

# Ch15-Overloading-Polymorphism

October 30, 2020

## 1 More OOP, Operator Overloading and Polymorphism

[http://openbookproject.net/thinkcs/python/english3e/even\\_more\\_oop.html](http://openbookproject.net/thinkcs/python/english3e/even_more_oop.html)

### 1.1 MyTime

- class that records the time of day
- provide **init** method so every instance is created with appropriate attributes and initialization

```
[3]: class MyTime:
    """MyTime class that keeps track of time of day"""
    def __init__(self, hrs=0, mins=0, secs=0):
        """ Creates a MyTime object initialized to hrs, mins, secs """
        self.hours = hrs
        self.minutes = mins
        self.seconds = secs

    def __str__(self):
        return "{:02}:{:02}:{:02}".format(self.hours, self.minutes, self.
↪seconds)
```

```
[4]: tim1 = MyTime(11, 59, 3)
```

```
[5]: print(tim1)
```

11:59:03

### 1.2 Functions can be pure and modifiers

- what functions should be part of class of methods?
- typically, all the rfunctions that operate on or use attributes of class should be part of the class called methods

### 1.3 pure functions

- pure functions do not have side effects, such as modifying parameters and global variables
- similar to constant functions in C++ world
- getter methods are pure functions
- e.g.: see `add_time()`

```
[31]: def add_time(t1, t2):
        h = t1.hours + t2.hours
        m = t1.minutes + t2.minutes
        s = t1.seconds + t2.seconds

        while s >= 60:
            s -= 60
            m += 1

        while m >= 60:
            m -= 60
            h += 1

        sum_t = MyTime(h, m, s)
        return sum_t
```

```
[32]: current_time = MyTime(9, 50, 45)
        bread_time = MyTime(2, 35, 20)
        done_time = add_time(current_time, bread_time)
        print(done_time)
```

12:26:05

## 1.4 modifiers

- functions that modify the object(s) it gets as parameter(s)
- setter methods are modifiers

```
[22]: # function takes MyTime t and secs to update t
        def increment(myT, seconds):
            myT.seconds += seconds
            mins = myT.seconds//60

            myT.seconds = myT.seconds%60
            myT.minutes += mins

            hours = myT.minutes//60
            myT.hours += hours
            myT.minutes = myT.minutes%60
```

```
[23]: current_time = MyTime(9, 50, 45)
        print(current_time)
```

09:50:45

```
[33]: increment(current_time, 60*60)
```

```
[34]: print(current_time)
```

10:50:45

## 1.5 Converting increment() to a method

- OOD prefers that functions that work with objects to be part of the class or methods
- increment can be a useful method for MyTime class

```
[35]: class MyTime:
    def __init__(self, hrs=0, mins=0, secs=0):
        """ Create a new MyTime object initialized to hrs, mins, secs.
            The values of mins and secs may be outside the range 0-59,
            but the resulting MyTime object will be normalized.
        """
        self.hours = hrs
        self.minutes = mins
        self.seconds = secs

        # Calculate total seconds to represent
        self.__normalize()

    def __str__(self):
        return "{:02}:{:02}:{:02}".format(self.hours, self.minutes, self.
→seconds)

    def to_seconds(self):
        """ Return the number of seconds represented
            by this instance
        """
        return self.hours * 3600 + self.minutes * 60 + self.seconds

    def increment(self, seconds):
        self.seconds += seconds
        self.__normalize()

    # should be treated as private method
    def __normalize(self):
        totalsecs = self.to_seconds()
        self.hours = totalsecs // 3600          # Split in h, m, s
        leftoversecs = totalsecs % 3600
        self.minutes = leftoversecs // 60
        self.seconds = leftoversecs % 60
```

```
[36]: # improved add_time function
def add_time(t1, t2):
    secs = t1.to_seconds() + t2.to_seconds()
    return MyTime(0, 0, secs)
```

```
[37]: # test add_time function
current_time = MyTime(9, 50, 45)
bread_time = MyTime(2, 35, 20)
done_time = add_time(current_time, bread_time)
print(done_time)
```

12:26:05

1.5.1 similarly, `add_time` can be moved inside `MyTime` class as a method

```
[38]: class MyTime:
    def __init__(self, hrs=0, mins=0, secs=0):
        """ Create a new MyTime object initialized to hrs, mins, secs.
            The values of mins and secs may be outside the range 0-59,
            but the resulting MyTime object will be normalized.
        """
        self.hours = hrs
        self.minutes = mins
        self.seconds = secs
        # Calculate total seconds to represent
        self.__normalize()

    def __str__(self):
        return "{:02}:{:02}:{:02}".format(self.hours, self.minutes, self.
→seconds)

    def to_seconds(self):
        """ Return the number of seconds represented
            by this instance
        """
        return self.hours * 3600 + self.minutes * 60 + self.seconds

    def increment(self, secs):
        self.seconds += secs
        self.__normalize()

    def __normalize(self):
        totalsecs = self.to_seconds()
        self.hours = totalsecs // 3600          # Split in h, m, s
        leftoversecs = totalsecs % 3600
        self.minutes = leftoversecs // 60
        self.seconds = leftoversecs % 60

    def add_time(self, other):
        return MyTime(0, 0, self.to_seconds() + other.to_seconds())
```

```
[39]: current_time = MyTime(9, 50, 45)
      bread_time = MyTime(2, 35, 20)
      done_time = current_time.add_time(bread_time)
      print(done_time)
```

12:26:05

## 1.6 special methods / operator overloading

- <https://docs.python.org/3/reference/datamodel.html>
- how about  $t1 = t2 + t3$  just like adding primitive types
- + operator appends two strings, but adds two integers or floats
- the same operator has different meaning for different types called operator overloading
- replace `add_time` with built-in special method **add** to overload + operator

```
[62]: class MyTime:

      def __init__(self, hrs=0, mins=0, secs=0):
          """ Create a new MyTime object initialized to hrs, mins, secs.
              The values of mins and secs may be outside the range 0-59,
              but the resulting MyTime object will be normalized.
          """
          self.hours = hrs
          self.minutes = mins
          self.seconds = secs
          # Calculate total seconds to represent
          self.__normalize()

      def __str__(self):
          return "{:02}:{:02}:{:02}".format(self.hours, self.minutes, self.
→seconds)

      def to_seconds(self):
          """ Return the number of seconds represented
              by this instance
          """
          return self.hours * 3600 + self.minutes * 60 + self.seconds

      def increment(self, secs):
          self.seconds += secs
          self.normalize()

      def __normalize(self):
          totalsecs = self.to_seconds()
          self.hours = totalsecs // 3600          # Split in h, m, s
          leftoversecs = totalsecs % 3600
          self.minutes = leftoversecs // 60
          self.seconds = leftoversecs % 60
```

```
def __add__(self, other):
    return MyTime(0, 0, self.to_seconds() + other.to_seconds())
```

```
[66]: current_time = MyTime(9, 50, 45)
bread_time = MyTime(2, 35, 20)
done_time = current_time + bread_time # equivalent to: done_time = current_time.
→ __add__(bread_time)
print(done_time)
```

12:26:05

## 1.7 add two points

- overloading our Point class to be able to add two points

```
[75]: class Point:
    """
    Point class represents and manipulates x,y coords
    """
    count = 0

    def __init__(self, xx=0, yy=0):
        """Create a new point with given x and y coords"""
        self.x = xx
        self.y = yy
        Point.count += 1

    def dist_from_origin(self):
        import math
        dist = math.sqrt(self.x**2+self.y**2)
        return dist

    def __str__(self):
        return "({}, {})".format(self.x, self.y)

    def move(self, xx, yy):
        self.x = xx
        self.y = yy

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)

    def __mul__(self, other):
        """
        computes dot product of two points
```

```

        """
        return self.x * other.x + self.y * other.y

    def __rmul__(self, other):
        """
        if the left operand is primitive type (int or float)
        and the right operand is a Point, Python invokes __rmul__
        which performs scalar multiplication
        """
        return Point(other * self.x, other * self.y)

```

```

[80]: p1 = Point(2, 2)
      p2 = Point(10, 10)
      p3 = p1 + p2
      print(p3)
      print(p1 * p3)
      print(4 * p1)

```

(12, 12)

48

(8, 8)

## 1.8 some special methods

exercise 1: implement some relevant special methods for Point class and test them

exercise 2: implement some relevant special methods for Triangle class defined in previous chapter and test them

## 1.9 Polymorphism

- most methods work on a specific new class type we create
- some methods we want to apply to many types, such as arithmetic operations + in previous example
- e.g., multadd operation (common in linear algebra) takes 3 arguments, it multiplies the first two and then adds the third
- function like this that can take arguments with different types is called polymorphic

```

[71]: def multadd(x, y, z):
      return x * y + z

```

```

[81]: multadd(3, 2, 1)

```

```

[81]: 7

```

```

[82]: p1 = Point(3, 4)
      p2 = Point(5, 7)

```

```
print(multadd(2, p1, p2))
```

```
(11, 15)
```

```
[83]: print(multadd (p1, p2, 1))
```

```
44
```

## 1.10 duck typing rule - dynamic binding

- duck test: “If it walks like a duck and it quacks like a duck, then it must be a duck”
- to determine whether a function can be applied to a new type, we apply Python’s fundamental rule of polymorphism, called duck typing rule: if all of the operations inside the function can be applied to the type, the function can be applied to the type
- e.g.: [https://en.wikipedia.org/wiki/Duck\\_typing](https://en.wikipedia.org/wiki/Duck_typing)

```
[85]: class Duck:
        def fly(self):
            print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

# polymorphism
def lift_off(entity):
    entity.fly()
    # only throws error if some entity doesn't have fly attribute during_
    →run-time!
    # statically typed languages such as C++ give compile time errors!

duck = Duck()
airplane = Airplane()
whale = Whale()

lift_off(duck) # prints `Duck flying`
lift_off(airplane) # prints `Airplane flying`
lift_off(whale) # Throws the error `Whale' object has no attribute 'fly'`
```

```
Duck flying
```

```
Airplane flying
```

```
-----
AttributeError
```

```
Traceback (most recent call last)
```



```

<ipython-input-85-44deada23680> in <module>()
    21 lift_off(duck) # prints `Duck flying`
    22 lift_off(airplane) # prints `Airplane flying`
--> 23 lift_off(whale) # Throws the error `Whale' object has no attribute_
    ↪ 'fly'`

<ipython-input-85-44deada23680> in lift_off(entity)
    12
    13 def lift_off(entity):
--> 14     entity.fly() # only throws error if some entity doesn't have fly_
    ↪ attribute during run-time!
    15     # statically typed languages such as C++ give compiler errors!
    16

AttributeError: 'Whale' object has no attribute 'fly'

```

[ ]: