ICS 32 Winter 2014
Code Example: Classes

Background

You've seen previously how to use *namedtuples*, which is one way in Python to collect heterogeneous data into a single object. Each piece of data is called a *field* and the fields are identified by their names. You can access the value of each field, but you can't change the value of any fields; instead, if you want to change a field, you build a new namedtuple with the one field's value replaced with a different one. Beyond that, namedtuples don't really know how to do much of anything; they're useful, but limited to the problem of bringing a few pieces of data together into the same object.

Namedtuples are handy, but when you think about the types of objects that are built into the Python language and its standard library, like strings, you realize that they don't just *store* things, like namedtuples do; they can also *do* things. For example, consider this interaction with a Python interpreter.

```
>>> s = 'Boo is happy today'
>>> s.upper()
'BOO IS HAPPY TODAY'
>>> s.split(' ')
['Boo', 'is', 'happy', 'today']
```

After creating a string and storing it in the variable **s**, we then ask that string to do a couple of things for us:

- *Hey, **s**, what would you look like if all your letters were uppercase?*
- *Hey, **s**, split yourself into a list of words, by using spaces to delineate where one word ends and another begins, and give me back the resulting list.*

The syntax **s.upper()** is called a *method call* on the object **s** and **upper()** is called a *method*. Since the type of **s** is **str**, and since **upper()** is a method belonging to the **str** type (or, more properly, the **str** *class*), we can call the method **upper()** on **s**, in which case we're asking **s** for an uppercase version of itself. However, we can't ask an integer, a float, a list, or a socket to do the same thing; there is no **upper()** method in these classes, because it wouldn't make any sense. Different classes support different methods that are specific to their own types of objects.

Seeing that strings can *do* things, in addition to just *storing* things, the natural next step might be for us to wonder why we can't create objects that are endowed similarly; why can't our objects be smarter and know how to do things for themselves? The answer is that they can. We just have to know how to define these kinds of objects in Python.

Classes

In Python, a *class* is a blueprint for a new kind of object. That blueprint specifies what objects of the class can do, by defining a set of *methods*, each representing one operation these objects are

capable of performing. For example, the **str** class, which is built into Python, contains a number of methods we've seen, like **upper()**, **split()**, and **startswith()**.

We define a new class by using the **class** construct in Python. The simplest kind of class is one that doesn't know how to do much of anything (aside from a few built-in things that all objects can do), which we could write this way.

```
class MyNewKindOfObject:
    pass
```

As in **if** statements and loops, **pass** is used to specify that a class is empty. But an empty class isn't a very interesting one; let's instead write a class that does something simple. First, though, we need to know a little bit about the mechanics of method calls in Python, which aren't quite as they seem.

Method calls in Python

When you make a method call in Python, there's more going on than meets the eye. For example, consider this method call that we made previously:

```
>>> s.split(' ')
['Boo', 'is', 'happy', 'today']
```

If I were to ask you "How many parameters are being passed to this method?", you might well be tempted to answer "One." However, methods aren't quite like functions; they get called *on* an object (i.e., they're something that you're asking some particular object to do for you). In this case, that object is **s**. In Python, there is an alternative syntax for calling methods, which we don't use often when we write programs because it's not especially clear, but which demonstrates why the answer to the question above isn't "One."

```
>>> str.split(s, ' ')
['Boo', 'is', 'happy', 'today']
```

This alternative syntax is written by first specifying the name of the *class* containing the method (in this case, the **str** class, which is the class from which string objects are built), followed by a dot, followed by the method's name, and followed by its parameters. Notice that the first parameter listed is **s**, the object on which we wanted to call the method.

Whenever we write method calls in the clearer and more familiar form **s.split(' ')**, Python essentially translates these calls behind the scenes to the longer form **str.split(s, ' ')**. While that sounds like a ticky-tack detail that isn't worth knowing — because writing the long-form method calls is generally not a good idea, because it's more verbose and less clear — it's a critical detail if you want to understand how to write methods, as we'll see; methods need what seems like an "extra" parameter, usually called **self**, which represents the object that the method is called on.

Constructing an object

If a class is a blueprint for a kind of object, it stands to reason that there must be some way to create an object from that blueprint. In Python, that is called *constructing* an object, and we do it by calling a class' *constructor*, which we do by calling a function whose name is the name of the class:

```
>>> x = MyNewKindOfObject()
```

After this statement executes, **x** will refer to a MyNewKindOfObject, an object constructed from our

MyNewKindOfObject class.

As we'll see, it's also possible to require a constructor to take parameters, and we can write code that is executed when an object is initialized and uses those parameters during the initialization.

Objects and their attributes

All objects in Python have a *class*, which specifies what kind of object they are. Additionally, all objects in Python have a collection of *attributes*, which are the information that they're storing at any given time. Assigning a value to an attribute is as simple as any other assignment statement, except that we use the "dot" operator to qualify which object we'd like to store the attribute into. For example, given the **MyNewKindOfObject** object we created earlier, we can store values in its attributes in the interpreter directly, and then read them back later.

```
>>> x.a1 = 3
>>> x.a2 = 4
>>> x.a3 = 5
>>> x.a1 + x.a2 + x.a3
12
```

As a general rule, we rarely assign to an object's attributes willy-nilly like this; instead, we assign values into attributes within the class, initializing them when the object is constructed, and changing them only within methods whenever they need to be changed. But the syntax here is important: you assign to an object's attribute by saying ***object_name.attribute_name = value***.

(Note: Some classes are written in a way that makes this kind of open-ended assignment to its objects' attributes impossible. For example, if you tried to do this with a string, an **AttributeError** would be raised.)

Writing a "counter" class

Next, we'll write a class that implements a "counter," whose role is to count how many times it's been asked for its count in its lifetime. Initially, the count is zero; each time it's asked for its count, that count is incremented, so it grows over time. It also allows you to reset its count, if you'd like. You could imagine using a class like this, for example, to implement a "hit counter" on a web page that keeps track of how many times the web page had been visited.

Before we spend too much time writing our class, we'll envision how we'd like it to work. Let's say that this is what we decide we want:

```
>>> c1 = Counter()
>>> c1.count()
1
>>> c1.count()
2
>>> c1.count()
3
>>> c2 = Counter()
>>> c2.count()
1
>>> c1.count()
4
>>> c1.reset()
```

```
>>> c1.count()
1
>>> c2.count()
2
```

So we want to be able to construct a Counter object and pass no parameters to it; its count would be initialized to zero. And we want a **count()** method, which increments and returns the count belonging to that Counter. Note in the example above how separate Counter objects have separate counts; attributes belong to *objects*, as opposed to *classes*.

Below is the complete Counter class. It consists of two methods:

- A special initializer method called **__init__**, which is called whenever an object of this class is being initialized just after it's been created.
- A **count()** method, which increments and returns the count.

```
class Counter:
    def __init__(self):
        self._count = 0

    def count(self) -> int:
        self._count += 1
        return self._count

    def reset(self) -> None:
        self._count = 0
```

You'll notice a few things here:

- Writing a method in a class is a lot like writing a function; the **def** construct is used. But there's an interesting difference: even though I didn't want these methods to accept any parameters, I listed a parameter called **self**. If you're not sure why, re-read the section titled *Method calls in Python* above and then come back here. The reason is that when you make calls to methods, an extra parameter is added before all the others; that parameter is the object the method was called on. By convention, we call that parameter **self**.
- When we wanted to assign a value to an attribute, we did so the same way we saw previously: specify the object whose attribute we want to assign, followed by a dot, followed by the name of the attribute. **self._count** means "The **_count** attribute belonging to the object **self**." Remember that **self** is the object the method was called on. So, for example, the **__init__** function says "When an object is being initialized, set the **_count** attribute *belonging to the object being initialized* to zero."
- Recall that we've been using a single underscore in front of the name of a function in a module to specify that it is intended to be *private* (i.e., it's a detail of how the module was implemented, but not one that other code using that module needs to know or use). We can do the same with attributes and methods in classes. Nothing stops someone from assigning a new value into the **_count** attribute belonging to a Counter object, but the underscore is at least a hint that we intend for the attribute to be private (i.e., visible to the Counter object, but not visible outside of it).
    - The value in keeping attributes private (and sometimes methods, too!) is in centralizing knowledge about a program's inner workings in as few places as possible. This makes a program easier to understand, because you don't have as far to look when you want to understand how something works, because you don't have constraints spread

throughout a large program, and because you can make a change in one part of a program without a cascading set of changes affecting everything else in the program. Keeping separate things separate is one of the hallmarks of good software engineering; we'll focus on this repeatedly in this course and again in ICS 33.

The code

Below are the Counter and Person classes that we wrote in lecture.

- **counter.py**
- **person.py**