codecademy

Classes

Python type() function

The Python `type()` function returns the data type of the argument passed to it.

```
a = 1
print type(a) # <type 'int'>

a = 1.1
print type(a) # <type 'float'>

a = 'b'
print type(a) # <type 'str'>

a = None
print type(a) # <type 'NoneType'>
```

Python class

In Python, a class is a template for a data type. A class can be defined using the `class` keyword.

```
# Defining a class
class Animal:
  def __init__(self, name,
number_of_legs):
    self.name = name
    self.number_of_legs = number_of_legs
```

Instantiate Python Class

In Python, a class needs to be instantiated before use. As an analogy, a class can be thought of as a blueprint (Car), and an instance is an actual implementation of the blueprint (Ferrari).

```
class Car:
  "This is an empty class"
  pass

# Class Instantiation
ferrari = Car()
```

`__main__` in Python

In Python, `__main__` is an identifier used to reference the current file context. When a module is read from standard input, a script, or from an interactive prompt, its `__name__` is set equal to `__main__`. Suppose we create an instance of a class called `CoolClass`. Printing the `type()` of the instance will result in:

```
<class '__main__.CoolClass'>
```

This means that the class `CoolClass` was defined in the current script file.

Python Class Variables

In Python, class variables are defined outside of all methods and have the same value for every instance of the class.
Class variables are accessed with the `instance.variable` or `class_name.variable` syntaxes.

```
class my_class:
  class_variable = "I am a Class
Variable!"

x = my_class()
y = my_class()

print(x.class_variable) #I am a Class
Variable!
print(y.class_variable) #I am a Class
Variable!
```

Python class methods

In Python, *methods* are functions that are defined as part of a class. It is common practice that the first argument of any method that is part of a class is the actual object calling the method. This argument is usually called **self**.

```
# Dog class
class Dog:
  # Method of the class
  def bark(self):
    print("Ham-Ham")

# Create a new instance
charlie = Dog()

# Call the method
charlie.bark()
# This will output "Ham-Ham"
```

Python dir() function

In Python, the built-in `dir()` function, without any argument, returns a list of all the attributes in the current scope.

With an object as argument, `dir()` tries to return all valid object attributes.

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

  def print_name(self):
    print("Hi, I'm " + self.name)


print(dir())
# ['Employee', '__builtins__', '__doc__',
'__file__', '__name__', '__package__',
'new_employee']

print(dir(Employee))
# ['__doc__', '__init__', '__module__',
'print_name']
```

Python repr method

The Python `__repr__()` method is used to tell Python what the *string representation* of the class should be. It can only have one parameter, `self`, and it should return a string.

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

  def __repr__(self):
    return self.name

john = Employee('John')
print(john) # John
```

Python init method

In Python, the `.__init__()` method is used to initialize a newly created object. It is called every time the class is instantiated.

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

# When a class instance is created, the instance variable
# 'voice' is created and set to the input value.
cat = Animal('Meow')
print(cat.voice) # Output: Meow

dog = Animal('Woof')
print(dog.voice) # Output: Woof
```

Python Inheritance

Subclassing in Python, also known as “inheritance”, allows classes to share the same attributes and methods from a parent or superclass. Inheritance in Python can be accomplished by putting the superclass name between parentheses after the subclass or child class name.

In the example code block, the `Dog` class subclasses the `Animal` class, inheriting all of its attributes.

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

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

Yoki = Dog("Yoki", 4)
print(Yoki.name) # YOKI
print(Yoki.legs) # 4
Yoki.sound() # Woof!
```

User-defined exceptions in Python

In Python, new exceptions can be defined by creating a new class which has to be derived, either directly or indirectly, from Python’s **Exception** class.

```
class CustomError(Exception):
  pass
```

Python issubclass() Function

The Python `issubclass()` built-in function checks if the first argument is a subclass of the second argument.

In the example code block, we check that `Member` is a subclass of the `Family` class.

```
class Family:
  def type(self):
    print("Parent class")

class Member(Family):
  def type(self):
    print("Child class")

print(issubclass(Member, Family)) # True
```

Method Overriding in Python

In Python, inheritance allows for method overriding, which lets a child class change and redefine the implementation of methods already defined in its parent class.

The following example code block creates a `ParentClass` and a `ChildClass` which both define a `print_test()` method.

As the `ChildClass` inherits from the `ParentClass`, the method `print_test()` will be overridden by `ChildClass` such that it prints the word “Child” instead of “Parent”.

```
class ParentClass:
  def print_self(self):
    print("Parent")

class ChildClass(ParentClass):
  def print_self(self):
    print("Child")

child_instance = ChildClass()
child_instance.print_self() # Child
```

Super() Function in Python Inheritance

Python’s `super()` function allows a subclass to invoke its parent’s version of an overridden method.

```
class ParentClass:
  def print_test(self):
    print("Parent Method")

class ChildClass(ParentClass):
  def print_test(self):
    print("Child Method")
    # Calls the parent's version of
print_test()
    super().print_test()

child_instance = ChildClass()
child_instance.print_test()
# Output:
# Child Method
# Parent Method
```

Polymorphism in Python

When two Python classes offer the same set of methods with different implementations, the classes are *polymorphic* and are said to have the same *interface*. An interface in this sense might involve a common inherited class and a set of overridden methods. This allows using the two objects in the same way regardless of their individual types.
When a child class overrides a method of a parent class, then the type of the object determines the version of the method to be called. If the object is an instance of the child class, then the child class version of the overridden method will be called. On the other hand, if the object is an instance of the parent class, then the parent class version of the method is called.

```
class ParentClass:
  def print_self(self):
    print('A')

class ChildClass(ParentClass):
  def print_self(self):
    print('B')

obj_A = ParentClass()
obj_B = ChildClass()

obj_A.print_self() # A
obj_B.print_self() # B
```

+ Operator

In Python, the `+` operation can be defined for a user-defined class by giving that class an `.__add()__` method.

```
class A:
  def __init__(self, a):
    self.a = a
  def __add__(self, other):
    return self.a + other.a

obj1 = A(5)
obj2 = A(10)
print(obj1 + obj2) # 15
```

Dunder methods in Python

Dunder methods, which stands for “Double Under” (Underscore) methods, are special methods which have double underscores at the beginning and end of their names.
We use them to create functionality that can’t be represented as a normal method, and resemble native Python data type interactions. A few examples for dunder methods are: `__init__` , `__add__` , `__len__` , and `__iter__` .
The example code block shows a class with a definition for the `__init__` dunder method.

```
class String:
  # Dunder method to initialize object
  def __init__(self, string):
    self.string = string

string1 = String("Hello World!")
print(string1.string) # Hello World!
```