CLASSES

- A class instance knows its class type
  - Because each instance has as its type the class that it was made from, an instance remembers its class
  - This is often called the ***instance-of*** relationship
  - It is stored in the __class__ attribute of the instance

```
1   class Person:
2       def __init__(self, name, age):
3           self.name = name
4           self.age = age
5
6   def main():
7       john = Person('John', 34)
8
9       print(john.__class__) # prints: <class '__main__.Person'>
10
11  main()
```

CLASSES

- methods vs. functions
  - A method and a function are closely related.
    - They are both "small programs" that have parameters (zero or more)
    - They perform some operation and (potentially) return a value
  - The main difference is that methods are functions that are tied to a particular object (for example a class instance)

CLASSES

- There is a difference in how we call a method and a function
  - functions are independent and can be called via their name
  - methods are called in the context of an object
    - Here a function called do_something is called:

```
do_something(param1)
```

    - Here a method called do_something is called:

```
an_object.do_something(param1)
```

This means that the object that the method is called on is *always implicitly a parameter (as self)*!

CLASSES

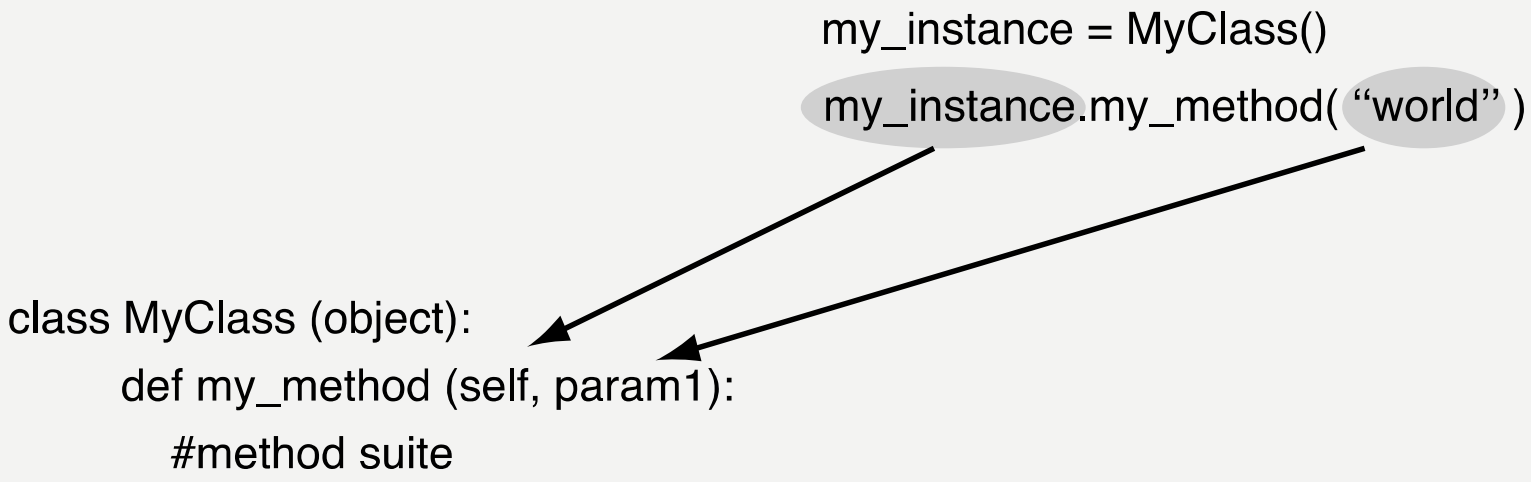
- There is a difference in how we define methods and functions
  - methods are defined ***inside*** a class
  - methods always bind the first parameter in the definition to the object that called it (as self)
  - Do note that this parameter can be named anything, but traditionally it is named ***self***

```
1   class MyClass:
2       
3       def my_method(self,param1):
4           #some code
```

CLASSES

- `self` is an important variable.
- In any method of a class, self is bound to the object that called that method
- It is through `self` that we can access the instance that called the method (and all of its attributes as a result)

CLASSES
BINDING SELF



CLASSES

- Consider the following code sample

```
1   class Person:
2       def __init__(self, name, age):
3           self.name = name
4           self.age = age
5
6       def print_age(self):
7           print("The age of {} is {} years old.".format(self.name, self.age))
8
9   def main():
10
11      john = Person('John', 34)
12
13      john.print_age()
14
15
16  main()
```

Here the variable john is implicitly sent in as the first parameter to the __init__ method within the Person class

Here the variable john is implicitly sent in as the first parameter to the print_age method within the Person class

CLASSES

- self is bound for us!
  - when a dot method call is made, the object that called the method is **automatically** assigned to the **`self`** parameter
  - we can use **`self`** to remember, and therefore refer, to the calling object
  - To reference any part of the calling object within the class, we must always precede it with **`self`**
    - This goes for methods and attributes

CLASSES

- Class namespaces are dicts
  - the namespaces in every object and module is indeed a namespace
  - that dictionary is bound to the special variable __dict__
  - it lists all the local attributes (variables, functions) in the object

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

    def print_age(self):
        print("The age of {} is {} years old.".format(self.name, self.age))

def main():

    john = Person('John', 34)
    mary = Person('Mary', 31)

    print(john.__dict__) # prints: {'name': 'John', 'age': 34}
    print(mary.__dict__) # prints: {'name': 'Mary', 'age': 31}

main()
```

CLASSES

- private variables in an instance
  - many OOP approaches allow you to make attributes and methods in a class instance ***private***
  - private means that these fields are not accessible by the class user, only the class creator
  - there are many advantages to controlling who can access the instance private values

CLASSES

- Python is different from many other OOP languages because there is no private keyword that enforces the privacy for a class field
  - Python takes the approach "We are all adults here". No hard restrictions.
- There is however a naming convention for private fields that is used to avoid accidents
  - The convention is to use __ (double underlines) in front of any attribute of method that should be private
  - This ***mangles*** the name to include the class, namely __var becomes _class__var
  - The fields are however still fully accessible, and the __dict__ makes it obvious

CLASSES

- Consider this example
- The Person class now has two private attribute
  - __name and __age
  - We cannot access these fields directly from outside the class

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

    def print_age(self):
        print("The age of {} is {} years old.".format(self.__name, self.__age))

def main():

    john = Person('John', 34)

    print(john.__name) # this won´t work because the field is private
main()
```

CLASSES

- This example shows that the fields aren‘t really private
  - The double underscore just changes the name of the field for the outside
  - But do note! This does not mean that we should use this approach to get the values of the private attributes
  - It is much more common to use special getter methods

```
1   class Person:
2       def __init__(self, name, age):
3           self.__name = name
4           self.__age = age
5       
6       def print_age(self):
7           print("The age of {} is {} years old.".format(self.__name, self.__age))
8   
9   def main():
10      
11      john = Person('John', 34)
12      
13      print(john._Person__name) # this will print the name attribute
14  main()
```

CLASSES

- We usually use get methods if we wan‘t to use private attributes
  - A get method is just a method that returns a private value

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

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def print_age(self):
        print("The age of {} is {} years old.".format(self.__name, self.__age))

def main():

    john = Person('John', 34)

    print(john.get_name()) # prints: John

main()
```

CLASSES

- But what if we wan‘t to change the value of a private attribute?
  - We can use set methods!
  - The benefit of using a set method is that we can set validation checks for the values that are passed to the method

CLASSES

- Consider the set_age method. It will only set the new age if it is a positive number
- This is how we can prevent the user of the class to assign invalid values to the private attributes

```
1   class Person:
2       def __init__(self, name, age):
3           self.__name = name
4           self.__age = age
5
6       def get_name(self):
7           return self.__name
8
9       def get_age(self):
10          return self.__age
11
12      def set_age(self, new_age):
13          if new_age > 0:
14              self.__age = new_age
15
16      def print_age(self):
17          print("The age of {} is {} years old.".format(self.__name, self.__age))
18
19  def main():
20
21      john = Person('John', 34)
22
23      john.set_age(24)
24
25      print(john.get_age()) # prints 24
26
27  main()
```