Intro to Computer Science
CS-UH 1001, Spring 2022

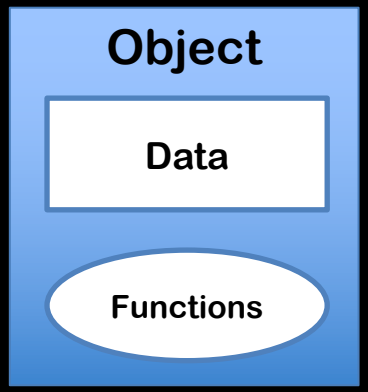
Lecture 12 – Object Oriented Programming

Object Oriented Programming

- There are two ways of programming:
  - Procedural Programming
  - Object Oriented Programming (OOP)
- Large programs are (sometimes)
  - difficult to design
  - difficult to extend and modify
  - become excessively complex
- In the real world, we often work with objects, such as departments, people, datasets, etc.

The OOP approach

- The disadvantages of procedural programming led to the development of **OOP**
- OOP does not focus on functions (procedures), but on **objects**
- OOP combines data and functions into a single unit



Main Concepts of OOP

- OOP addresses the problem of data and code separation through:
  - Abstraction / Encapsulation
  - Inheritance
    - reusing classes and modifying their behavior
  - Polymorphism
    - overloading object behavior

Object Oriented Programming

- Recall: In Python, everything is an object
- We have already seen objects and methods in Python:
  - str, int, float, list, file object, dict, etc.
  - isdigit(), append(), remove(), join(), etc.
- Objects are a data abstraction that contain
  - data attributes (object variables)
  - interfaces for interacting with the data
    - through methods (object functions)

Example: Lists

- my_list = [1,2,3,4] is an object of the data-type list
- Internally a list object is represented as a linked list (attribute) of other objects

- Example methods of the list class:
  append(), remove(), count(), etc.

Example: Tally Counter

- All objects have common behavior, but each of them can have a different state

- A tally counter can be represented as an object (class)
  - all have the same functionality (methods)
  - their instance variables (attributes) can be set to different values

Classes

- An Object is described by a class
  - A class is a “blueprint” of an object
- A class is a piece of code that describe a particular object type
  - Defines the object attributes (variables)
  - Defines the object methods (functions)
- By default, all objects are mutable

Defining a Class

- A class is defined as follows:

```
class ClassName:

    def __init__ (self):
        # initialize attributes, e.g.
        # self.attribute1 = 0

    def my_method(self):
        # method code here

    def my_method2(self, argument):
        # method code here
```

For class names the same variable naming rules apply.
However, it's a convention to start with a capital letter to distinguish variables from classes.

The self parameter is required in every method of a class.

A method operates on a specific object attribute, so when we execute a method it must know which object's data it should operate on.

The self parameter reference the specific object that the method should operate on.

The __init__() method is a special method in classes

Class Methods

- Methods are function definitions that belong to a class
  - A class can have as many methods as needed
- Methods are usually used to modify class attributes or execute tasks specific to the class
- Like functions, methods can take any number of arguments
  - however, the first argument is always self!
- Methods can be void or fruitful
- For example:

```
def append(self, value):
    # code for appending the
    # value to the list object
```

The Class Constructor __init__()

- The __init__(self) method is a special method in classes, called constructor or initializer
- The constructor is responsible for defining and initializing all of the instance variables (attributes) that are to be contained in the object
- The constructor is automatically called whenever an object (class) is created (instantiated)
  - A class can only have one constructor
- Like functions, the constructor can have multiple arguments that can be used to initialize attributes of a class

Class Attributes

- Class attributes (instance variables) are variables describing the state of an object
  - they are bound to an instance of a class
- Attributes are typically created in the constructor
- Attributes are declared and accessed using the self reference
- For example:

```
def __init__(self):
    self.value = 0 # creating an attribute
```

The self Reference

- self is a reference to the instance of the class

- By using the self reference, class attributes and class methods can be created, accessed or modified from within the class

Example: The self Reference

- Remember, methods are like functions!

```
class SimpleClass:

    def __init__(self):
        value = 1

    def my_method(self):
        print(value)
```

✖ Error

Local variable!!

```
class SimpleClass:

    def __init__(self):
        self.value = 1

    def my_method(self):
        print(self.value)
```

Reference to the instance of the class

Instantiating a Class

- The process of creating an object from a class is called instantiation
- The object created from a class is an instance of the class
- Instantiating an object from a class:

```
my_object = SimpleClass()
```

Class name

Variable holding an instance of the class, i.e. object

Instantiating a Class

```
my_object = SimpleClass()
```

Once a class is instantiated, all attributes and methods can be used from outside the class

For example:

```
my_object.my_method() # call method
my_object.value # access attribute
```

Initializing Class's Attributes

An object can be initialized with custom values by passing arguments to the __init__() method.

```
class SimpleClass:

    def __init__ (self, argument):
            self.value = argument

my_object = SimpleClass(7)
```

Remember: Arguments can have default values!

Instantiating Objects of Built-in Data Types

- Remember, everything in Python is an object!

Python shortcut	OOP way
my_int = 100	

Instantiating Objects of Existing Data Types

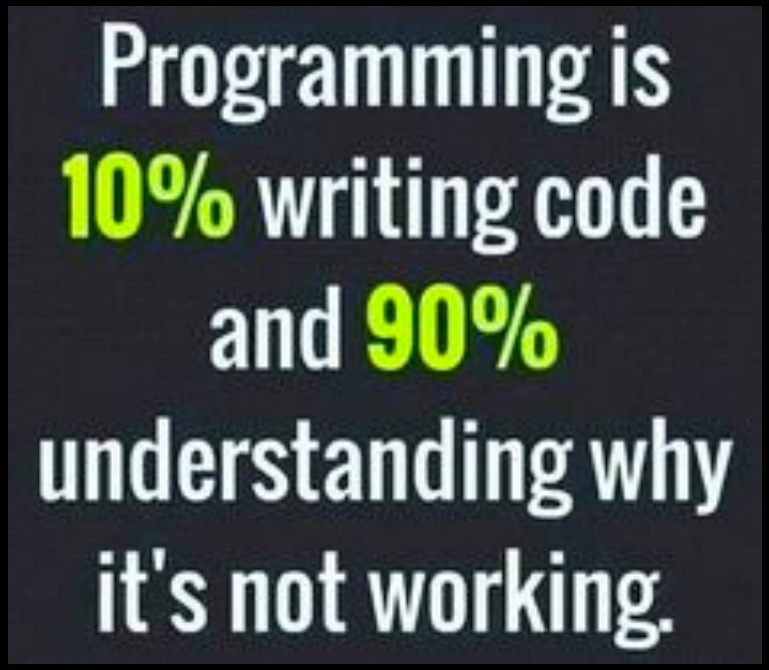
- Remember, everything in Python is an object!

Python shortcut	OOP way
my_int = 100	my_int = int(100)
my_float = 2.5	my_float = float(2.5)
my_string = “Hello World”	my_string = str(“Hello World”)
my_list = []	my_list = list()
my_dict = {}	my_dict = dict()

How about: my_int = int() ?

Example: Tally Counter (ex_12.1.py)

- Let's create a class that models a tally counter
- The class consists of
  - 3 attributes (class variables):
    counter: holds the current counter value
    max_value: maximum value the counter can take
    color: color of the tally counter
  - 2 methods (class functions):
    - click(): increments the value of the counter
    - reset(): resets the value of the counter to 0


Programming is
10% writing code
and 90%
understanding why
it's not working.

Hands-on Session

Coin class

Coin Class (ex_12.2.py)

Create a Coin object to simulate a coin toss in OOP:

- Create a Coin class
- The Coin class has only one attribute (side_up), which holds either “Heads” or ”Tails”
- The Coin class has one method:
  - toss(self)
    - use random.randint() to simulate a coin toss whenever the toss() method is called

Encapsulation/Abstraction

- An important advantage of OOP is encapsulation/abstraction of data
- A mechanism for restricting/hiding the access to some object data/methods
  - private and public attributes

Public Attributes

- In the previous example, the side_up attribute is public:
  - print(coin.side_up) prints the coin side
  - coin.side_up = 'No face' overwrites the attribute
- When setting a class attribute (variable) to public, there is no control which values the attribute can take
  - Any external code will be able to change it without any constraints
  - Not protected against corruption

Private Attributes

- When defining an attribute in the __init__ method, an attribute can be made private by adding two underscores before the attribute name

```
def __init__ (self):
    self.__side_up = 'Heads'
```

- Now, the attribute is called __side_up and is private
  - Can't be accessed from outside
  - Can't be modified from outside (only internally)

How to access private attributes from outside?

Getter and Setter Method

- Getter and setter methods provide access to private attributes (variables)

- By using getter and setter methods, the programmer can control how their attributes are accessed and updated in the proper manner
  - such as changing the value of a variable within a specified range

Coin Class (ex_12.2.py)

- Extend the Coin class:
  1. make the side_up attribute private
  2. add getter and setter methods to get and set the side_up attribute
     - only Heads and Tails are valid values

Bank Account (ex_12.3.py)

Create a BankAccount class, where objects created from the BankAccount class simulate a bank account

Attributes:

- Balance (should be private)

Methods:

- Deposit money
- Withdraw money (Make sure there is sufficient balance in the account)
- Print the balance

Extra task: Add more attributes to the class, e.g. name, account number, etc