Anatomy of a python class

This serves as a short explanation of the inner workings of python classes, using Ethan's class `ConvolvedContinuousAndDiscrete` as an example. Note that this serves to get you familiarized with the basic, general structure of classes. This probably is not enough information for you to write your own classes; if you are interested in learning more detail, see the tutorial linked to on piazza or come see me in office hours.

```
from scipy.stats import distributions as iid

# Code to convolve a random variable with a pmf and another having a cdf
# Exploits =scipy.stats= base rv_continuous class.

class ConvolvedContinuousAndDiscrete(iid.rv_continuous):  1

    """Convolve (add) a continuous rv x and a discrete rv s,
       returning the resulting cdf."""

    def __init__(self,x,s):  2
        self.continuous_rv = x
        self.discrete_rv = s  3
     4  super(ConvolvedContinuousAndDiscrete, self).__init__(name="ConvolvedContinuousAndDiscrete")

    def _cdf(self,z):  5
        F = 0
        s = self.discrete_rv
        x = self.continuous_rv

        for k in range(len(s.xk)):
            F = F + x.cdf(z-s.xk[k])*s.pk[k]
        return F

    def _pdf(self,z):
        f = 0
        s = self.discrete_rv
        x = self.continuous_rv

        for k in range(len(s.xk)):
            f = f + x.pdf(z-s.xk[k])*s.pk[k]
        return f
```

1. **Definition of the class.** Here, we are doing a couple of things:
    a. Define a class with the `class` command
    b. Name the class `ConvolvedContinuousAndDiscrete` (class names should be uppercase to follow typical standards).
    c. Finally, this class is doing something special - it is *inheriting* from a class that already exists: `rv_continuous` from the `scipy.stats` module, which is why we call this class in parentheses. I won't get into too much detail about inheritance, but this really cool property allows us to build off a structure that already exists (here is a simple example of inheritance). So in this case, we are defining a class for convolving a discrete and a continuous random variable, and

we are building this off of a class for continuous random variables, since we know that the convolution will be continuous.

2. **Constructor.** the `__init__()` function is what is known as a *constructor* in object-oriented programming; it is the method that is called whenever we want to *instantiate*[1], or create a new, specific instance, of a class. In the constructor, we call *self* (don't worry too much about that), and we also call the **inputs** (in this case, `x, s`) that the class must take to be defined. In this case, `x, s` are discrete and continuous random variables, so we create a convolution by taking as inputs the discrete and continuous RV's we want to convolve.
3. **Attributes**. Attributes are exactly what they sound like; they are attributes that define or belong to the class (to signal that they belong to the class, we define them with `self.attributeName = ...`). We can make anything that makes sense to us an attribute of the class.
    a. You can google any class that belongs to a package to see its attributes; for example, check out this page to see the attributes of the `iid.discrete_rv` class (there called parameters). Note that a lot of these attributes have default values, which is why we don't have to specify all of them every time we want to make a discrete random variable. Furthermore, some of these will be filled in when we instantiate the class! To see this in action, open the random_variables0 notebook, and after defining the discrete random variable `s`, type `s.a` We can see that `a` is an attribute of the class from the webpage, and even though we didn't specify it, it was defined in the instantiation of the class, so it is callable!
    b. Note from the above that since attributes belong to the class, we can retrieve (call) them by typing `instanceName.attributeName`
4. **Super**. This forms another part of the inheritance syntax; we are asking the constructor to pull attributes and methods from the "parent" class.
5. **Methods**. Similarly to attributes, methods belong to a class but unlike attributes, they *do things.*
    a. Here, we put an underscore before the name because we are actually overwriting methods that are already named/exist in the parent class, `rv_continuous`. We can find the methods that are overwritable here. We won't need to later call the method with the underscore, the underscore is not part of the name and only serves to say that we are overwriting previously defined methods (to see someone do this incorrectly and be corrected, see my answer to the Convolutions thread on piazza! Can you understand what I did wrong?).
    b. We can also add totally new methods to any class! To do this, just define a method inside the class (no underscore needed).
    c. To call methods, as you can see later in the random_variables0 notebook, it is again similar to attributes: `instanceName.methodName(input1, ...)`

[1] Note that classes are inherently abstract; they have general attributes and methods, but it is not until we create an *instance* of the class with *specific* values for these attributes that the class can really do things. This is the essence of instantiation; we want to create a specifically defined version of the class. For example, a normal random variable class has the abstract attribute of a mean. It is not until we *instantiate* a normal RV with mean 0 that that abstract attribute has specific meaning.