ECE326

PROGRAMMING LANGUAGES

Lecture 7 : Object-Oriented Programming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Object-Oriented Programming

- Object
  - Contains both state (data) and behaviour (code)

	Data	Code
C++	Member variable	Member function
Java	Field	Method
Python	Data attribute	Method

  - Models real world things and concepts
  - Provides *encapsulation*
    - E.g. restricts direct access to some parts of an object

Object-Oriented Programming

- Sending messages
  - object *maps* messages to values.
  - Responds to a message by looking up the corresponding value
- Class
  - A “blueprint” to create objects of the same behaviour
  - Can create *instances* of a class, which determines their type
- Instance
  - Manifestation of an object
  - Emphasizes the distinct identity of the object

Object-Oriented Programming

- Class-based programming
  - Inheritance occurs by deriving classes from existing ones
    - i.e. occurs through *subtyping*
  - E.g. Student is a subclass of Person
- Prototype-based programming
  - Inheritance occurs by copying existing object and adding to it
  - Individual objects would be cloned from the prototype
  - E.g. Student object is Person object with an extra ID field
    - jack instance is created by copying student object and setting the names and ID fields to real values (instead of the defaults)

Object-Oriented Programming

- Class-based programming
  - Usually used by statically-type languages: e.g. C++, Java
- Prototype-based programming
  - Usually only possible for dynamically-typed languages
    - Only one stands out: JavaScript
- Python
  - Class-based programming
  - BUT
    - Can support prototype-based programming

Python Class

- Declared via `class` keyword
- Can be completely empty
  - Same as C++, but MUCH MORE POWERFUL

```
class Foo:
    pass

# create an instance of Foo
>> a = Foo()

# unlike C++, you can ADD
# new fields
>> a.bar = 3
>> a.bar
3
```

```
>> a.baz
AttributeError: Foo instance
has no attribute 'baz'

>> type(a)
<type ‘instance’>
>> print a
<__main__.Foo instance at …>
>> isinstance(a, Foo)
True
```

Constructor

- Special `__init__` method
  - `Self` variable
    - Variable that references the current instance (`this` in C++)
    - Must be first parameter of *all* instance methods

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

# create an instance of Point (calls Point.__init__)
>> a = Point(3, 4)
>> a.x
3
```

Instance Method

- Always takes class instance variable as first argument
- `self:` named by convention, denotes ‘this’ instance

```
class Point:
    …
    def distance(self, other):
        dx = self.x – other.x
        dy = self.y – other.y
        return math.sqrt(dx*dx + dy*dy)

>> a = Point(4, 5)
>> b = Point(1, 1)
>> a.distance(b)   # a is the first argument to distance()
5.0
```

Attribute

- In Python, means either data members or methods
- Can add new ones *after* __init__
  - Although it may confuse other programmers

```
class Point:
    def move(self, dx, dy):
        self.x, self.y = self.x + dx, self.y + dy
        self.moved = True

>> a = Point(0, 0)
>> a.moved
AttributeError: ‘Point’ object has no attribute ‘moved’
>> a.move(5, 5)
>> a.moved
True
```

Attribute

- Can also remove attributes

```
class Point:
    def move(self, dx, dy):
        self.x, self.y = self.x + dx, self.y + dy
        self.moved = True
        if self.x == 0 and self.y == 0:
            del self.moved

>> a = Point(3, 2)
>> a.move(-3, -2)
>> a.moved
AttributeError: ‘Point’ object has no attribute ‘moved’
```

dir

- Convenience function to learn about an object
  - Returns list of attributes
- Use this as a way to debug your program

```
class A:
    def __init__(self):
        pass

>> dir()
['A', '__builtins__', '__name__', …]

>> dir(A)
['__init__', '__class__', '__delattr__', '__dict__', …]
```

__dict__

- Let’s dig a little deeper into how an instance works

```
>> a = [1, 2, 3]
>> a.x = 5
AttributeError: 'list' object has no attribute 'x'

>> b = Foo()
>> b.x = 5
>> b.__dict__
{‘x’: 5}
>> a.__dict__
AttributeError: 'list' object has no attribute '__dict__'

>> b.__dict__[‘y’] = “hello”
>> b.y
“hello”
```

__slots__

- Denies creation of __dict__
  - Prevents instance from accepting new attributes
    - Evil: can put __dict__ into __slots__
- Also reduce memory consumption

```
class Point:
    __slots__ = (‘x’, ‘y’)
    def __init__(self, x=0, y=0): # uses default arguments
        self.x, self.y = x, y

>> a = Point()
>> a.z = 5
AttributeError: ‘Point’ object has no attribute ‘z’
```

Encapsulation

- There is no private keyword in Python
  - Cannot (easily) protect attributes from direct access
- Convention
  - Start name of attribute with underscore _
    - Tells other programmers: “it is private (pretty please)”

```
class Submarine:
    def _launch_missile(self):
        …
    def try_launch_missile(self, password):
        if self.password == password:
            self._launch_missile()
        else:
            raise PermissionError(“access denied”)
```

Class Attribute

- In Python, class is also an object
- An instance can access its class's attributes
  - That's how it calls methods defined in the class
  - It can also access class member variables

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

>> a = Point()
>> a.origin
(0, 0)
```

Class Attribute

- Class attributes are *read only* by their instances
  - Reassignment bounds new attribute to the instance
  - Typically not what you want to do

```
class Point:
    origin = (0, 0)          # static class variable in C++

>> a, b = Point(), Point()
>> a.origin = (1, 1)         # now local to a
>> a.origin
(1, 1)
>> b.origin                  # still refers to Point.origin
(0, 0)
>> Point.origin              # not changed by a
(0, 0)
```

Instance Method

- When instance calls functions defined by its class, it passes itself in as the first argument
- Alternatively, you can do it manually

```
class Point:
    def move(self, dx, dy):
        …

>> a = Point()          # (0, 0)
>> Point.move(a, 3, 4) # (3, 4)
>> func = Point.move   # similar to member function pointer
>> func(a, 2, 2)
>> a.x, a.y
(5, 6)
```

Class Method

- Takes **class** as first argument
- Use `@classmethod` decorator (more on this later)

```
class Point:
    origin = (0, 0)
    @classmethod
    def debase(cls, dx, dy):
        cls.origin = (dx, dy)

>> a = Point()
>> a.origin
(0, 0)
>> Point.debase(3, 7)
>> a.origin
(3, 7)
```

Ad-Hoc Polymorphism

- Ability for an entity to behave differently based on input or contained types
- Function Overloading
  - Functions of same name with different implementations
  - Not supported by Python (why?)
- Operator Overloading
  - Operator has different implementation based on operand(s)
  - Allows use of notation/syntax similar to basic types
    - E.g. use `a + b` to add two complex numbers instead of `a.add(b)`

Operator Overloading

- In Python, almost every operator has a corresponding special method that can be invoked if defined

```
class Complex:
    def __init__(self, r=0, i=0):
        self.r, self.i = r, i
    def __add__(self, other):
        if isinstance(other, (int, float)):
            self.r += other
        elif isinstance(other, Complex):
            self.r += other.r
            self.i += other.i
        return self

>> Complex(3, 2) + 1.5
(4.5, 2)                    # I cheated
```

__str__

- Converts objects to string
  - Automatically done when passed to `print`

```
class Complex:                  # continue from previous example
      def __str__(self):
          if self.i == 0:
              return str(self.r)
          elif self.i > 0:
              return "{} + {}i".format(self.r, self.i)
          elif self.i < 0:
              return "{} - {}i".format(self.r, -self.i)
          else:
              return "{}i".format(self.i)

>> print(Complex(3, -2))
3 – 2i
```

Index Operator

```
class Bitmap:
    def __init__(self, data=0):
        self.data = data
    def __getitem__(self, idx):
        return self.data & (1 << idx)
    def __setitem__(self, idx, val):
        if val:
            self.data |= (1 << idx)
        else:
            self.data &= ~(1 << idx)
```

```
>> bm = Bitmap(0xE3)
>> print(bm[4])
0
>> print(bm[1])
2
```

```
>> bm[0] = 0
>> bm[4] = 1
>> print("0x%x"%bm.data)
0xf2
```

Operator Overloading

- Relational Operators
  - `__eq__`, `__ne__`, …etc
- Reverse Arithmetic Operators
  - Used when instance at right side of operator
    - E.g. 3 + Complex(-1, 2)
  - `__radd__`, `__rsub__`, `__rmul__`, …etc
- Assignment Operator
  - Performs name binding in Python
  - Cannot be overloaded