Python III
Objects

~/IFT383/module-7

Introduction

- Python is an object-oriented language
  - The data types we have used are examples of objects
- An object defines the state of something and how we can interact with it
  - list
    - data is stored in indexes
    - list has functions that allow us to manipulate that data
      - append(), reverse()
  - string (str)
    - Stores data as a list of characters
    - rstrip(), isalpha(), isnum()

Class

- a **Class** is a blueprint used to create objects
- A class contains
  - properties
    - hold the state of our object
    - take the form of variables
  - actions
    - define how we can interract with an object
    - take the form of functions
- We use **instantiation** to create an object **instance** from a Class

Defining a class

- Parts of a class
  - name
  - data fields
  - methods
- Conventions
  - Class names start with an uppercase letter
    - MyClass, SodaCan, PickupTruck, Sadan
  - data field and function names start with a lowercase letter
    - Some people use cammelCase, others all lowercase
    - myProperty, myFunction(), myproperty, myfunction()

Class definition syntax

```
# legacy-style definition
class Computer:
    # Data fields
    memory=2048
    cores=4
    power=False

    def __init__:
        pass

    # methods
    def powerOn(self):
        self.power=True
```

```
# modern definition
class Computer(object):
    # Data fields
    memory=2048
    cores=4
    power=False

    def __init__:
        pass

    # methods
    def powerOn(self):
        self.power=True
```

Creating objects

- Once a class is defined, it can be used to create objects using a constructor
  - Creates an object in memory according to the class definition
  - uses the __init__ function to set up the new object
- Creating on abject
  - Without any arguments
    - myComputer = Computer()
  - With arguments
    - myComputer = Computer(4096, 2)
    - Arguments are passed to the __init__ function of the class

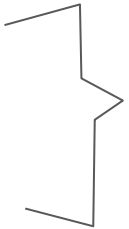
Class instantiation (Computer.py)

```
#!/usr/bin/python
class Computer(object):

    # Data fields
    memory=2048
    cores=4
    wireless=False
    power=False

    def __init__(self, newMemory=None, newCores=None):
        if type(newMemory) is int:
            self.memory = newMemory
        if type(newCores) is int:
            self.cores = newCores
        self.power=False

    def powerOn(self):
        self.power=True
```



class variables

class method

Another class example (Pizza.py)

```
#!/usr/bin/python
class Pizza(object):
    slices=16
    temperature=75.0

    def __init__(self, slices=16, temperature=75.0):
        self.slices = slices
        self.temperature = temperature

    def hasSlices(self):
        if self.slices > 0:
            return True
        else:
            return False

    def eat(self, slices=1):
        if self.slices >= slices:
            self.slices -= slices
            return True
        else:
            return False

if __name__ == "__main__":
    myPizza = Pizza(8)
    while myPizza.hasSlices():
        print("Eating pizza...")
        myPizza.eat()
```

Variable Scope

- In the previous module; we say how global and local variable scope differs
- We now also have **instance** and **class** scopes
  - Instance
    - Defined within the methods of a class
    - available to instances of that object that call the method
  - Class variables
    - Defined as part of the class definition
    - Available to all instances of that class

Class vs Instance vars (SupremePizza.py)

```
#!/usr/bin/python
class Pizza(object):
    slices=16
    temperature=75.0

    def __init__(self, slices=16, temperature=75.0):
        self.slices = slices
        self.temperature = temperature
        self.toppings = 'all'
        self.crust = 'thin'

    def hasSlices(self):
        if self.slices > 0:
            return True
        else:
            return False
[...]
```

Class variables

Instance variables

Encapsulation

- In the previous examples; the programmer could manually modify the slices variable
  - This is considered a bad practice in Object-Oriented Programming
  - The functions in our class cannot reliably predict what the slices var will be
    - Example; the slices variable could be changed to a string
- Solution: encapsulation
  - Protect the variable from being modified outside the class
  - Provide functions that control how the variable can be accessed
  - This is the basis of encapsulation

Visibility modifiers

- Visibility modifiers in Python can be used to prevent direct access to class and instance variables
- **NOTE**: In Python, they are more conventional than technical; not a guarantee
- Public
  - Intended to be accessed and modified from outside of the object
    - Example; myPizza.slices = “OM NOM NOM!”
- Private
  - Intended to only be accessed by functions within the class itself
- Protected
  - Intended to only be accessed by functions within the class definition and functions of child classes
  - More on child classes when we cover inheritance

Visibility modifier naming convention

- Public
  - name starts with an uppercase or lowercase letter
- private
  - name starts with two underscores;
    - Example __myVariable
- Protected
  - name starts with a single underscore
    - Example: _myVariable

Providing access to data

- For class variables marked as protected or private; access is provided via methods
- Accessor
  - Returns the current value of the class variable, but does not modify it
  - Names typically follow the pattern of; getVariableName()
  - Example; getSlices()
- Mutator
  - Provide a mechanism for updating class variables
  - Naming convention; setVariableName(newValue)
  - Example; setSlices(24)

Visibility modifiers (ProtectedPizza.py)

```
#!/usr/bin/python
class ProtectedPizza(object):
    __slices=16
    __temperature=75.0

    def __init__(self, slices=16, temperature=75.0):
        self.__slices = slices
        self.__temperature = temperature

    def hasSlices(self):
        if self.__slices > 0:
            return True
        else:
            return False

    def getSlices(self):
        return self.__slices

    def setSlices(self, newVal):
        if type(newVal) is int:
            self.__slices = newVal;
```

"private" class variables

Accessor

Mutator

Inheritance

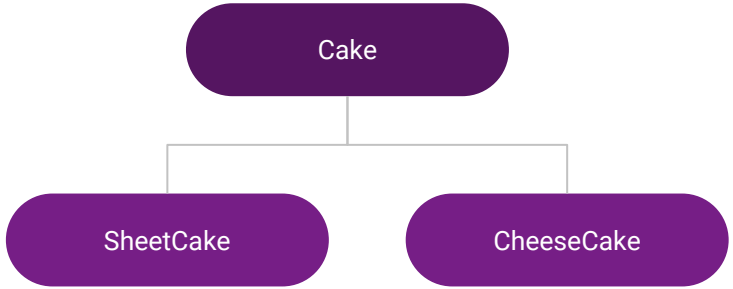
Inheritance

- A foundational concept of Object-Oriented Programming (OOP)
- A mechanism for adding functionality to a parent class by creating a child class
- Child classes
  - inherit methods and properties from their parent class
  - Can implement additional functions and properties specific to the child class
  - Can override (supercede) elements inherited from their parent class
- Programmers sometimes use the shorthand ‘is-a’ to describe inheritance
  - a PickupTruck is-a Vehicle
  - a SupremePizza is-a Pizza

Designing for inheritance

What properties and functions are common to all types of cake?
- calories
- sugar
- eat()

- What properties or functions are unique to the children?
  - sheet cake
    - length, width
  - Cheesecake
    - circumference



Inheritance example (Cake.py)

```
#!/usr/bin/python
class Cake(object):
    _calories = 0
    def __init__(self, calories=5000):
        self._calories = calories

    def getCalories(self):
        return self._calories

class SheetCake(Cake):
    length=0
    width=0

class CheeseCake(Cake):
    radius=0
```

Overriding methods

- Child classes can override the methods/functions of parent classes
- This replaces the behavior of the parent function with that of the child
- Any parent function can be overridden; including __init__

```
class Cake(object):
    _calories = 0
    def __init__(self, calories=5000):
        self._calories = calories

    def getCalories(self):
        return self._calories

class SheetCake(Cake):
    length=0
    width=0
    def __init__(self, calories=5000, length=24, width=12):
        self._calories = calories
        self.length = length
        self.width = width
```

Operator Overloading

- How operators interact with an object is defined on the ‘object’ class
  - Recall from the Cake example; Cake was a child of ‘object’
- We can *overload* these to define the behavior between an object and an operator
- Addition
  - __add__(self, other) # where other is the other object being added
- Subtraction
  - __sub__(self, other)
- Multiplication
  - __mul__(self, other)
- Division
  - __truediv__(self, other)

Even more operators!

- <. <=, ==, !=, >, >=
  - __lt__. __le__, __eq__, __ne__, __gt__, __ge__
  - All take (self, other) as arguments
- [index]
  - If your object is accessed like a list; you can perform an action here too!
  - __getitem__(self, index)
- len()
  - Your object is used with the len() function
  - __len__(self)
- str()
  - Your object is being converted to a string
  - __str__(self)

Operator overloading (Cake.py)

```
class Cake(object):
    _calories = 0
    def __init__(self, calories=5000):
        self._calories = calories

    def getCalories(self):
        return self._calories

    def __str__(self):
        return "I am a delicous cake consisting of %d calories!" % (self._calories)

    def __eq__(self, other):
        return self._calories == other._calories

    def __add__(self, other):
        return Cake(self._calories + other._calories)
```