Objects in Python

Intro to OOP

This is a quick, 15-minute review of Python's syntax for object-oriented programming.

If you are familiar with Python classes and objects already... see you in 15 minutes!

Creating classes

Use the `class` keyword. `__init__` is the constructor.

All methods, including `__init__`, take `self` as their first argument. Similar to "`this`" in other languages, but explicit.

```
# Python 3
class Pet:
    def __init__(self, name):
        self.name = name
```

Create objects by calling the class name like a function - no "new" keyword. The "self" parameter is automatically inserted.

```
>>> pet = Pet("Fido")
>>> pet.name
'Fido'
```

Inheritance

Declare subclasses on the `class` line, with parentheses.

```
# Python 3, still
class Pet:
    def __init__(self, name):
        self.name = name
class Dog(Pet):
    pass
class Cat(Pet):
    pass
```

```
>>> pet = Dog("Fido")
>>> pet.name
'Fido'
>>> isinstance(pet, Pet)
True
>>> isinstance(pet, Dog)
True
>>> isinstance(pet, Cat)
False
```

`isinstance` is a built-in function you can use to check object types

Methods

Methods are defined just like functions, except:

- Indented in the class, and
- take `self` as the first argument.

`self` is always implicitly prepended.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    def full_name(self):
        return self.first + " " + self.last
    def formal_name(self, title):
        return title + " " + self.full_name()
```

```
>>> guy = Person("John", "Smith")
>>> guy.formal_name("Mr.")
'Mr. John Smith'
```

Everything inherits from "object"

In modern Python, all classes need to inherit from a built-in base class called `object`. In Python 2, you must explicity declare it:

```
>>> # Python 2
... class PetFromObject(object):
...     def __init__(self, name):
...         self.name = name
>>> class PetNotFromObject:
...     def __init__(self, name):
...         self.name = name
>>> issubclass(PetFromObject, object)
True
>>> issubclass(PetNotFromObject, object)
False
```

Everything inherits from "object"

In Python 3, all classes automatically inherit from `object`.

```
>>> # Python 3
... class PetFromObject:
...     def __init__(self, name):
...         self.name = name
>>> issubclass(PetFromObject, object)
True
```

Inheriting from `object` gives you many benefits, and no downsides (unless you are working with ancient legacy Python code).

One benefit: your subclass methods can invoke the superclass version.

Superclass methods

In Python 3, invoke the superclass' method using `super()`.

```
class LapDog:
    def speak(self):
        return "Yip!"
class LoudLapDog(LapDog):
    def speak(self):
        sound = super().speak().upper()
        return sound + sound + sound
```

```
>>> fifi = LoudLapDog()
>>> fifi.speak()
'YIP!YIP!YIP!'
```

Superclass methods (Py 2)

In Python 2, you need to pass arguments to `super()`.

```
class LapDog(object): # Note the (object)
    def speak(self):
        return "Yip!"
class LoudLapDog(LapDog):
    def speak(self):
        # super(LoudLapDog, self).speak refers to LapDog.speak
        sound = super(LoudLapDog, self).speak().upper()
        return sound + sound + sound
```

```
>>> fifi = LoudLapDog()
>>> fifi.speak()
'YIP!YIP!YIP!'
```

Default Attribute Values

Classes can define attributes (member variables). Each instance gets its own copy to modify.

```
>>> class Coin:
...     value = 1
...
>>> penny = Coin()
>>> penny.value
1
>>>
>>> nickel = Coin()
>>> nickel.value = 5
>>> nickel.value
5
>>>
>>> another_penny = Coin()
>>> another_penny.value
1
```

Overriding Values

A subclass can change the value.

```
class Pet:
    sound = ""
    def speak(self):
        print("The pet says: " + self.sound)
class Dog(Pet):
    sound = "Woof!"
class Cat(Pet):
    sound = "Meow"
class Turtle(Pet):
    pass
```

```
>>> Dog().speak()
'The pet says: Woof!'
>>> Cat().speak()
'The pet says: Meow'
>>> Turtle().speak()
'The pet says: '
```

No Multiple Dispatch

In some languages, you can define a method twice with different signatures.

Not in Python. The second definition **silently masks** the first one.

```
>>> class BarkingDog:
...     def speak(self):
...         return "Bark"
...     def speak(self, punctuation):
...         return "BARK" + punctuation
...
>>> jojo = BarkingDog()
>>> jojo.speak("!")
'BARK!'
>>> jojo.speak()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: speak() missing 1 required positional argument: 'punctuation'
```

"Private" Methods and Attributes

Python doesn't have private variables or methods, like Java and C++.

Convention: prefixing with a single underscore means "don't rely on this being available." But the language doesn't enforce it.

```
class SignalParser:
    def __init__(self):
        self._state = 'waiting'
    def receive(self):
        self._state = 'receiving'
```

```
>>> sp = SignalParser()
>>> sp.receive()
>>> print("Mwahaha, I can see your hidden state is " + sp._state)
Mwahaha, I can see your hidden state is receiving
```

A step better

You can prefix the member attribute or method with ___, and Python will make it harder to access.

```
class SignalParser:
    def __init__(self):
        self.__state = 'waiting'
    def receive(self):
        self.__state = 'receiving'
```

```
>>> sp = SignalParser()
>>> sp.receive()
>>> print("I cannot read your hidden state! " + sp.__state)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SignalParser' object has no attribute '__state'
```

But actually, it just mangles the name:

```
>>> print("Oh wait, I CAN! " + sp._SignalParser__state)
Oh wait, I CAN! receiving
```

Special Methods

Python's object system has special methods you can implement, surrounded by pairs of underscores. These are called *magic methods*.

```
class Money:
      def __init__(self, dollars, cents):
          self.dollars = dollars
          self.cents = cents
      def __str__(self):
          return "${}.{:02}".format(self.dollars, self.cents)
```

```
>>> print(Money(2,3))
$2.03
```

In speech, we say "dunder str", rather than "underscore underscore str underscore underscore".