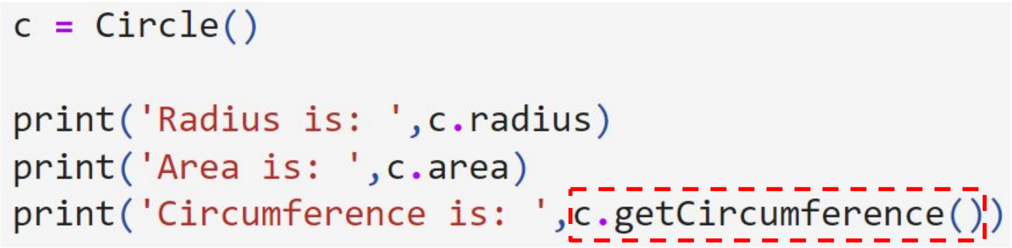
Methods

- Methods are functions defined inside the body of a class.
- They are used to perform operations with the attributes of our objects.

Methods Example



```python
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2
```



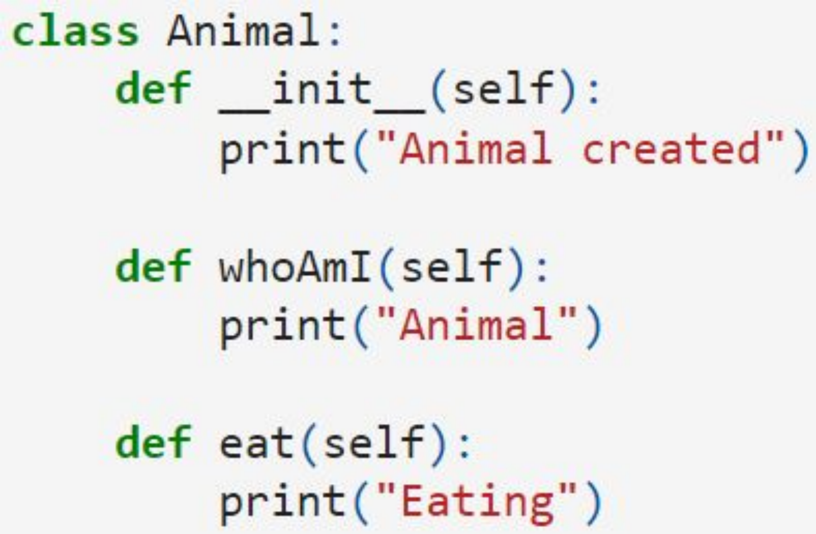
```python
c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

Inheritance

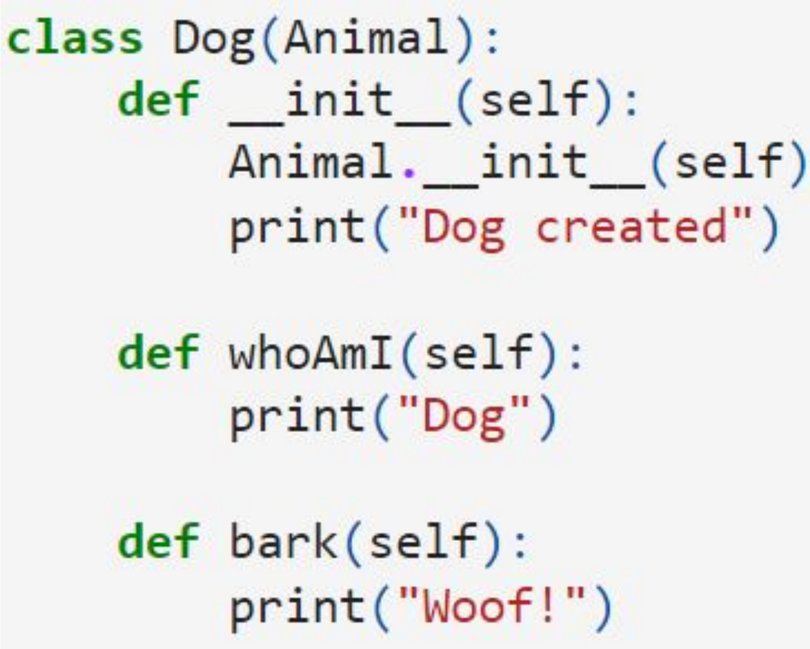
Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes.

Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Inheritance Example



```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")
```

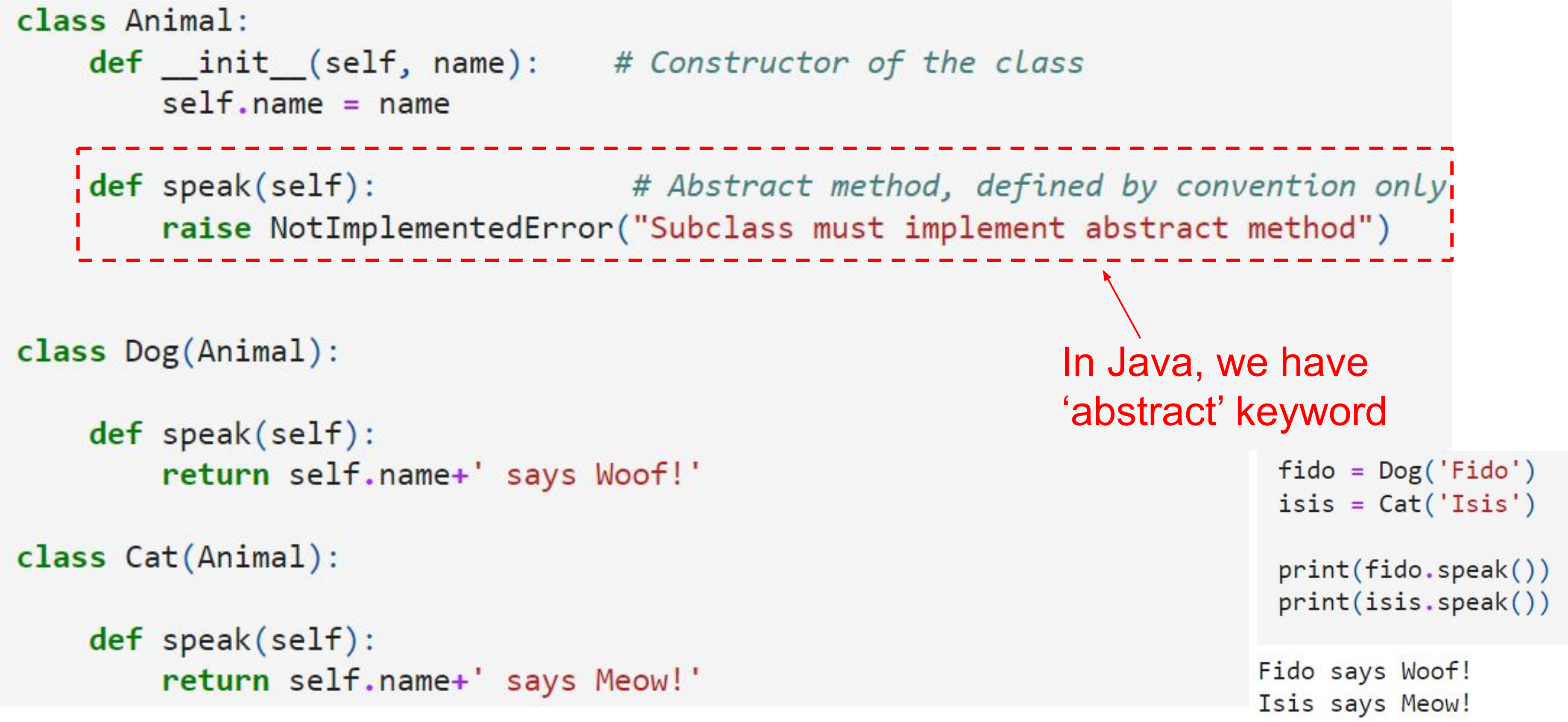


```
class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables the same method name to be used for different types of objects, where each object responds to the method in its own way.

Polymorphism Example



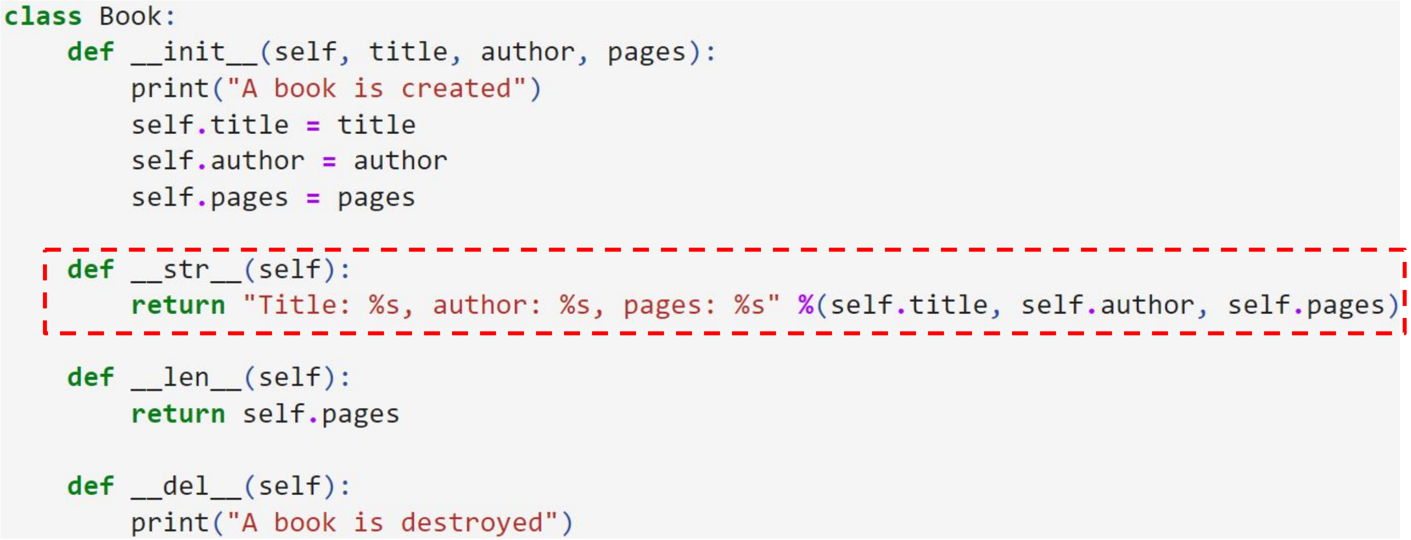
```python
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'
```

In Java, we have ‘abstract’ keyword

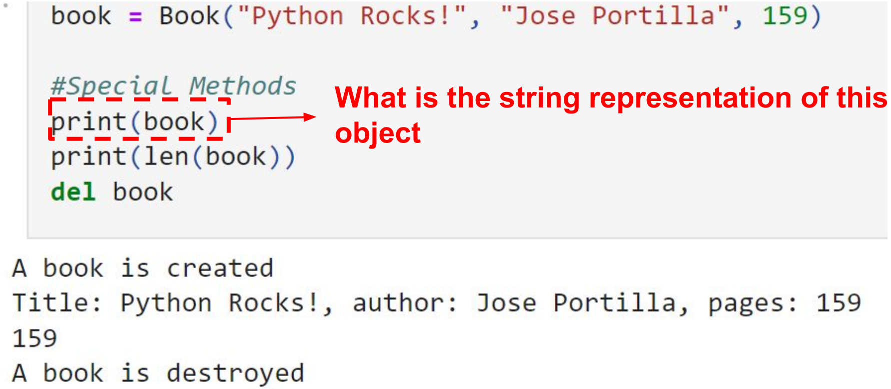
```python
fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Special (Magic / Dunder) Methods



```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```



```
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

What is the string representation of this object

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

Special (Magic / Dunder) Methods in Python

__*str*__ Method:

Defines the string representation of the object. It’s intended to return a human-readable string that represents the object, typically used for easy debugging or displaying information to the user.

__*len*__ Method:

It allows you to specify what should be considered the “length” of an object, such as the number of elements in a collection or another meaningful measure.

__*del*__ Method:

The __*del*__ method is Python’s destructor. It’s a special method that is called when an object is about to be destroyed, which usually happens when there are no more references to the object. It allows you to define cleanup actions that should be performed before an object is removed from memory, such as closing files or releasing resources.