Object-Oriented Programming

Classes

We have previously looked at two paradigms of programming - **imperative** (using statements, loops, and functions as subroutines), and **functional** (using pure functions, higher-order functions, and recursion).

Another very popular paradigm is **object-oriented programming** (OOP).
Objects are created using **classes**, which are actually the focal point of OOP.
The **class** describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition.
You can use the same class as a blueprint for creating multiple different objects.

Classes are created using the keyword **class** and an indented block, which contains class **methods** (which are functions).
Below is an example of a simple class and its objects.

```
class Cat:
  def __init__(self, color, legs):
    self.color = color
    self.legs = legs

felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)
```

This code defines a class named **Cat**, which has two attributes: **color** and **legs**.
Then the class is used to create 3 separate objects of that class.
Tap **Continue** to learn more!

__init__

The **__init__** method is the most important method in a class.
This is called when an instance (object) of the class is created, using the class name as a function.

All methods must have **self** as their first parameter, although it isn't explicitly passed, Python adds the **self** argument to the list for you; you do not need to include it when you call the methods. Within a method definition, **self** refers to the instance calling the method.

Instances of a class have **attributes**, which are pieces of data associated with them.
In this example, **Cat** instances have attributes **color** and **legs**. These can be accessed by putting a **dot**, and the attribute name after an instance.
In an **__init__** method, **self.attribute** can therefore be used to set the initial value of an instance's attributes.
Example:

Result:

```
>>>
ginger
>>>
```

In the example above, the **__init__** method takes two arguments and assigns them to the object's attributes. The **__init__** method is called the class **constructor**.

Methods

Classes can have other **methods** defined to add functionality to them.
Remember, that all methods must have **self** as their first parameter.
These methods are accessed using the same **dot** syntax as attributes.
Example:

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

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

fido = Dog("Fido", "brown")
print(fido.name)
fido.bark()
```

Try It Yourself

Result:

```
>>>
Fido
Woof!
>>>
```

Classes can also have **class attributes**, created by assigning variables within the body of the class. These can be accessed either from instances of the class, or the class itself.
Example:

```
class Dog:
 legs = 4
 def __init__(self, name, color):
  self.name = name
  self.color = color

fido = Dog("Fido", "brown")
print(fido.legs)
print(Dog.legs)
```

Try It Yourself

Result:

```
>>>
4
4
>>>
```

Class attributes are shared by all instances of the class.

Classes

Trying to access an attribute of an instance that isn't defined causes an **AttributeError**. This also applies when you call an undefined method.

Example:

```
class Rectangle:
 def __init__(self, width, height):
  self.width = width
  self.height = height

rect = Rectangle(7, 8)
print(rect.color)
```

Try It Yourself

Result:

```
>>>
AttributeError: 'Rectangle' object has no attribute 'color'
>>>
```

Tap **Try It Yourself** to play around with the code!

Inheritance

Inheritance provides a way to share functionality between classes.
Imagine several classes, **Cat**, **Dog**, **Rabbit** and so on. Although they may differ in some ways (only **Dog** might have the method **bark**), they are likely to be similar in others (all having the attributes **color** and **name**).
This similarity can be expressed by making them all inherit from a **superclass Animal**, which contains the shared functionality.
To inherit a class from another class, put the superclass name in parentheses after the class name.
Example:

```
class Animal:
 def __init__(self, name, color):
  self.name = name
  self.color = color

class Cat(Animal):
 def purr(self):
  print("Purr...")

class Dog(Animal):
 def bark(self):
  print("Woof!")

fido = Dog("Fido", "brown")
print(fido.color)
fido.bark()
```

Try It Yourself

Result:

```
>>>
brown
Woof!
>>>
```

Tap **Try It Yourself** to play around with the code!

Inheritance

A class that inherits from another class is called a **subclass**.
A class that is inherited from is called a **superclass**.
If a class inherits from another with the same attributes or methods, it overrides them.

```
class Wolf:
 def __init__(self, name, color):
  self.name = name
  self.color = color

 def bark(self):
  print("Grr...")

class Dog(Wolf):
 def bark(self):
  print("Woof")

husky = Dog("Max", "grey")
husky.bark()
```

Try It Yourself

Result:

```
>>>
Woof
>>>
```

In the example above, **Wolf** is the superclass, **Dog** is the subclass.

Inheritance

Inheritance can also be indirect. One class can inherit from another, and that class can inherit from a third class.
Example:

```
class A:
 def method(self):
  print("A method")

class B(A):
 def another_method(self):
  print("B method")

class C(B):
 def third_method(self):
  print("C method")

c = C()
c.method()
c.another_method()
c.third_method()
```

Try It Yourself

Result:

```
>>>
A method
B method
C method
>>>
```

However, circular inheritance is not possible.

Inheritance

The function **super** is a useful inheritance-related function that refers to the parent class. It can be used to find the method with a certain name in an object's superclass.
Example:

```
class A:
 def spam(self):
  print(1)

class B(A):
 def spam(self):
  print(2)
  super().spam()

B().spam()
```

Try It Yourself

Result:

```
>>>
2
1
>>>
```

super().spam() calls the **spam** method of the superclass.

Magic Methods

Magic methods are special methods which have **double underscores** at the beginning and end of their names.
They are also known as **dunders**.
So far, the only one we have encountered is **__init__**, but there are several others.
They are used to create functionality that can't be represented as a normal method.

One common use of them is **operator overloading**.
This means defining operators for custom classes that allow operators such as + and * to be used on them.
An example magic method is **__add__** for +.

```
class Vector2D:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __add__(self, other):
    return Vector2D(self.x + other.x, self.y + other.y)

first = Vector2D(5, 7)
second = Vector2D(3, 9)
result = first + second
print(result.x)
print(result.y)
```

Try It Yourself

Result:

```
>>>
8
16
>>>
```

The **__add__** method allows for the definition of a custom behavior for the + operator in our class.
As you can see, it adds the corresponding attributes of the objects and returns a new object, containing the result.
Once it's defined, we can add two objects of the class together.

Magic Methods

More magic methods for common operators:
__sub__ for -
__mul__ for *
__truediv__ for /
__floordiv__ for //
__mod__ for %
__pow__ for **
__and__ for &
__xor__ for ^
__or__ for |

The expression **x + y** is translated into **x.__add__(y)**.
However, if x hasn't implemented __add__, and x and y are of different types, then **y.__radd__(x)** is called.
There are equivalent **r** methods for all magic methods just mentioned.
Example:

```
class SpecialString:
  def __init__(self, cont):
    self.cont = cont

  def __truediv__(self, other):
    line = "=" * len(other.cont)
    return "\n".join([self.cont, line, other.cont])

spam = SpecialString("spam")
hello = SpecialString("Hello world!")
print(spam / hello)
```

Try It Yourself

Result:

```
>>>
spam
============
Hello world!
>>>
```

In the example above, we defined the **division** operation for our class **SpecialString**.

Magic Methods

Python also provides magic methods for comparisons.
__lt__ for <
__le__ for <=
__eq__ for ==
__ne__ for !=
__gt__ for >
__ge__ for >=

If **__ne__** is not implemented, it returns the opposite of **__eq__**.
There are no other relationships between the other operators.
Example:

```
class SpecialString:
  def __init__(self, cont):
    self.cont = cont

  def __gt__(self, other):
    for index in range(len(other.cont)+1):
      result = other.cont[:index] + ">" + self.cont
      result += ">" + other.cont[index:]
      print(result)

spam = SpecialString("spam")
eggs = SpecialString("eggs")
spam > eggs
```

Try It Yourself

Result:

```
>>>
>spam>eggs
e>spam>ggs
eg>spam>gs
egg>spam>s
eggs>spam>
>>>
```

As you can see, you can define any custom behavior for the overloaded operators.

Magic Methods

There are several magic methods for making classes act like containers.
__len__ for len()
__getitem__ for indexing
__setitem__ for assigning to indexed values
__delitem__ for deleting indexed values
__iter__ for iteration over objects (e.g., in for loops)
__contains__ for in

There are many other magic methods that we won't cover here, such as **__call__** for calling objects as functions, and **__int__**, **__str__**, and the like, for converting objects to built-in types.
Example:

```
import random

class VagueList:
  def __init__(self, cont):
    self.cont = cont

  def __getitem__(self, index):
    return self.cont[index + random.randint(-1, 1)]

  def __len__(self):
    return random.randint(0, len(self.cont)*2)

vague_list = VagueList(["A", "B", "C", "D", "E"])
print(len(vague_list))
print(len(vague_list))
print(vague_list[2])
print(vague_list[2])
```

Try It Yourself

Result:

```
>>>
6
7
D
C
>>>
```

We have overridden the len() function for the class VagueList to return a random number.
The indexing function also returns a random item in a range from the list, based on the expression.

Object Lifecycle

The lifecycle of an object is made up of its **creation**, **manipulation**, and **destruction**.

The first stage of the life-cycle of an object is the **definition** of the class to which it belongs.
The next stage is the **instantiation** of an instance, when **__init__** is called. Memory is allocated to store the instance. Just before this occurs, the **__new__** method of the class is called. This is usually overridden only in special cases.
After this has happened, the object is ready to be used.

Other code can then interact with the object, by calling functions on it and accessing its attributes.
Eventually, it will finish being used, and can be **destroyed**.

Object Lifecycle

When an object is **destroyed**, the memory allocated to it is freed up, and can be used for other purposes.
Destruction of an object occurs when its **reference count** reaches zero. Reference count is the number of variables and other elements that refer to an object.
If nothing is referring to it (it has a reference count of zero) nothing can interact with it, so it can be safely deleted.

In some situations, two (or more) objects can be referred to by each other only, and therefore can be deleted as well.
The **del** statement reduces the reference count of an object by one, and this often leads to its deletion.
The magic method for the **del** statement is **__del__**.
The process of deleting objects when they are no longer needed is called **garbage collection**.
In summary, an object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary). The object's reference count decreases when it's deleted with **del**, its reference is reassigned, or its reference goes out of scope. When an object's reference count reaches zero, Python automatically deletes it.
Example:

```
a = 42  # Create object <42>
b = a  # Increase ref. count  of <42>
c = [a]  # Increase ref. count  of <42>

del a  # Decrease ref. count  of <42>
b = 100  # Decrease ref. count  of <42>
c[0] = -1  # Decrease ref. count  of <42>
```

Lower level languages like C don't have this kind of automatic memory management.

Data Hiding

A key part of object-oriented programming is **encapsulation**, which involves packaging of related variables and functions into a single easy-to-use object - an instance of a class.
A related concept is **data hiding**, which states that implementation details of a class should be hidden, and a clean standard interface be presented for those who want to use the class.
In other programming languages, this is usually done with private methods and attributes, which block external access to certain methods and attributes in a class.

The Python philosophy is slightly different. It is often stated as **"we are all consenting adults here"**, meaning that you shouldn't put arbitrary restrictions on accessing parts of a class. Hence there are no ways of enforcing a method or attribute be strictly private.

However, there are ways to discourage people from accessing parts of a class, such as by denoting that it is an implementation detail, and should be used at their own risk.

Data Hiding

Weakly private methods and attributes have a **single underscore** at the beginning.
This signals that they are private, and shouldn't be used by external code. However, it is mostly only a convention, and does not stop external code from accessing them.
Its only actual effect is that **from module_name import *** won't import variables that start with a single underscore.
Example:

```
class Queue:
  def __init__(self, contents):
    self._hiddenlist = list(contents)

  def push(self, value):
    self._hiddenlist.insert(0, value)

  def pop(self):
    return self._hiddenlist.pop(-1)

  def __repr__(self):
    return "Queue({})".format(self._hiddenlist)
```

```
queue = Queue([1, 2, 3])
print(queue)
queue.push(0)
print(queue)
queue.pop()
print(queue)
print(queue._hiddenlist)
```

Try It Yourself

Result:

```
>>>
Queue([1, 2, 3])
Queue([0, 1, 2, 3])
Queue([0, 1, 2])
[0, 1, 2]
>>>
```

In the code above, the attribute **_hiddenlist** is marked as private, but it can still be accessed in the outside code.
The **__repr__** magic method is used for string representation of the instance.

Data Hiding

Strongly private methods and attributes have a **double underscore** at the beginning of their names. This causes their names to be mangled, which means that they can't be accessed from outside the class.
The purpose of this isn't to ensure that they are kept private, but to avoid bugs if there are subclasses that have methods or attributes with the same names.
Name mangled methods can still be accessed externally, but by a different name. The method **__privatemethod** of class **Spam** could be accessed externally with **_Spam__privatemethod**.
Example:

```
class Spam:
 __egg = 7
 def print_egg(self):
  print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
print(s.__egg)
```

Try It Yourself

Result:

```
>>>
7
7
AttributeError: 'Spam' object has no attribute '__egg'
>>>
```

Basically, Python protects those members by internally changing the name to include the class name.

Class Methods

Methods of objects we've looked at so far are called by an instance of a class, which is then passed to the **self** parameter of the method.
Class methods are different - they are called by a class, which is passed to the **cls** parameter of the method.
A common use of these are factory methods, which instantiate an instance of a class, using different parameters than those usually passed to the class constructor.
Class methods are marked with a **classmethod** decorator.
Example:

```
class Rectangle:
  def __init__(self, width, height):
    self.width = width
    self.height = height

  def calculate_area(self):
    return self.width * self.height

  @classmethod
  def new_square(cls, side_length):
    return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

Try It Yourself

Result:

```
>>>
25
>>>
```

new_square is a class method and is called on the class, rather than on an instance of the class. It returns a new object of the class **cls**.

Technically, the parameters **self** and **cls** are just conventions; they could be changed to anything else. However, they are universally followed, so it is wise to stick to using them.

Static Methods

Static methods are similar to class methods, except they don't receive any additional arguments; they are identical to normal functions that belong to a class.
They are marked with the **staticmethod** decorator.
Example:

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings

  @staticmethod
  def validate_topping(topping):
    if topping == "pineapple":
      raise ValueError("No pineapples!")
    else:
      return True
```

```
ingredients = ["cheese", "onions", "spam"]
if all(Pizza.validate_topping(i) for i in ingredients):
  pizza = Pizza(ingredients)
```

Try It Yourself

Static methods behave like plain functions, except for the fact that you can call them from an instance of the class.

Properties

Properties provide a way of customizing access to instance attributes.
They are created by putting the **property** decorator above a method, which means when the instance attribute with the same name as the method is accessed, the method will be called instead.
One common use of a property is to make an attribute **read-only**.
Example:

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings

  @property
  def pineapple_allowed(self):
    return False

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

Try It Yourself

Result:

```
>>>
False

AttributeError: can't set attribute
>>>
```

Tap **Try It Yourself** to play around with the code!

Properties

Properties can also be set by defining **setter/getter** functions.
The **setter** function sets the corresponding property's value.
The **getter** gets the value.
To define a **setter**, you need to use a decorator of the same name as the property, followed by a dot and the **setter** keyword.
The same applies to defining **getter** functions.
Example:

```
class Pizza:
  def __init__(self, toppings):
    self.toppings = toppings
    self._pineapple_allowed = False

  @property
  def pineapple_allowed(self):
    return self._pineapple_allowed

  @pineapple_allowed.setter
  def pineapple_allowed(self, value):
    if value:
      password = input("Enter the password: ")
      if password == "Sw0rdf1sh!":
        self._pineapple_allowed = value
      else:
        raise ValueError("Alert! Intruder!")

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
print(pizza.pineapple_allowed)
```

Try It Yourself

Result:

```
>>>
False
Enter the password: Sw0rdf1sh!
True
```

Tap **Try It Yourself** to play around with the code!

A Simple Game

Object-orientation is very useful when managing different objects and their relations. That is especially useful when you are developing games with different characters and features.

Let's look at an example project that shows how classes are used in game development.
The game to be developed is an old fashioned text-based adventure game.
Below is the function handling input and simple parsing.

```
def get_input():
  command = input(": ").split()
  verb_word = command[0]
  if verb_word in verb_dict:
    verb = verb_dict[verb_word]
  else:
    print("Unknown verb {}". format(verb_word))
    return

  if len(command) >= 2:
    noun_word = command[1]
    print (verb(noun_word))
  else:
    print(verb("nothing"))

def say(noun):
  return 'You said "{}"'.format(noun)
```

```
verb_dict = {
  "say": say,
}

while True:
  get_input()
```

Result:

```
>>>
: say Hello!
You said "Hello!"
: say Goodbye!
You said "Goodbye!"

: test
Unknown verb test
```

The code above takes input from the user, and tries to match the first word with a command in **verb_dict**. If a match is found, the corresponding function is called.

A Simple Game

The next step is to use classes to represent game objects.

```
class GameObject:
  class_name = ""
  desc = ""
  objects = {}

  def __init__(self, name):
    self.name = name
    GameObject.objects[self.class_name] = self

  def get_desc(self):
    return self.class_name + "\n" + self.desc

class Goblin(GameObject):
  class_name = "goblin"
  desc = "A foul creature"

goblin = Goblin("Gobbly")

def examine(noun):
  if noun in GameObject.objects:
    return GameObject.objects[noun].get_desc()
  else:
    return "There is no {} here.".format(noun)
```

We created a **Goblin** class, which inherits from the **GameObjects** class.
We also created a new function **examine**, which returns the objects description.
Now we can add a new "examine" verb to our dictionary and try it out!

```
verb_dict = {
  "say": say,
  "examine": examine,
}
```

Combine this code with the one in our previous example, and run the program.

```
>>>
: say Hello!
You said "Hello!"

: examine goblin
goblin
A foul creature

: examine elf
There is no elf here.
:
```

Combine this code with the one in our previous example, and run the program.

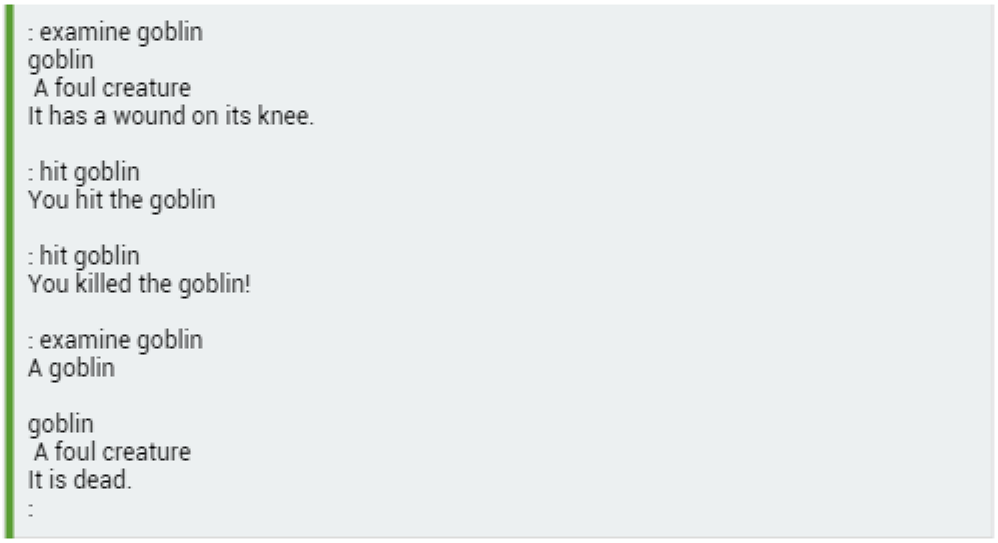
A Simple Game

This code adds more detail to the **Goblin** class and allows you to **fight** goblins.

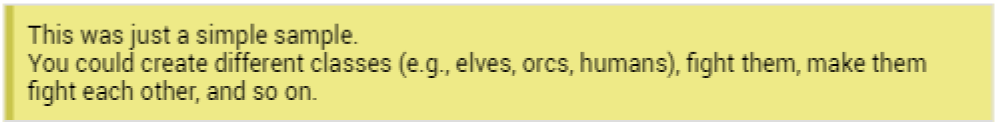
```
class Goblin(GameObject):
  def __init__(self, name):
    self.class_name = "goblin"
    self.health = 3
    self._desc = " A foul creature"
    super().__init__(name)

  @property
  def desc(self):
    if self.health >=3:
      return self._desc
    elif self.health == 2:
      health_line = "It has a wound on its knee."
    elif self.health == 1:
      health_line = "Its left arm has been cut off!"
    elif self.health <= 0:
      health_line = "It is dead."
    return self._desc + "\n" + health_line

  @desc.setter
  def desc(self, value):
    self._desc = value

def hit(noun):
  if noun in GameObject.objects:
    thing = GameObject.objects[noun]
    if type(thing) == Goblin:
      thing.health = thing.health - 1
      if thing.health <= 0:
        msg = "You killed the goblin!"
      else:
        msg = "You hit the {}".format(thing.class_name)
  else:
    msg ="There is no {} here.".format(noun)
  return msg
```

Result:

```
>>>
: hit goblin
You hit the goblin
```



```
: examine goblin
goblin
 A foul creature
It has a wound on its knee.

: hit goblin
You hit the goblin

: hit goblin
You killed the goblin!

: examine goblin
A goblin

goblin
 A foul creature
It is dead.
:
```



This was just a simple sample.
You could create different classes (e.g., elves, orcs, humans), fight them, make them fight each other, and so on.

End.