Lecture 7: Classes
Classes are programmer-defined types

Sometimes we use a collection of variables to represent a specific object

**Example:** we used a tuple of tuples to represent a matrix

**Example:** representing state of a board game
  List of players, piece positions, etc.

**Example:** representing a statistical model
  Want to support methods for estimation, data generation, etc.

**Important point:** these data structures quickly become very complicated, and we want a way to encapsulate them. This is a core motivation (but hardly the only one) for **object-oriented programming**.
Classes encapsulate data types

**Example:** I want to represent a point in 2-dimensional space $\mathbb{R}^2$

**Option 1:** just represent a point by a 2-tuple

**Option 2:** make a point *class*, so that we have a whole new data type

Additional good reasons for this will become apparent shortly!

```
1 class Point:
2     '''Represents a 2-d point.'''

1 print(Point)
<class '__main__.Point'>
```

Class header declares a new class, called `Point`.

Docstring provides explanation of what the class represents, and a bit about what it does. This is an ideal place to document your class.

Credit: Running example adapted from A. B. Downey, *Think Python*
Classes encapsulate data types

Example: I want to represent a point in 2-dimensional space $\mathbb{R}^2$

Option 1: just represent a point by a 2-tuple

Option 2: make a point class, so that we have a whole new data type

Additional good reasons for this will become apparent shortly!

```
1 class Point:
2     '''Represents a 2-d point.'''
```

Credit: Running example adapted from A. B. Downey, *Think Python*
Creating an object: Instantiation

This defines a class `Point`, and from here on we can create new variables of type `Point`.
Creating an object: Instantiation

Creating a new object is called **instantiation**. Here we are creating an instance \( p \) of the class `Point`.

Indeed, \( p \) is of type `Point`.

**Note:** An instance is an individual object from a given class. In general, the terms **object** and **instance** are interchangeable: an object is an instantiation of a class.
Assigning Attributes

This dot notation should look familiar. Here, we are assigning values to attributes \( x \) and \( y \) of the object \( p \). This both creates the attributes, and assigns their values.

Once the attributes are created, we can access them, again with dot notation.

Attempting to access an attribute that an object doesn’t have is an error.
Thinking about Attributes: Object Diagrams

At this point, p is just an object with no attributes.
After these two lines, \( p \) has attributes \( x \) and \( y \).

```
1 class Point:
2     '''Represents a 2-d point.'''
3
4 p = Point()
5 p.x = 3.0
6 p.y = 4.0
```
After these two lines, \( p \) has attributes \( x \) and \( y \).

So dot notation \( p.x \), essentially says, look inside the object \( p \) and find the attribute \( x \).
Objects can have other objects as their attributes. We often call the attribute object embedded.

class Point:
  '''Represents a 2-d point.'''
class Rectangle:
  '''Represents a rectangle whose sides are parallel to the x and y axes. Specified by its upper-left corner, height, and width.'''
P = Point(); p.x = 3.0; p.y = 4.0
r = Rectangle()
r.corner = p
r.height = 5.0
r.width = 12.0
Nesting Objects

Both of these blocks of code create equivalent `Rectangle` objects.

Note here that instead of creating a point and then embedding it, we embed a `Point` object and then populate its attributes.
Objects are mutable

```python
pl = Point(); pl.x = 3.0; pl.y = 4.0
r1 = Rectangle()
r1.corner = pl
r1.height = 5.0; r1.width = 12.0
r1.height = 2*r1.height

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dx

shift_rectangle(r1, 2, 3)
(rl.corner.x, rl.corner.y)
```

If my `Rectangle` object were immutable, this line would be an error, because I'm making an assignment.

Since objects are mutable, I can change attributes of an object inside a function and those changes remain in the object in the `__main__` namespace.
Functions can return objects. Note that this function is implicitly assuming that `rdouble` has the attributes `corner`, `height` and `width`. We will see how to do this soon.

The function creates a new `Rectangle` and returns it. Note that it doesn’t change the attributes of its argument.
Copy and Aliasing

Recall that aliasing is when two or more variables have the same referent, i.e., when two variables are identical. Aliasing can often cause unexpected problems. **Solution:** make copy of object; variables equivalent, but not identical.

```python
pl = Point(); pl.x = 3.0; pl.y = 4.0
import copy
p2 = copy.copy(pl)
p1 is p2
```

False

The `copy` module provides functions for copying objects. P2 is a copy of p1, so they should **not** be identical...

```python
p1 == p2
```

False

...but they **should** be equivalent.
Copying and Aliasing

Recall that aliasing is when two or more variables have the same referent i.e., when two variables are identical.

Aliasing can often cause unexpected problems.

**Solution:** make copy of object; variables equivalent, but not identical.

```python
p1 = Point(); p1.x = 3.0; p1.y = 4.0
import copy
p2 = copy.copy(p1)
p1 is p2
```

False

Hey, those were supposed to be equivalent! What’s up with that? **Answer:** by default, for programmer-defined types, `==` and `is` are the same. It’s up to you, the programmer, to tell Python how to tell if two objects are equivalent, by defining a method `object.__eq__`. We’ll come back to this.

```python
p1 == p2
```

False

Documentation for the *copy* module: [https://docs.python.org/3/library/copy.html](https://docs.python.org/3/library/copy.html)
Copying and Aliasing

Here we construct a Rectangle, and then copy it. Expected behavior is that mutable attributes should not be identical, and yet...

...evidently our copied objects still have attributes that are identical.

```python
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r2 = copy.copy(r1)

r1.corner is r2.corner
```

True
By default, `copy.copy` only copies the “top level” of attributes. This is a problem if, for example, we have a method like `shift_rectangle` that changes the `corner` attribute. Calling `shift_rectangle(r1)` would also change the `corner` attribute of `r2`. 
Copying and Aliasing

```
pl = Point(); pl.x = 3.0; pl.y = 4.0
r1 = Rectangle()
r1.corner = pl
r1.height = 5.0; r1.width = 12.0
r2 = copy.deepcopy(r1)
```

Now when we test for identity we get the expected behavior. Python has created a copy of `r1.corner`.

`copy.deepcopy` is a recursive version of `copy.copy`. So it recursively makes copies of all attributes, and their attributes and so on.

We often refer to `copy.copy` as a shallow copy in contrast to `copy.deepcopy`.

`copy.deepcopy` documentation explains how the copying operation is carried out: https://docs.python.org/3/library/copy.html#copy.deepcopy
Pure functions vs modifiers

A **pure function** is a function that returns an object ...and **does not** modify any of its arguments

A **modifier** is a function that changes attributes of one or more of its arguments

```python
# double_sides is a pure function. It creates a new object and returns it, without changing the attributes of its argument.
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

# shift_rectangle changes the attributes of its argument rec, so it is a modifier. We say that rec has side effects, in that it causes changes outside its scope.
def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dy
```

Pure functions vs modifiers

Why should one prefer one over the other?

Pure functions
 Are often easier to debug and verify (i.e., check correctness)
https://en.wikipedia.org/wiki/Formal_verification
Common in functional programming

Modifiers
 Often faster and more efficient
Common in object-oriented programming
Modifiers vs Methods

A modifier is a function that changes attributes of its arguments.

A method is like a function, but it is provided by an object.

Define a class representing a 24-hour time.

```python
class Time:
    """Represents time on a 24 hour clock. Attributes: int hours, int mins, int secs""

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time()
t.hours=12; t.mins=34; t.secs=56
t.print_time()
```

Class supports a method called print_time, which prints a string representation of the time.

Every method must include self as its first argument. The idea is that the object is, in some sense, the object on which the method is being called.
int_to_time is a pure function that creates and returns a new Time object.

Time.time_to_int is a method, but it is still a pure function in that it has no side effects.
More on Modifiers

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

def incrementPure(self, seconds):
    '''Return new Time object representing this time
    incremented by the given number of seconds.''
    t = Time()
    t = int_to_time(self.time_to_int() + seconds)
    return t

def incrementModifier(self, seconds):
    '''Increment this time by the given
    number of seconds.''
    (mins, self.secs) = divmod(self.secs+seconds, 60)
    (hours, self.mins) = divmod(self.mins+mins, 60)
    self.hours = (self.hours + hours) % 24

t1 = int_to_time(1234)
t1.increment_modifier(1111)
t1.time_to_int()
```

I cropped out `time_to_int` and `print_time` for space.

Two different versions of the same operation. One is a pure function (pure method?), that does not change attributes of the caller. The second method is a modifier.

The modifier method does indeed change the attributes of the caller.
More on Modifiers

Here's an error you may encounter.

How the heck did `increment_pure` get 3 arguments?!

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

def time_to_int(self):
    return (self.secs + 60*self.mins + 3600*self.hours)

def print_time(self):
    print("%2d:%2d:%2d" % (self.hours, self.mins, self.secs))

def increment_pure(self, seconds):
    '''Return new Time object representing this time
    incremented by the given number of seconds.''
    t = Time()
    t = int_to_time(self.time_to_int() + seconds)
    return t

t1.increment_pure(100, 200)
```

Answer: the caller is considered an argument (because of `self`)!
Readings (this lecture)

Required:
- Downey Chapters 15, 16
- Python documentation on classes (only through section 9.3):
  [https://docs.python.org/3/tutorial/classes.html](https://docs.python.org/3/tutorial/classes.html)
- Python documentation on copy module
  [https://docs.python.org/3/library/copy.html](https://docs.python.org/3/library/copy.html)

Recommended:
Readings (next lecture)

Required:
  Downey Chapters 17 and 18

Recommended:
  Python documentation on operators
    https://docs.python.org/3/reference/datamodel.html#specialnames
  Coding style guides
    https://google.github.io/styleguide/pyguide.html
    https://www.python.org/dev/peps/pep-0008/