ECE 364
Software Engineering Tools Laboratory

Lecture 7

Python: Object Oriented Programming

Lecture Summary

- Object Oriented Programming Concepts
- Object Oriented Programming in Python

Object Oriented Programming

- OOP is a programming style that emphasizes interaction between objects

- Objects represent things in the universe
  - Car, Plane, Phone, TV, Computer etc.

- Objects can represent abstract things also
  - Mathematical system, tree (data structure), file stream/network channel, web page, etc.

Composition

- Objects can be composed of other objects
  - Ex: Car (Engine, Transmission, Wheels etc.)
  - Ex: GUI (Window, Text Box, Button etc.)
  - Ex: Operating System (Process Scheduler, File System, Memory Manager etc.)

- We call this the HAS-A relationship
  - Car HAS-A engine
  - Bank Account HAS-A Balance

Encapsulation

- Objects hide their complex behaviour from the outside world by exposing only a small set of functions and properties
  - We call this encapsulation

- Example: Car
  - Complicated actions take place when a car is started
  - But with no (or limited) knowledge of the internals of the car you can change the state

Member Variables

- Objects have state (most of the time)
  - State is stored in member variables
  - A member variable is similar to a field in a C structure
  - Example: Persons age, particle mass, account balance, read position in file
- Member variables can also store complex objects
  - This is composition
- Object state can be changed by modifying member variables directly or by invoking a function

Member Functions

- Objects have functions (most of the time)
  - Called member functions
  - A member function belongs to an object
  - When called it has access to the internal state of an object

- Member functions do not necessarily affect the state of an object
  - Example: `ListVar.pop()`
  - If the list `ListVar` is empty nothing changes

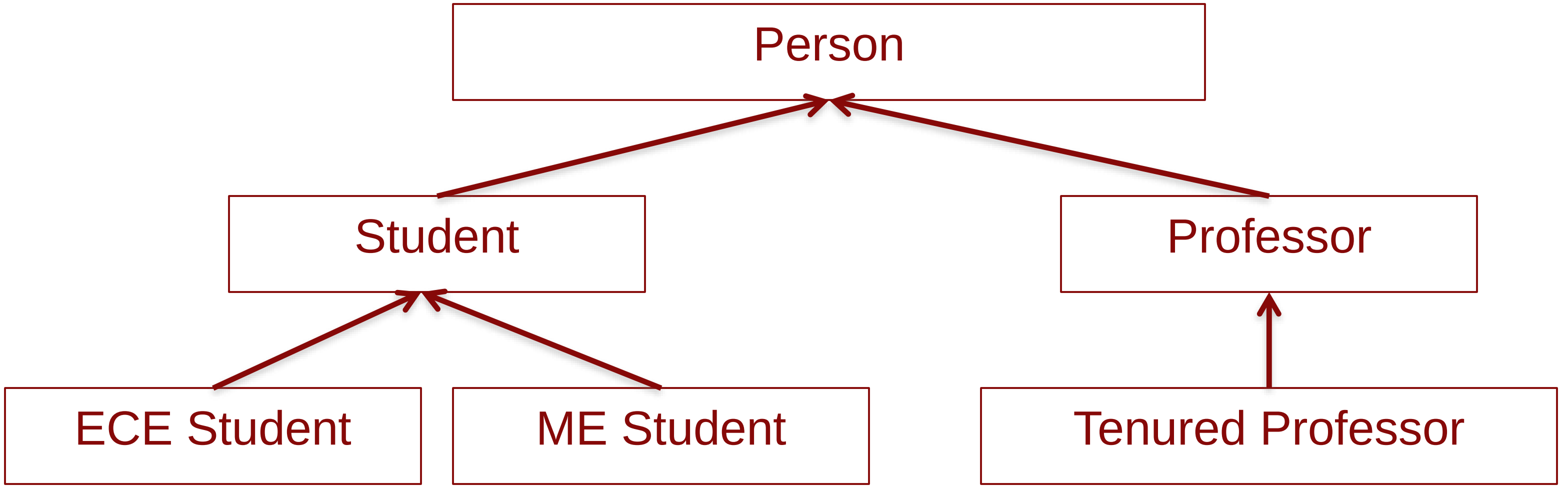
Inheritance

- Objects can inherit properties and functions from other objects
  - We call this inheritance
  - Inheritance expresses the IS-A relationship

- A derived object is an object that inherits from one or more base objects
  - Student is derived from Person
  - Student inherits from Person
  - Student IS-A Person

Inheritance (2)

- Inheritance expresses a hierarchy of IS-A relationships
  - Directed edge indicates IS-A



Inheritance (3)

- The “direction” of inheritance is strictly upwards towards the parent
  - ECE Student IS-A Student
  - ECE Student IS-A Person

- The IS-A relationship does NOT hold across or downwards
  - Can NOT say ECE Student IS-A Professor
  - Can NOT say Person IS-A Student (Not All Are)

Function Overriding

- When a derived object inherits from a base object it can choose to keep the original behavior of a member function or implement different behavior

- Function overriding enables a derived class to replace or enhance the behavior of a function of the same name from its parent class

Polymorphism

- An object can behave like (be treated the same as) a different object if both objects implement the same interface
  - We call this polymorphism

- An interface is a well defined set of functions and attributes that are implemented by objects

- A derived object can be treated as if it were the same as its base object without having to know what the specific object type is ahead of time

Polymorphism (2)

- An easier way to view polymorphism:
- Consider various kinds of Students: GoodStudent, AvgStudent, BadStudent
- All Student objects implemented a Study() function but each type varies in behavior

Polymorphism (3)

- `for Student s in Class.getStudents():`
- `  s.Study()`

- From the viewpoint of a Student we can call the correct Study() function without knowing the specific type of student ahead of time

Function Overloading

- Function overloading allows the definition of multiple functions with the same name but different arguments
  - Reduces the number of different function names
  - Avoids creating function names that encode the arguments (e.g. `print_2float, print_1int`)

- Example:
  - `print(s)    # s is a string`
  - `print(i)    # i is an integer`
  - `print(r)    # r is a float`

Operator Overloading

- Operator overloading is a feature of many object oriented languages that allow the functionality of built-in operators to apply to programmer defined objects

- Example: Matrix Object
- `M3 = M1 * M2` vs. `M3 = M1.multiply(M2)`

- Poorly designed operator semantics can lead to confusing behavior (e.g. + performs *)
  - Need to consider mutability of operands also!

Classes

- A class is the definition of an object
  - A class specifies member functions and member variables that belong to an object
- An object is an instance of a class
  - Creating a new object is called instantiation
  - A class can have many instances

Constructors

- A constructor is a special member function that is called to instantiate a class
- The constructor is only called once during the lifetime of an object
- The constructor is responsible for initializing the state (member variables) of an object
  - May also invoke constructors of other objects

Destructors

- A destructor is a special member function that is called right before an object goes out of existence or is explicitly de-allocated
- Destructors are used to release resources or finalize the object
  - Objects may have open files or network streams that must be closed
  - Can also be used to notify other objects about destruction

OOP in Python

- Almost everything in Python is an object
  - Numbers, strings, list, dictionary, tuple, etc.
  - File streams, network sockets, GUI elements etc.

- Up to this point you have only made use of existing objects and their functions
  - Now you will learn how to extend or create new objects in Python

Pass Statement

- Python contains a special `pass` statement that performs no operation or changes of state
  - Used when you do not want to specify any functionality or behavior but syntactically need a statement

```
def empty_function():
    pass
```

```
class empty_class:
    pass
```

Classes

```
class ClassName:
    <statements>
```

- Instantiation of a new object:

```
foo = ClassName()
bar = Mod_Name.ClassName() # class is in module Mod_Name
```

- `foo` is an object that is an instance of `ClassName`
- `bar` is an object that is an instance of `ClassName`

- Notice that to instantiate a class the name of the class is called like a function

Classes (2)

- Classes can be placed in module or directly in your script file

- Consider using a module to organize or group similar classes together
  - Classes are imported just like functions

Member Variables

- Member variables represent the state of an object

- Accessing a member variable from outside of the class:

```
ObjA.my_var = 10
ObjB.my_var = 20
```

- Each instance of the above objects maintains it's own copy of a member variable called `my_var`
  - Member variables can be mutable types so a single value can be shared between many objects

```
ObjA.my_list = range(10)
ObjB.my_list = ObjA.my_list   # my_list refers to the same list in
ObjB.my_list.append("hello")  # both objects
```

Member Variables (2)

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

- The class Cat is defined with two member variables: `name` and `age`
- All member variables should be initialized explicitly in the constructor
  - `self` is a special reference to the specific object that is being instantiated by the constructor
  - See the next section for more details of the `self` reference

Member Variables (3)

```
# Instantiate a new instance of Cat
kitty = Cat("Garfield", 32)

# Print the values of its member variables
Print('My name is {}.'.format(kitty.name))
Print('I am {} years old.'.format(kitty.age))

>>> My name is Garfield.
>>> I am 32 years old.
```

Member Functions

- Member functions are declared just like normal Python functions

```
def function(self, arg1, arg2, …):
     <function body>
```

- The first argument to a member function is a special "`self`" value
  - `self` is a reference to a specific instance
  - `self` is required for any member function

Member Functions (2)

- So why do we need the `self` argument?

- When we define a class we are specifying the member variables and member functions for every possible instance of an object
  - At any time there are multiple objects of the same class that exist

- To differentiate between all of the potential objects that exist a reference to a specific object is provided
  - State for a particular object can then be modified or accessed through the self reference

Member Functions (3)

- Two ways to invoke member functions

```
ClassName.Function(Var, args…)

Var.Function(args…)
```

- Most of the time the second method is used and `Var` is implicitly passed as the first argument of the function

Member Functions (4)

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

      def speak(self):
            print('My name is {}.'.format(self.name))
            print('I am {} years old.'.format(self.age))


# Instantiate a new instance of Cat
kitty = Cat("Garfield", 32)

# Invoke the speak member function
kitty.speak()

>>> My name is Garfield.
>>> I am 32 years old.
```

`__init__(self, …)`

- `__init__` is reserved for defining the constructor
  - See previous slides for an example
  - `__init__` is not called explicitly, the class name is used instead
    - Unless you are calling the constructor of a parent object (inheritance)

```
some_obj = ObjType(arg1, arg2, …)
my_pet = Cat("Spot", 12)
```

Returning Objects

- Many functions you write may produce an object as the return value
  - You can return the result of a constructor

```
def make_foo(i):
     # Return a new Foo object
     return Foo(i)

my_foo = make_foo(10)
```

Special Member Functions

- Some member functions are “special”
  - Begin and end with two (2) underscores
  - Already saw the constructor `__init__`

- Most of them provide convenience and help integrate your objects naturally into Python

Special Member Functions (2)

`__add__(self, other)`	Overloads the `+` operator
`__sub__(self, other)`	Overloads the `+` operator
`__mul__(self, other)`	Overloads the `*` operator
`__truediv__(self, other)`	Overloads the `/` operator
`__lt__(self, other)`	Overloads the `<` operator
`__gt__(self, other)`	Overloads the `>` operator
`__ge__(self, other)`	Overloads the `>=` operator
`__le__(self, other)`	Overloads the `<=` operator
`__eq__(self, other)`	Overloads the `==` operator
`__ne__(self, other)`	Overloads the `!=` operator

Special Member Functions (3)

`__str__(self)`	Returns a string representation Overloads `str(obj)`
`__int__(self)`	Returns an integer representation Overloads `int(obj)`
`__float__(self)`	Returns a float representation Overloads `float(obj)`
`__len__(self)`	Returns a lengths Overloads `len(obj)`
`__getitem__(self, k)`	Overloads the `[]` operator e.g. `obj[k]`
`__setitem__(self, k, v)`	Overloads the `[]` operator e.g. `obj[k] = v`
`__contains__(self, item)`	Overloads the `in` operator e.g. item in obj

Inheritance

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

# Base class name is in () after class name
class Derived(Base):
    def __init__(self, name, age):
        # Need to call parent constructor!
        Base.__init__(self, name)
        self.age = age
```

Inheritance (2)

- When an object inherits from another object the derived constructor should call the parent constructor

```
Base.__init__(self, name)
```

  - The explicit function call must be used to disambiguate the `__init__` because it is a member function of both objects

- This ensures that all of the member variables inherited from the parent are initialized in the most derived object

Inheritance (3)

```
# Base class is Student
class Student:
        def __init__(self, name):
                self.name = name
                self.knowledge_level = 0

        def study(self, hours):
                self.knowledge_level += hours * 0.01

        def print_knowledge_level(self):
                print("{} has a knowledge level of {}".
                format(self.name, self.knowledge_level)

class GoodStudent(Student):
        def __init__(self, name):
                Student.__init__(self, name)

        def study(self, hours):
                # Entirely replace the behavior of study
                # Function override
                self.knowledge_level += hours * 10
```

Inheritance (4)

```
class BadStudent(Student):
  def __init__(self, name):
    # initialize the member variables of Student
    Student.__init__(self, name)

  def study(self, hours):
    # Enhance behavior of study
    # Implemented in terms of Student study()
    hours = hours - 1

    # Calling the base class functionality
    Student.study(self, hours/5)
```

Inheritance (5)

```
class AvgStudent(Student):
  def __init__(self, name):
    Student.__init__(self, name)

  # Avg student will not specialize Study so no need
  # to re-define study
```

- Functions in a class may call functions of the same name from an inherited class
- Allows you to extend the functionality or completely redefine functions in other classes

Inheritance (6)

```
Good = GoodStudent("Goldfarb") # instantiate various
Bad = BadStudent("Mike")       # Student objects
Avg = AvgStudent("Foo")         # Always use the most derived
object name

Good.study(1)
Bad.study(1)
Avg.study(1)

Good.print_knowledge_level()
>>> "Goldfarb has a knowledge level of 10"

Bad.print_knowledge_level()
>>> "Mike has a knowledge level of 0"

Avg.print_knowledge_level()
>>> "Foo has a knowledge level op 0.01"
```

Polymorphism

- Polymorphism in Python comes directly from dynamic typing

- As long as an object has a function with the same name and arguments it can be treated in a uniform way
  - Even if the objects do not inherit from a common parent!

Polymorphism (2)

```
class Dog:
  def bark(self, s="woof woof"):
    print("Dog Bark: {}".format(s))

class Duck:
  def bark(self, s="quack quack"):
      print("Duck Bark: {}".format(s))

# "a" behaves like a Dog or a Duck depending on the
object
for a in [Dog(), Duck(), Dog()]:
  a.bark()

>>> Dog Bark: woof woof
>>> Duck Bark: quack quack
>>> Dog Bark: woof woof
```

Function Overloading

- Python does not support traditional function overloading, rather, the special `*args` and `**kwargs` function arguments are used instead. These are not keywords, but used by convention.

- `*args` – Represents a variable number of function arguments.
  - Stored as a tuple. The * acts as an “unpacking” operator.

- `**kwargs` – Represents a variable number of named arguments
  - Stored as a dictionary
  - Arguments are specified as `argName=argValue`

Function Overloading (2)

```
def foo(*args):
  # args is a tuple of values
  print("Num args = {}".format(len(args)))
  if len(args) >= 2:
    print("Arg2: {}".format (str(args[1]))

foo("bar", [1,2,3], "hello")
>>> "Num args = 3"
>>> "Arg2: [1,2,3]"

Foo()
>>> "Num args = 0"
```

Function Overloading (3)

```
def foo(**kwargs):
  # kwargs is a dictionary of arguments
  if “val” in kwargs:
    print("val = {}".format((str(kwargs["val"])))

  print(kwargs.keys())
  print(kwargs.values())

foo(a="bar", val=[1,2,3], bar="hello")
>>> "val = [1,2,3]"
>>> ['a', 'val', 'bar']
>>> ['bar', [1,2,3], 'hello']
```

Operator Overloading

- Many of the special member functions are actually called using the built in operators
  - `+`, `-`, `*`, `/`, `in`, `<`, `>=` etc.

- Be very careful about how you implement the operator
  - Arithmetic operators could potentially induce side affects that are not intended
  - If your object should be immutable then operators should always return a new object
  - Forgetting to return a result of addition may also make usage confusing

Operator Overloading (2)

```
class FooNum:
  def __init__(self, i):
    self.i = int(i)

  def __add__(self, o):
      # Provide + to FooNumber
      # Addition creates a new FooNum
      # Self and o are never changed!
      tmp = FooNum((self.i + o.i) * 2)

      return tmp

A=FooNum(1)
B=FooNum(2)
C = A + B   # C.i is 6, A.i is 1, B.i is 2
```

Operator Overloading (3)

```
class BarNum:
  def __init__(self, i):
    self.i = int(i)

  def __add__(self, o):
      # Self is modified!
      # Side effect in + might confuse people!
      self.i = (self.i + o.i)

      # Even if we create a new BarNum
      return BarNum(self.i*2)

A=BarNum(1)
B=BarNum(2)
C = A + B   # C.i is 6, A.i is 3, B.i is 2
```

Operator Overloading (4)

```
class BarNum:
  def __init__(self, i):
    self.i = int(i)

  def __eq__(self, o):
    # Support == operator
    return self.i == o.i

  def __lt__(self, o):
    # Support < operator
    return self.i < o.i

  def __gt__(self, o):
    # Other comparison operators can be defined in terms on == and <
    return not self.__lt__(o) and not self.__eq__(o)

  def __ge__(self, o):
    return self.__gt__(o) or self.__eq__(o)

  def __ne__(self, o):
    return not self.__eq__(o)
```

Operator Overloading (5)

```
class MySet:
  def __init__(self):
    self.items=[]

  def append(self, item):
    if item not in self.items:
      self.items.append(item)

  def __len__(self):
    # Add support for len(x) function
    return len(self.items)

  def __contains__(self, item):
    # Add support for in operator
    return item in self.items

S = MySet()
S.append(2)
S.append("bar")
if "bar" in S:   # Prints 2
  print(len(S))
```