Questions to be asked before designing classes:

- **What is the data that you want to deal with?** (Data about a bunch of songs from iTunes? Data about a bunch of tweets from Twitter? Data about a bunch of hashtag searches on Twitter? Two numbers that represent coordinates of a point on a 2-dimensional plane?)
- **What will one instance of your class represent?** In other words, which sort of new *thing* in your program should have fancy functionality? One song? One hashtag? One tweet? One point? The answer to this question should help you decide what to call the class you define.
- **What information should each instance have as instance variables?** This is related to what an instance represents. See if you can make it into a sentence. *"Each instance represents one < song > and each < song > has an < artist > and a < title > as instance variables."* Or, *"Each instance represents a < Tweet > and each < Tweet > has a < user (who posted it) > and < a message content string > as instance variables."*
- **What instance methods should each instance have?** What should each instance be able to *do*? To continue using the same examples: Maybe each song has a method that uses a lyrics API to get a long string of its lyrics. Maybe each song has a method that returns a string of its artist's name. Or for a tweet, maybe each tweet has a method that returns the length of the tweet's message. (Go wild!)
- **What should the printed version of an instance look like?** (This question will help you determine how to write the __str__ method.) Maybe, "Each song printed out will show the song title and the artist's name." or "Each Tweet printed out will show the username of the person who posted it and the message content of the tweet."

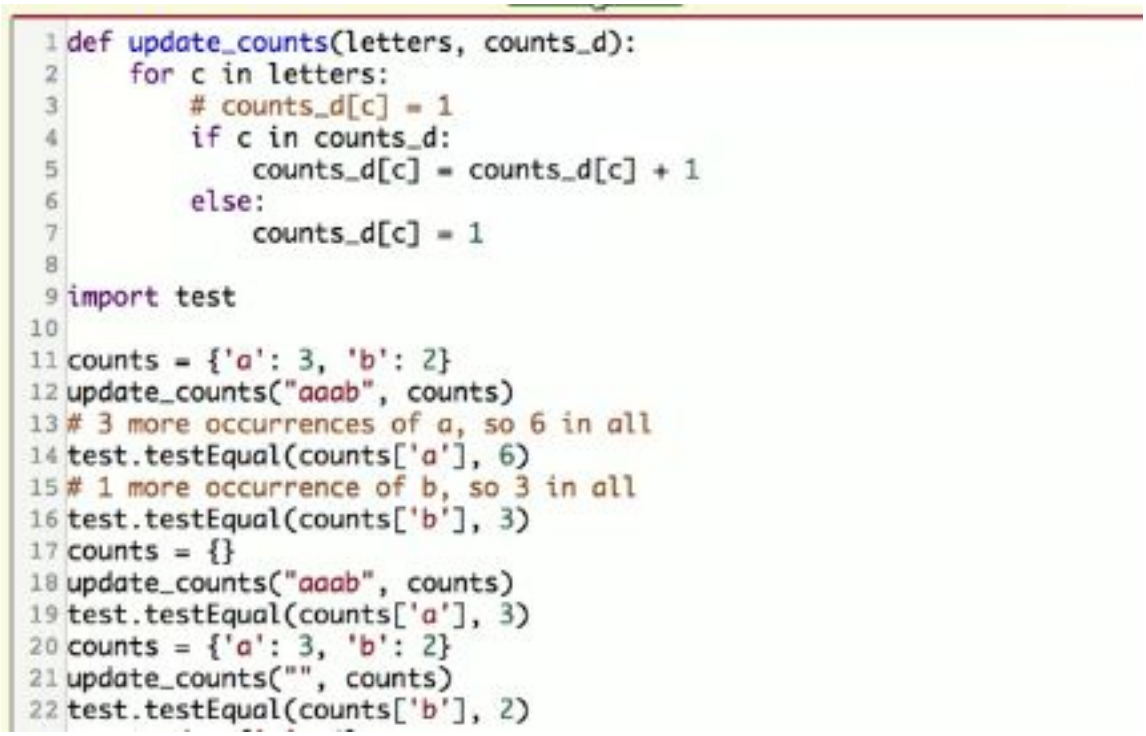
Unit Tests

There are two types of tests:

- Return value tests: Tests that base their output on values returned by what they check. To put it simply, these tests are written when a function is supposed to do some computation on the inputs it receives and then produce an output. E.g.:

```
1 def square(x):
2     return x*x
3
4 import test
5
6 test.testEqual(square(3), 9)
7
```

- Side effect tests: Tests that base their output to see if any side effects of the code you've written have occurred e.g. list has been mutated, variable values have been changed etc. E.g.:



```
def update_counts(letters, counts_d):
    for c in letters:
        # counts_d[c] = 1
        if c in counts_d:
            counts_d[c] = counts_d[c] + 1
        else:
            counts_d[c] = 1

import test

counts = {'a': 3, 'b': 2}
update_counts("aaab", counts)
# 3 more occurrences of a, so 6 in all
test.testEqual(counts['a'], 6)
# 1 more occurrence of b, so 3 in all
test.testEqual(counts['b'], 3)
counts = {}
update_counts("aaab", counts)
test.testEqual(counts['a'], 3)
counts = {'a': 3, 'b': 2}
update_counts("", counts)
test.testEqual(counts['b'], 2)
```

One way to think about how to generate edge cases is to think in terms of **equivalence classes** of the different kinds of inputs the function might get. For example, the input to the square function could be either positive or negative. We then choose an input from each of these classes. **It is important to have at least one test for each equivalence class of inputs.**

Semantic errors are often caused by improperly handling the boundaries between equivalence classes. The boundary for this problem is zero. **It is important to have a test at each boundary.**

Another way to think about edge cases is to imagine things that could go wrong in the implementation

The key aspects of the incremental testing process are:

1. Make sure you know what you are trying to accomplish. Then you can write appropriate unit tests.
2. Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
3. Use temporary variables to hold intermediate values so that you can easily inspect and check them.

4. Once the program is working, you might want to consolidate multiple statements into compound expressions, but only do this if it does not make the program more difficult to read.

Exceptions:

An *exception* is a signal that a condition has occurred that can't be easily handled using the normal flow-of-control of a Python program. *Exceptions* are often defined as being "errors" but this is not always the case. All errors in Python are dealt with using *exceptions*, but not all *exceptions* are errors.

```
try:
   <try clause code block>
except <ErrorType>:
   <exception handler code block>
```

The syntax is fairly straightforward. The only tricky part is that after the word except, there can optionally be a specification of the kinds of errors that will be handled. The catchall is the class Exception. If you write except Exception: all runtime errors will be handled. If you specify a more restricted class of errors, only those errors will be handled; any other kind of error will still cause the program to stop running and an error message to be printed.

3 kinds of errors:

- Syntactic: Python doesn't run your code at all until it's fixed.
- Runtime: Python runs code until the line that caused the error.
- Semantic: You didn't code correctly and Python will just execute and produce output.

Functionality of try-except can be achieved via if-else i.e. that you don't want the program to crash. Choose if-else when you know the error(s) that can occur and they're few in number. Otherwise, use try-except when you're unsure of what could go wrong with the code.

NOTE: If exception occurs in code that is not within the **try** block, the program will stop executing and give you a runtime error.

Prof. Severence's website on Django tutorial:

www.dj4e.com

Credits:

Notes from **Python Classes and Inheritance** - Coursera, University of Michigan course.