

## LECTURE 18 — CLASSES, PART 1

### 33.1 Overview

- Define our own types and associated functions
- Encapsulate data and functionality
- Raise the “level of abstraction” in our code
- Make code easier to write and test
- Reuse code

### 33.2 Potential Examples

In each of these, think about what data you might need to store to represent the “object” and what functionality you might need to apply to the data.

- Date
- Time
- Point
- Rectangle
- Student
- Animal
- Molecule

### 33.3 An Example from Earlier in the Semester

- Think about how difficult it was to keep track of the information about each restaurant in the Yelp data.
- You had to:
  - Remember the indices of (a) the restaurant name, (b) the latitude and longitude, (c) the type of restaurant, (d) the address, etc.
  - Form a separate list inside the list for the ratings.
  - Write additional functions to exploit this information
- If we used a class to represent each restaurant:

- All of the information about the restaurant would be stored and accessed as named attributes
- Information about the restaurants would be accessed through functions that we write for the class.

## 33.4 Point2d Class

- Simplest step is to just tell Python that `Point2d` will exist as a class, deferring the addition of information until later.

```
class Point2d(object):  
    pass
```

- The Python reserved word `pass` says that this is the end of the class definition.
  - We will not need this later when we put information into the class.

## 33.5 Attributes

- Classes do not get interesting until we put something in them.
- The first thing we want is variables so that we can put data into a class.
  - In Python these variables are often called *attributes*.
  - Other languages call them *member variables*.
- We will see three different ways to specify attributes.

## 33.6 Assigning Attributes to Each Instance

- Points have an `x` and a `y` location, so we can write, for example,

```
from math import sqrt  
p = Point2d()  
p.x = 10  
p.y = 5  
dist_from_origin = sqrt(p.x**2 + p.y**2)
```

- We have to do this for each class instance.
- This is prone to mistakes:
  - Could forget to assign the attributes
  - Could accidentally use different names for what is intended to be the same attribute.
- Example of an error

```
q = Point2d()  
q.x = -5  
dist_from_origin = sqrt(q.x**2 + q.y**2)    # q.y does not exist
```

## 33.7 Defining the Attributes Inside the Class

- The simplest way to make sure that all variables that are instances of a class have the appropriate attributes is to define them inside the class.
- For example, we could redefine our class as

```
class Point2d(object):
    x = 0
    y = 0
```

- All instances of `Point2d` now have two attributes, `x` and `y`, and they are each initialized to 0.
- We no longer need the pass because there is now something in the class.

## 33.8 Defining the Attributes Through An Initializer / Constructor

- We still need to initialize `x` and `y` to values other than 0:

```
p = Point2d()
p.x = 10
p.y = 5
```

- What we'd really like to do is initialize them at the time we actually create the `Point2d` object:

```
p = Point2d(10,5)
```

- We do this through a special function called an *initializer* in Python and a *constructor* in some other programming languages.
- Inside the class this looks like

```
class Point2d(object):
    def __init__( self, x0, y0 ):
        self.x = x0
        self.y = y0
```

- Our code to create the point now becomes

```
p = Point2d(10,5)
```

- Notes:
  - Python uses names that start and end with two `'_'` to indicate functions with special meanings. More on this later in the lecture.
  - The name `self` is special notation to indicate that the object itself is passed to the function.
- If we'd like to initialize the point to (0,0) without passing these values to the constructor every time then we can specify default arguments

```
class Point2d(object):
    def __init__( self, x0=0, y0=0 ):
        self.x = x0
        self.y = y0
```

allowing the initialization

```
p = Point2d()
```

## 33.9 Methods — Functions Associated with the Class

- We create functions that operate on the class objects inside the class definition:

```
import math

class Point2d(object):
    def __init__( self, x0, y0 ):
        self.x = x0
        self.y = y0

    def magnitude(self):
        return math.sqrt(self.x**2 + self.y**2)

    def dist(self, o):
        return math.sqrt( (self.x-o.x)**2 + (self.y-o.y)**2 )
```

these are called *methods*

- This is used as

```
p = Point2d(0,4)
q = Point2d(5,10)
leng = q.magnitude()
print("Magnitude {:.2f}".format( leng ))
print("Distance is {:.2f}".format( p.dist(q)))
```

- The method `magnitude` takes a single argument, which is the `Point2d` object called `self`. Let's examine this:

- The call `q.magnitude()` appears to have no arguments, but when Python sees this, it turns it into its equivalent:

```
Point2d.magnitude(q)
```

which is completely legal Python syntax.

- The name `self` is not technically special in Python, but it is used by convention to refer to the object that the method is “called upon”. This is *q* in the call `q.magnitude()`

- The method `dist` takes two `Point2d` objects as arguments. The example call

```
p.dist(q)
```

becomes

```
Point2d.dist(p,q)
```

so now argument `p` maps to parameter `self` and argument `q` maps to parameters `o`

## 33.10 Lecture Exercises, Part 1

Our lecture exercises for today will involve adding to the `Point2d` class and testing it. Make sure you have downloaded the `Point2d.py` file from the Piazza site.

We will allow some time to work on the first lecture exercise.

## 33.11 Operators and Other Special Functions

- We'd like to write code that uses our new objects in the most intuitive way possible.
- For our point class, this involves use of operators such as

```
p = Point2d(1,2)
q = Point2d(3,5)
r = p+q
s = p-q
t = -s
```

- Notice how in each case, we work with the `Point2d` variables (objects) just like we do with `int` and `float` variable (objects).
- We implement these by writing the special functions `__add__`, `__sub__`, and `__neg__`
- For example, inside the `Point2d` class we might have

```
def __add__(self,other):
    return Point2d(self.x + other.x, self.y+other.y)
```

Very important: this creates a new `Point2d` object.

- When Python sees `p+q`, it turns it into the function call

```
Point2d.__add__(p,q)
```

which is exactly the syntax of the function definition we created.

- We have already seen this with operators on integers and strings. As examples,

```
5+6
```

is equivalent to

```
int.__add__(5,6)
```

and

```
str(13)
```

is equivalent to

```
int.__str__(13)
```

- Implicit in this discussion is the notion that `int` is in fact a class in Python. The same is true of `str` and `float` and `list`.
- Note that we can also define boolean operators such as `==` and `!=` through the special functions `__eq__` and `__neq__`

## 33.12 Classes and Modules

- Each class should generally be put into its own module, or several closely-related classes should be combined in a single module.
  - We are already doing this with `Point2d`.
- Doing so is good practice for languages like C++ and Java, where classes are placed in separate files.
- Testing code can be included in the module or placed in a separate module.
- We will demonstrate this in class and post the result on the course website.

## 33.13 More Lecture Exercises

At this point we will stop and take a bit of time to work on the next part of the lecture exercises.

## 33.14 When to Modify, When to Create New Object

- Some methods, such as `scale`, modify a single `Point2d` object
- Other methods, such as our operators, create new `Point2d` objects without modifying existing ones.
- The choice between this is made on a method-by-method basis by thinking about the meaning — the *semantics* — of the behavior of the method.

## 33.15 Programming Conventions

- Don't create attributes outside the class.
- Don't directly access or change attributes except through class methods.
  - Languages like C++ and Java have constructions that enforce this.
  - In languages like Python it is not a hard-and-fast rule.
- Class design is often most effective by thinking about the required methods rather than the required attributes.
  - As an example, we rarely think about how the Python `list` and `dict` classes are implemented.

## 33.16 Time Example

- In the remainder of the lecture, we will work through an extended example of a `Time` class
- By this, we mean the time of day, measured in hours, minutes and seconds.
- We'll brainstorm some of the methods we might need to have.
- We'll then consider several different ways to represent the time internally:
  - Hours, minutes and seconds
  - Seconds only

- Military time
- Despite potential internal differences, the methods — or at least the way we call them — will remain the same
  - This is an example of the notion of *encapsulation*, which we will discuss more in Lecture 19.
- At the end of lecture, the resulting code will be posted and tests will be generated to complete the class definition.

## 33.17 Summary

- Define new types in Python by creating classes
- Classes consist of *attributes* and *methods*
- Attributes should be defined and initialized through the special method call `__init__`. This is a *constructor*
- Other special methods allow us to create operators for our classes.
- We looked at a *Point2d* and *Time* example.

