Classes and Objects

Built-in objects

- You are already familiar with several kinds of objects: strings, lists, sets, tuples, and dictionaries
- An ***object*** has two aspects:
  - Some ***fields*** (or ***instance variables***) containing data, such as numbers, booleans, or other objects; these describe the ***state*** of the object
  - Some ***methods*** that provide means of examining or manipulating the object
- In other words, an object bundles together data, and methods for working with that data
- **Example:**
  - The list `[1, 2, 3]` is an object
  - If `s = [1, 2, 3]`, then `s` is a *name* for the object, but it isn't *part* of the object
  - `append` is a method you can use with lists: If `s` is as defined above, `s.append(4)` changes the named list to have the value `[1, 2, 3, 4]`
- Objects are sometimes called ***instances***, or ***instances of (some class)***

Functions and methods

- ***Functions*** are independent of objects, and "stand alone"
  - We ***call*** functions to get a result
  - **Examples:**
    - `len(`*list*`)` tells the number of things in the *list*
    - `len(`*string*`)` tells the number of characters in the *string*
- ***Methods*** are associated with objects
  - "Calling" a method is best thought of as ***talking to*** the object
  - The technical jargon is ***sending a message to*** the object
  - If `s` is a list, then `s.append(4)` is saying, "`s`, append 4 to yourself"
- Methods can:
  - Ask the object to tell us something (usually about itself)
  - Tell the object to modify its state in some way
  - Tell the object to give us a modified copy of itself

Dot notation

- Dots (periods) are used in two very similar ways:
  - When you "talk to" an object, you name the object, put a dot, and then the message
    - **Example:** `my_list.append(another_element)`
      - In this example the "message" contains additional information (`another_element`)
        - The object changes itself (by adding an element)
    - **Example:** `low_string = my_string.lower()`
      - In this example, no additional information is required
      - The object is unchanged, but returns a new, similar object
- When you send a message to a module, you name the module, put a dot, and then the message (the function you want the module to execute)
  - **Example:** `import copy ; new_list = copy.deepcopy(my_list)`
  - `copy.deepcopy` is a ***qualified name***

Special syntax

- For convenience, all the built-in objects have quite a bit of special syntax
  - For example, while you can do things like `my_list.append(an_element)` (usual object syntax), you can also do things like `my_list + my_other_list` (special syntax)
    - `my_list[index]` is yet more special syntax
- In Python (unlike Java), numbers and booleans are also objects
  - In fact, in Python, **everything is an object!**
- There is *so much* special syntax associated with numbers and booleans that we almost never use the standard object notation
  - **Example:** `f.is_integer()` returns `True` if the floating point number `f` has an integral value (like `2.0`)

Classes and objects

- A ***class*** is a recipe for creating objects
  - Classes define the fields (or instance variables) and methods that each object of the class will have
  - The methods are shared by all objects of that class
  - The fields are **not** shared; every object has its own
- A class is sometimes described as a "blueprint," or as a "cookie cutter," since the primary purpose of a class is to describe objects
- Everything in Python is an object
  - Hence, classes are themselves objects--more on this later

Example class

```
class Person:
        "Simple example of a class"

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

        def get_older(self, n = 1):
            self.age += n
            return self.age

        def get_first_name(self):
            return self.name.split()[0]
```

Defining a class

- **Syntax:**

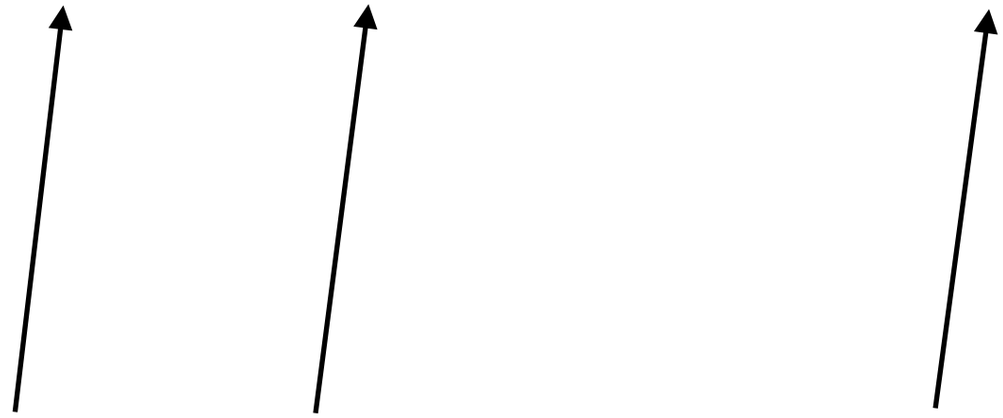
```
class NameOfClass:
    """Documentation string (optional)"""
    Method definitions
```

- By convention, class names start with a capital letter, and are CamelCase
- Every method definition has the word **`self`** as its first parameter
  - Exceptions to this rule will be covered later
- There is almost always one special method, called **`__init__`**, used to construct new objects of this class
  - That's eight characters: **`_`**, **`_`**, **`i`**, **`n`**, **`i`**, **`t`**, **`_`**, **`_`**

Parameters and arguments

- You define a method like this:

```
def methodname(self, par1, ..., parN)
```



- But you call it like `object(arg1, ..., argN)`
- How do arguments match up to parameters?
- They match up by position

Initializing an object

- Almost every class you write will have an **`__init__`** method
- The purpose of the **`__init__`** method is to initialize some instance variables of the object, usually based on the parameters
- **Example:**

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

  - This example (in the **`Person`** class) will:
    - Create two instance variables in the object, **`self.name`** and **`self.age`**
    - Provide initial values for the instance variables
  - In this example, the initial values for the variables are just copied from the parameters, but you can set them any way you like
- Although you define the **`__init__`** method in your class, you ***don't call it!***

Creating an object

- To create an object, you use the name of the class, followed by some arguments in parentheses
  - Example:
    ```
    >>> jenny = Person("Jennifer Jones", 23)
    ```
- We can demonstrate that this worked
  - ```
    >>> jenny
    <__main__.Person object at 0x10666b470>
    ```
- Although it is questionable style (as will be explained later), we can see that the "internals" of the object have been correctly initialized
  - ```
    >>> jenny.name
    'Jennifer Jones'
    ```
  - ```
    >>> jenny.age
    23
    ```
- You can see from the above that when we created the object, the **`__init__`** method was automatically called with the ***two*** given parameters (**`name`** and **`age`**) and the new object (**`self`**)
    ```

A class without `__init__`

- ```
  >>> class Boring:
      pass
  ```

- ```
  >>> blah = Boring()
  ```

- ```
  >>> blah
  <__main__.Boring object at 0x105ad2cc0>
  ```

- Objects like this are not necessarily useless
  - They can hold methods
  - Instance variables can be added later
  ```

Talking to an object

- To use the instance variables or instance methods of an object, you name the object, put a dot, and then the name of the variable or method
  - `>>> jenny.get_older()`
    `24`
- But the object refers to itself by using the name “self”
  - ```
    def get_older(self, n = 1):
        self.age += n
        return self.age
    ```
  - What actually happens is that **`jenny`**, although listed separately from the other arguments, ***is*** an argument, and it gets passed into the **`self`** parameter
    ```

Special functions

- **`__init__`** is a special function; if you define it, Python can use it
- Another special function is **`__str__`**, which is used by the **`str`** and **`print`** methods to provide a string useful for printing
  - ```
    def __str__(self):
        return self.name
    ```
  - ```
    >>> print(jenny)
    Jennifer Jones
    ```
  - ```
    >>> str(jenny)
    'Jennifer Jones'
    ```
- Another special function is **`__repr__`**, whose purpose is to provide a representation of the object that could be used by **`eval`** to recreate the object
  - ```
    def __repr__(self):
        return "Person('" + self.name + "'," + str(self.age)
    + ")"
    ```
  - ```
    >>> print(repr(jenny))
    Person('Jennifer Jones',23)
    ```
  - ```
    >>> eval(repr(jenny))
    Person('Jennifer Jones’,23)
    ```

Special variables

- The documentation string of a function can be retrieved with the **`__doc__`** special variable
  - **`>>> jenny.__doc__`**
    `'Simple example of a class'`
- A module's **`__name__`** is set equal to **`'__main__'`** when read from standard input, a script, or from an interactive prompt.
  - If this file is being imported from another module, **`__name__`** will be set to the name of that module
  - As a result, we have this common idiom:
    - **`if __name__ == '__main__':`**
      Call to the function that starts the program

Subclasses

- A new class can ***extend*** a previously-defined class and add new instance variables and methods
- Such a class is called a ***subclass*** of the earlier class
- To create a subclass, put the name of the ***superclass*** in parentheses after the name of the subclass
  - `class Employee(Person):`
    `pass`
- The subclass ***inherits*** the variables and methods defined in the superclass
  - `>>> sam = Employee('Sam Smith', 40)`
  - `>>> sam`
    `Person('Sam Smith',40)`
- The ***type*** of the new object is the superclass type
  - `>>> type(sam)`
    `<class '__main__.Employee'>`

Creating an instance of a subclass

- A subclass ***inherits*** the variables and methods of its superclass
- A subclass can (and usually does) ***extend*** the superclass with additional variables and methods
- To initialize any additional instance variables, the subclass usually has its own **`__init__`** method
  - **`class Employee(Person):`**

```
    def __init__(self, name, role):
        super().__init__(name, -1)
        self.role = role
        self.age = 'irrelevant'
```

- To ***extend*** a Person object, we must first ***have*** a Person object
  - In a subclass, we can refer to the methods of the superclass with **`super()`**
  - The first thing to do is to explicitly call **`super().__init__`**
  - Then we can add instance variables (**`role`**) or modify existing ones (**`age`**)

Overriding

- When we have the same method in a subclass as in a superclass, a subclass instance will use its own version
  - This is called ***overriding*** a method
  - Instances (objects) of the superclass will continue to use the method defined there
- **Example** (this would be bad):

```
def __init__(self, name, role):
    __init__(name, -1) # infinite
recursion
```

  - `super()` lets us avoid this default behavior
  -

```
def __init__(self, name, role):
        super().__init__(name, -1)
```

Overriding II

- ```
  class Employee(Person):

      def __init__(self, name, role):
          super().__init__(name, -1)
          self.role = role
          self.age = 'irrelevant'

      def __str__(self):
          if self.role == 'professor':
              return 'Dr. ' + self.name
          else:
              return self.name
  ```
- ```
  >>> jenny = Employee('Jennifer Jones', 'professor')
  >>> sam = Employee('Sam Smith', ‘clerk')
  ```
- ```
  >>> jenny.age
  ‘irrelevant'
  ```
- ```
  >>> jenny.name
  'Jennifer Jones'
  ```
- ```
  >>> sam.name
  'Sam Smith'
  ```
- ```
  >>> print(jenny)
  Dr. Jennifer Jones
  ```

Classes are objects, too

- Classes can have ***attributes***, or ***class variables***, and can have class methods
  - These are the same for **every** object of that class
- ```
  class Person:
      species = 'human'

      def get_species():
          return Person.species
  ```
- ```
  >>> jenny = Person("Jennifer Jones", 23)
  ```
- ```
  >>> bill = Person('William Brown', 48)
  ```
- ```
  >>> jenny.species
  'human'
  ```
- ```
  >>> bill.species
  'human'
  ```
- ```
  >>> Person.get_species()
  'human'
  ```
- ```
  >>> bill.get_species()
  ...
  TypeError: get_species() takes 0 positional arguments but 1
  was given
  ```
  ```

Classes are objects, too

- Classes can have ***attributes***, or ***class variables***, and can have class methods
  - These are the same for **every** object of that class
- ```
  class Person:
      species = 'human'

      def get_species():
          return Person.species
  ```
- ```
  >>> jenny = Person("Jennifer Jones", 23)
  ```
- ```
  >>> bill = Person('William Brown', 48)
  ```
- ```
  >>> jenny.species
  'human'
  ```
- ```
  >>> bill.species
  'human'
  ```
- ```
  >>> Person.species
  'human'
  ```
- ```
  >>> Person.get_species()
  'human'
  ```
- ```
  >>> bill.get_species()
  ...
  TypeError: get_species() takes 0 positional arguments but 1 was given
  ```

The End

User, n. The word computer professionals use when they mean “idiot.”

~Dave Barry