Object Oriented Programming with Python

CST 205

“Programs must be written for people to read, and only incidentally for machines to execute.”

Hal Abelson, *Structure and Interpretation of Computer Programs*

Review: Data types

A **data type** is:

- a set of values
- a set of operations that can be performed on those values

Example: The `int` data type

- The set of values of type `int` are the set of all whole numbers and their opposites.

- Operations we can perform on integers include addition and substraction, multiplication and division, modulus and exponentiation.

Class and objects

- A custom type in Python is called a **class**.
- Classes provide a means of bundling data and functionality together.
- Objects are **instances** of classes.
- Everything in Python is an object.

Exploring built-in types

- We have used built-in objects such as integers and strings.

```
a = 1
print(type(a))

b = 'hello world'
print(type(b))
print(isinstance(a, str))

c = ['one', 2, True]
print(type(c))

def my_func(a,b,c):
    return c-b*a

print(type(my_func))
```

- `type()` returns the type of an object
- `isinstance()` can be used to test the type of an object
- `dir()` can be used to list object attributes

```
s = 'cst 205'
print(dir(s))
```

Classes

- Classes are the core concept in object oriented programming.
- Classes represent **types**
- In OOP terminology, objects that belong to a type are called **instances** of the type.

Conceptual example

- We can define an `Account` class.
    - Javier's bank account is an instance of the `Account` class.
- We can create a `Dog` class.
    - Martha's pet dog is an instance of the `Dog` type.

Type hierarchies

- The `Dog` type can be a specific sub-type of the `Canine` type.
    - The `Wolf` type can be another sub-type of the `Canine` type.
- The abstraction of types and instances is central to the concept of OOP which has fundamental influences on programming languages, making code easier to understand and maintain.

Python `class` statement

- Classes are created in Python using the `class` statement.

- Classes have both a constructor `__new__()` and an intializer `__init__()`, as well as a destructor mechanism `__del__()`.

- Destructor is not guaranteed to be called.

Python's double underscores

- Underscores, "_", have a variety of important meanings in Python.

- Names that have both leading and trailing double underscores reserved for special use by Python

- Python developers usually user the term *dunder*, instead of double underscore.

  - For example, `__init__()` will usually be said as *dunder init*.

Creating a simple class

docstring

creates a new instance of the class

```
class MyClass:
    """A simple example class"""
    i = 12345

    def hello(self):
        return 'hello world'

x = MyClass()
print(x.i)
print(MyClass.__doc__)
print(MyClass.i)
print(x.hello())
```

Initializing objects

- Can define initial values or inital processing steps when a new object is instantiated.

- When Python instantiates a new object, it looks to see if there is a method called `__init__()`.

    - `__init__()` (a type of constructor) is a special method, called automatically when a new object is constructed.

- The parameter `self` refers to the object being instantiated, allowing you to interact with the object during initialization.

Let's create our own `Color` class

```
class Color:
    """A class to define RGB colors"""
    def __init__(self, name, red, green, blue):
        # instance variables unique to each instance
        self.name = name
        self.red = red
        self.green = green
        self.blue = blue


blue = Color("boring blue", 0, 0, 255)
green = Color("normal green", 0, 255, 0)

# print(isinstance(a, str))
print(blue.name)
```