FIT9136: Algorithms and programming foundations in Python

Week 5: Classes & Variable Scope

Agenda

Week 5 Synopsis

- **Week 5 is aimed to provide you with:**
    - Concepts of **Object Oriented Programming** which includes:
        - Objects
        - Classes
        - Methods
        - Variable Scopes
        - Inheritance

Learning Objective

- Understand & Recognize the concept of object-oriented programming (OOP) & Classes
- Understand the concept of variable scope and lifetime in Python.

What is Object-Oriented Programming (OOP)?

Object-Oriented Programming

- Object-oriented programming (OOP):
    - It is a **method** of **structuring a program** by bundling **related properties and behaviors** into individual objects.
    - Conceptualise a **real-world** scenario as to how **multiple groups** of **objects** interact to build an **application**
    - Each type of objects represents one specific kind of concept in the real world
- Fundamental concepts of OOP:
    - Creation of objects
    - Encapsulate both the attributes and the behaviours of the objects (i.e. the ways how objects interact with each other)

Advantages of OOP

- Divide-and-conquer development:
    - Implement and test behaviour of each class separately
    - Increased modularity reduces complexity
- Easy reuse of code
    - Many python modules define new classes
    - Each class has a separate environment (no collision on function names)
    - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

Dis-advantages of OOP:

- Steep Learning Curve:
    - Thought process of OOP might not be normal for everyone.
- Larger Program Size:
    - OOP involves more line of code then procedural programming, which can lead to slower programs
- Not suitable for all types of problem:
    - Problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

Objects

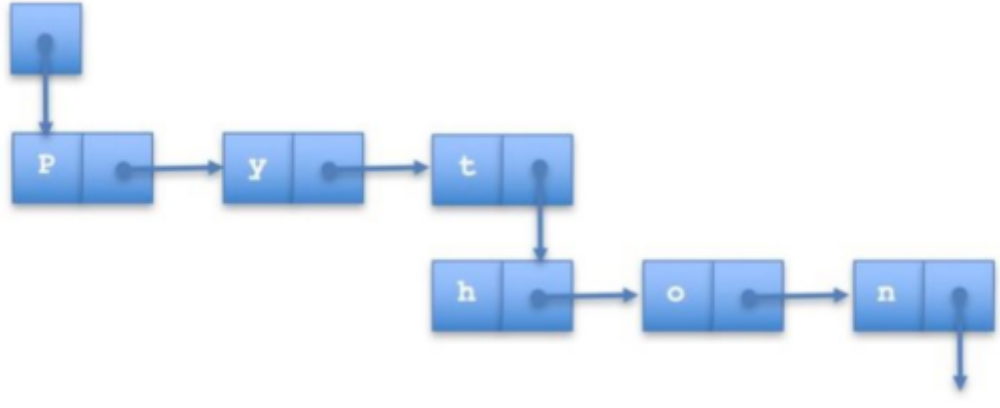
What is Object?

- **Object is a data abstraction including:**
    - An internal representation
        - through data attributes
    - An interface for interacting with object
        - by methods (functions)

Example of objects

a_list = [“p”, “y”, “t”, “h”, “o”, “n”]

- An internal representation through data attributes (linked list of cells)



- an interface for interacting with object
    - [i], [i:j]
    - .append(), .remove()
    - len(), min()

Objects (continue)

Every object has:

- A type
- An internal data representation (primitive/composite)
- A set of procdures for interacting with the object

An object is an instance of type

- 1234 is an instance of an int
- "Python" is an instance of an str

Objects (continue)

Everything in python is object with a type:

- Can create new objects of some type
- Can manipulate objects
- Can destroy objects
    - Explicitly using del or just “forget” about them
    - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”

Classes

Classes

- **Definition of classes:**
  - Designed to represent only one concept within an application
  - Defined as a template ("blue-print") to create objects of a specific type
  - Multiple classes are integrated to build a complete application
- **Instances of a class:**
  - Each instance is assigned to a variable name (reference) to access its internal data values and the associated methods
  - **Each class defines a set of "instance variables" (data values to be represented for each object) and a set of "methods" (operations) that can be applied on the objects.**

Class Implementation

- **Class header:**
    - Starts with the keyword class and followed by a class name
    - Naming convention: CapWords
- **Example: The Point class**

In [ ]:
```
# class is keyword to define the class
# point is class name
# pass is keyword that is used as placeholder for statments
class point:
    #defining attributes
    pass
```

Class Implementation (continue)

- **Constructor:**
    - The essential method for object creation
    - Initialise the values of the instance variables of each object
    - Invoked based on the class name
    - Instance variables: x and y values for representing each point in a two-dimensional space

In [ ]:

```
"""
def -> keyword for defining function/method
__ -> double under-score represnt the special method
__init__ -> used as constructor
self -> refers to own object
self.x(y) -> two points
"""
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Using Class

- Data attributes of an instance are called instance variables

In [ ]:
```
# created point class
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# creating instance of point class
p = Point(1,2)
# printing values of x,y
print(p.x)
print(p.y)
```

Class Implementation

- **Method:**
    - Procedural attribute, like a function, but works only with this class
    - Operations to interact with the class
    - Invoked based on the class name

In [ ]:

```python
# created point class
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # making distance method
    def distance(self, other):
        x_diff = (self.x - other.x)**2
        y_diff = (self.y - other.y)**2
        distance = (x_diff+y_diff)**0.5
        return distance
```

Using Class

- **Using Class Method**
    - Conventional way:
        - Using object to call method and pass another object as argument
    - Equivalent to:
        - Using class and passing both object to methods as argument

In [ ]:

```
# Conventional way

# creating instances of Point class
p = Point(1,2)
q = Point(2,3)
# printing distance value using object.method(object)
print(p.distance(q))

# Equivalent to:

# printing distance using classname.method(object, object)
print(Point.distance(p, q))
```

Using Class (continue)

- **Print representation of an object**
- Define a **str** method for a class for nice printing
- Python automatically calls the **str** method when used with print() on the class object

In [ ]:
```
# printing "p" object of class Point
print(p)
```

Using the Class (Continue)

- **Define your own print method**

In [ ]:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff = (self.x - other.x) ** 2
        y_diff = (self.y - other.y) ** 2
        distance = (x_diff + y_diff) ** 0.5
        return distance

    # name of special method "__"
    def __str__(self):
        # must return string
        return "<" + str(self.x) + ", " + str(self.y) + ">"

p = Point(1,2)
print(p)
```

Class Implementation (Continue)

- **SPECIAL OPERATORS**
    - +, -, ==, len(), print, and many others
    - define them with double underscores before/after
        - **add**(self, other)-> self + other
        - **sub**(self, other)-> self - other
        - **eq**(self, other) -> self == other
        - **lt**(self, other) -> self < other
        - **len**(self) -> len(self)
        - **str**(self) -> print(self)
        - ... and others
    - https://docs.python.org/3/reference/datamodel.html#basic-customization (https://docs.python.org/3/reference/datamodel.html#basic-customization)

Class Implementation (Continue)

- **Self:**
  - Each method defined within the class must have **self** as the first argument
  - No need to be specified during method invocation
  - Automatically set to reference the object on which the method is invoked

In [ ]:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    # getter/Accessor methods
    def get_x(self):
        return self.x
    def get_y(self):
        return self.y

    # setter/Mutators method
    def set_x(self, x = 0):
        self.x = x
    def set_y(self, y = 0):
        self.y = y
```

(More on) Class Implementation

- Show the Point.py file

Object Instantiation

- To use a class for object creation in another program:
- Must first import the class:
    - from import
- **To construct a new object:**
    - Syntax: **object_name = ClassName(arg1, arg2, ...)**
    - E.g.:
        - **a_point = Point(1,0)**
        - **a_point = Point()**
    - Note that **self** is not passed as an argument

In [ ]:
```
from Point import Point
point1 = Point()
point2 = Point(1,2)

print(point1.get_x())
print(point1.get_y())
print(point2.get_x())
print(point2.get_y())
```

Variable Scope

Variable Scoope

- **Scoping:**
    - Define the part of the program where a variable is accessible
- **Lifetime:**
    - Define the duration for which a variable exists during the program execution
- **Global variables:**
    - Can be accessed throughout the entire program
    - Exists until the execution of the program terminated
- **Local variables:**
    - Can only be accessed within the function it was defined
    - Exists until the function exists

Variable Scoping & Lifetime in Function

In [ ]:

```
# defining function
def f(x):
    # variable "x" has local scope / scope with in function
    x = x + 1
    print ("in f(x): x =", x)
    return x

# varable "x" & "z" have global scope
x = 3
z = f(x)
```

Variable scoping & Lifetime in Class

- **Instance variables:**
    - Associated to individual objects and are unique to each other
    - Local to the class and cannot be accessed outside of the class
- **Class variables:**
    - Define outside the body of any methods in a class
    - Global in scope and can be accessed both inside and outside of the class

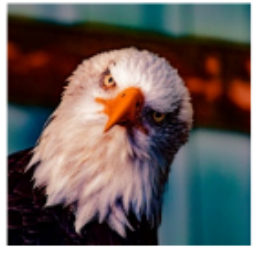
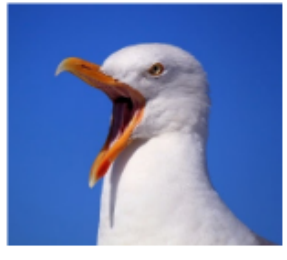
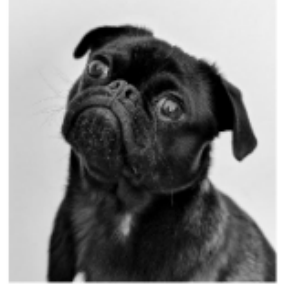
In [ ]:

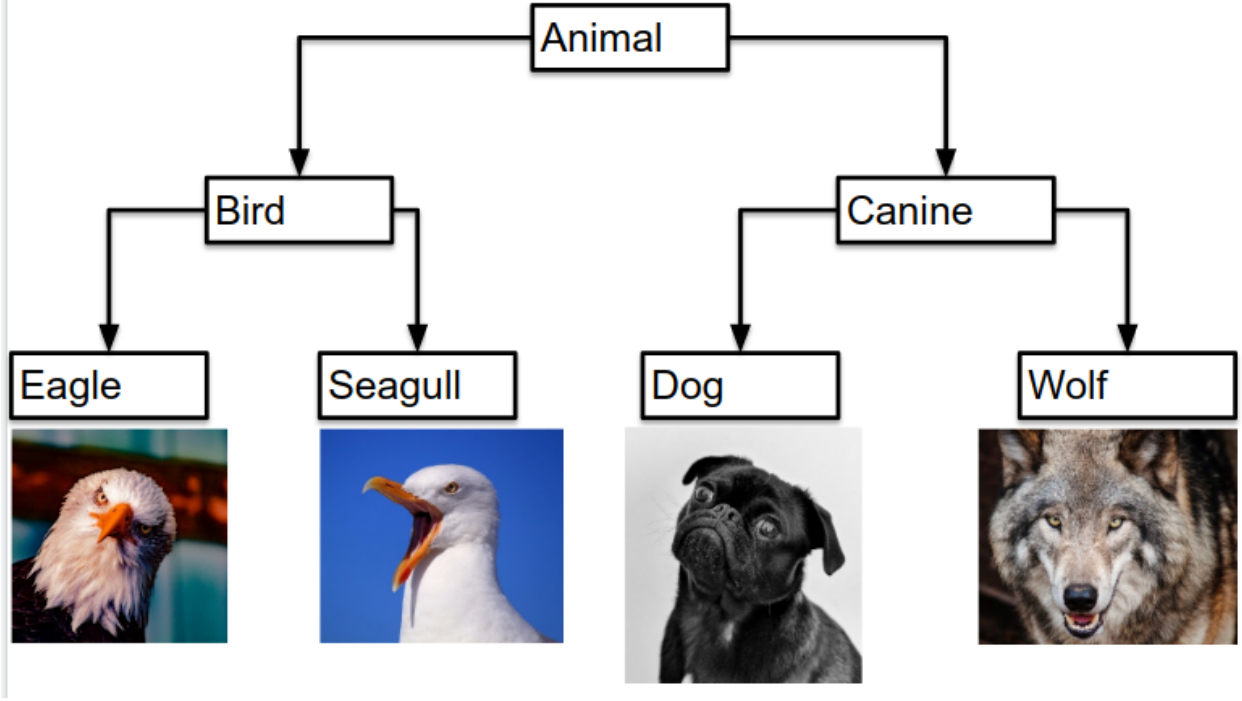
```
# defining Point class
class Point:
    # class variable
    x = 0
    y = 0

    def __init__(self, x, y): # "x" & "y" local variable
        # "self.x" & "self.y" are instance varibale
        self.x = x
        self.y = y
```

Inheritance

Inheritance

- Allows for aspects of an object to be passed on to a new object being defined.
- Very useful when new attributes or methods need to be added to the class, but the original class needs to remain the same.
- E.g. Animals

parent class
(superclass)

child class
(superclass)

- Inherit all data and behaviors of parent class
- Add more info
- Add more behaviour
- Override behavior

Example of Inheritance

- **A parent class Animal**

In [ ]:
```
class Animal:  # defining Animal Class
    def __init__(self, name="animal", age=0): # Creating constructor for Animal
 class
        # assigning name and age to class variable
        self.name = name
        self.age = age
    # abstract methods defining
    def eat(self):
        pass

    def drink(self):
        pass

    def poop(self):
        passs
```

Implicit init

In [ ]:

```
class Canine(Animal):

    def eat(self):
        pass

    def drink(self):
        pass

    def poop(self):
        pass

    def scream(self):
        pass

    def run(self):
        pass

    def walk(self):
        pass
```

init has not be defined. What happens if we create:

dog = Canine(“Doge”,2)?

In [ ]:
```
dog = Canine("Doge",2)
```

Overriding init

- **init** from the parent class is called.
- What if we want to add new attributes to our class? Redefine **init**.

In [ ]:

```
class Canine(Animal):

    def __init__(self, name="animal", age=0, fur_colour = "brown"):
        self.fur_colour = fur_colour # add the new attribute

        Animal.__init__(self) # call Animal's init, passing the Canine as "self"

    def eat(self):
        pass

    def drink(self):
        pass

    def poop(self):
        pass

    def scream(self):
        pass

    def run(self):
        pass

    def walk(self):
        pass
```

super()

- **Why should you care about super()?**
    - Sometimes you are writing a class that will inherit but the parent's name is unknown.
    - Super will call every parent's version of the function.
- Classes can inherit from **multiple parents**. **super()** allows one line of code to invoke **EVERY parent's functions**.

In [ ]:

```
class Canine(Animal):

    def __init__(self, name="animal", age=0, fur_colour = "brown"):
        self.fur_colour = fur_colour # add the new attribute

        super().__init__(self) # call Animal’s init using super)(, passing the C
anine as “self”

    def eat(self):
        pass

    def drink(self):
        pass

    def poop(self):
        pass

    def scream(self):
        pass

    def run(self):
        pass

    def walk(self):
        pass
```

Example of Inheritance

- **Multiple inheritance** can be used to allow for **classes** to be put together like **they are components**.
- Using Animals as an example, there might be a “Fly” class, and a “Swim” class.
- A Pelican could then inherit both of these to give it all the attributes and functions it needs to Fly, and the same for Swim
- E.g. Pelican(Fly, Swim)

Summary

This session we covered topics:

* Object-Oriented Programmin
* Classes, Objects, & Methods
* Variable Scoping
* Inheritance

Practise Questions

Create a Dog class, which has constructor asking name & age of the dog, it's own print method printing name and age of the dog, and has method call speak with accepts sound of the dog,

In [ ]:
```
"""
The python script creates Class Dog with constructor, print method, and speak me
thod.
"""

# Define class with name "Dog"
class Dog:

    # defining constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # defining print method


    # defining speak method
```

Solution

In [ ]:

```
"""
The python script creates Class Dog with constructor, print method, and speak me
thod.
"""

# Define class with name "Dog"
class Dog:

    # defining constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # defining print method
    def __str__(self):
        return f"{self.name} is {self.age} years old"

    # defining speak method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Create a GoldenRetriever & GermanShepherd class that inherits from the Dog class. Give the sound argument of GoldenRetriever.speak() a default value of "Bark", and GoldenRetriever.speak() a default value of "Howl". Use the above code for parent Dog class

In [ ]:

```
"""
This python scripts creates class GoldenRetriever & GermanShepherd, using parent
class as Dog. Overwriting speak methhod for both
classes and default values of both class are "bark" & "howl" respectively.
"""
# defining GlodenRetriever Class
class GoldenRetriever(Dog):
    # Overwriting speak method

        # calling parent class using super()


# defining GermanShepherd Class
class GermanShepherd():
    pass
    # Overwriting speak method

        # calling parent class using Dog
```

Solution

In [ ]:

```
"""
This python scripts creates class GoldenRetriever & GermanShepherd, using parent
class as Dog. Overwriting speak methhod for both
classes and default values of both class are "bark" & "howl" respectively.
"""
# defining GlodenRetriever Class
class GoldenRetriever(Dog):
    # Overwriting speak method
    def speak(self, sound="Bark"):
        # calling parent class using super()
        return super().speak(sound)

# defining GermanShepherd Class
class GermanShepherd(Dog):
    # Overwriting speak method
    def speak(self, sound="Howl"):
        # calling parent class using Dog
        return Dog.speak(sound)
```

Write a Python program to convert an integer to a roman numeral.

In [ ]:
```
"""
This python script converts numerical value to Roman letter by Class NumeralToRo
man and method called int_to_roman, which accepts num(number) as argument.
"""
#defining Class NumeralToRoman
class NumeralToRoman:
    # defining method int_to_roman(self, num)
    def int_to_Roman(self, num):
        # define mapping list for numeral to roman letter
        val = [] # write number eqivalent to roman
        syb = [] # write roman eqivalent tonumbers
        roman_num = ''

        i = 0
        # Using while loop to check number is greater than 0
        while  num > 0:
            # using for loop to iterate over numbers
            for _ in range(num // val[i]):
                # start your logic

            #end your logic

        # return roman num string
        return roman_num

# Calling int_to_roman method using class name
print(NumeralToRoman().int_to_Roman(1))
print(NumeralToRoman().int_to_Roman(4000))
```

Solution

In [ ]:

```
"""
This python script converts numerical value to Roman letter by Class NumeralToRo
man and method called int_to_roman, which accepts num(number) as argument.
"""
#defining Class NumeralToRoman
class NumeralToRoman:
    # defining method int_to_roman(self, num)
    def int_to_Roman(self, num):
        # define mapping list for numeral to roman letter
        val = [1000, 900, 500, 400,100, 90, 50, 40, 10, 9, 5, 4, 1]
        syb = ["M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV"
, "I"]
        roman_num = ''

        i = 0
        # Using while loop to check number is greater than 0
        while  num > 0:
            # using for loop to iterate over numbers
            for _ in range(num // val[i]):
                # start your logic
                roman_num += syb[i]
                num -= val[i]
            i += 1
            #end your logic

        # return roman num string
        return roman_num

# Calling int_to_roman method using class name
print(NumeralToRoman().int_to_Roman(1))
print(NumeralToRoman().int_to_Roman(4000))
```

Write a Python program to get all possible unique subsets from a set of distinct integers.

In [ ]:

```
"""
This python Script gives us unique subsets from a list of numbers
"""
# defining UniqueSubset class
class UniqueSubset:
    # defining method sub_set (self, sset)
    def sub_sets(self, sset):
        # return subset (sorted) and current set (which will be empty)

    # defining method subsetRecur (self, current, sset)
    def subsetsRecur(self, current, sset):
        # check if the sset (subset) is empty
        if sset:
            pass
            # if subset is not empty, calling subsetRecur method 2 times to divi
de set till atomic value.

        # return list of current
        return [current]

print(UniqueSubset().sub_sets([4,5,6]))
```

Solution

In [ ]:

```
"""
This python Script gives us unique subsets from a list of numbers
"""
# defining UniqueSubset class
class UniqueSubset:
    # defining method sub_set (self, sset)
    def sub_sets(self, sset):
        # return subset (sorted) and current set (which will be empty)
        return self.subsetsRecur([], sorted(sset))

    # defining method subsetRecur (self, current, sset)
    def subsetsRecur(self, current, sset):
        # check if the sset (subset) is empty
        if sset:
            # if subset is not empty, calling subsetRecur method 2 times to divi
de set till atomic value.
            return self.subsetsRecur(current, sset[1:]) + self.subsetsRecur(curr
ent + [sset[0]], sset[1:])
        # return list of current
        return [current]

print(UniqueSubset().sub_sets([4,5,6]))
```

To do:

- **How much Assessment one have you completed?**

Thank You :)