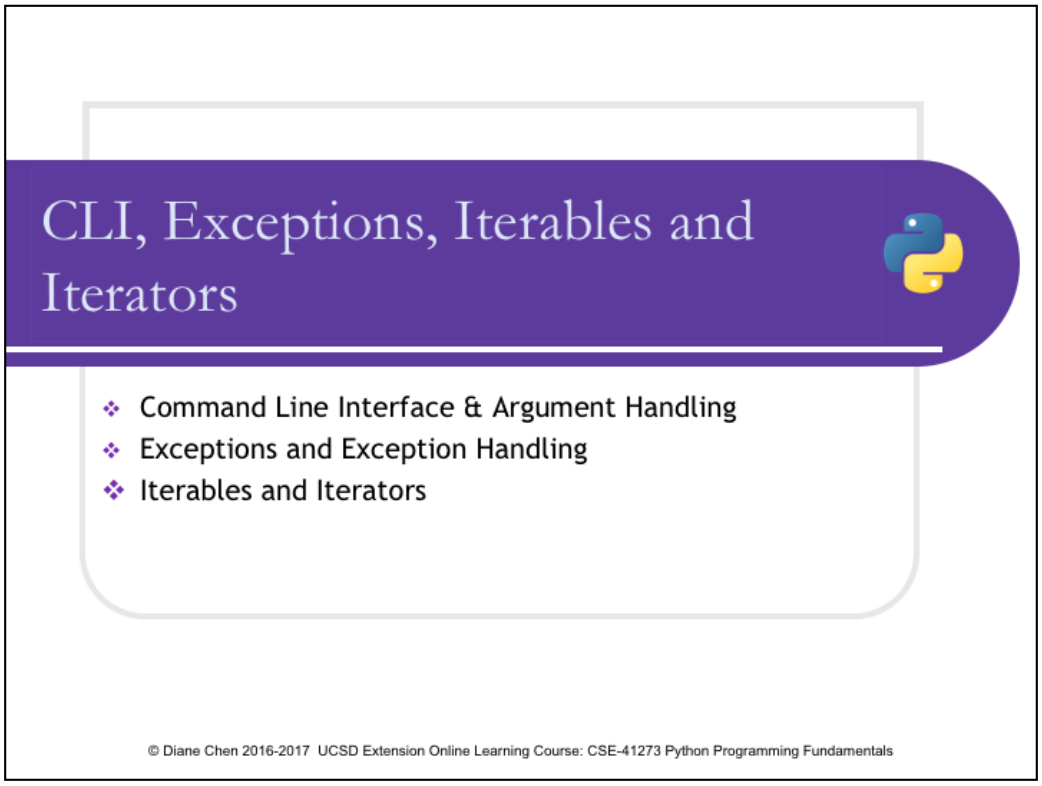


Here are the things we will cover this week: First off, in this lecture, I'll talk about the Command Line Interface; that is, making programs to run from the command line, and handling arguments passed to the program on the command line. Next I'll cover Exceptions and how to handle them.

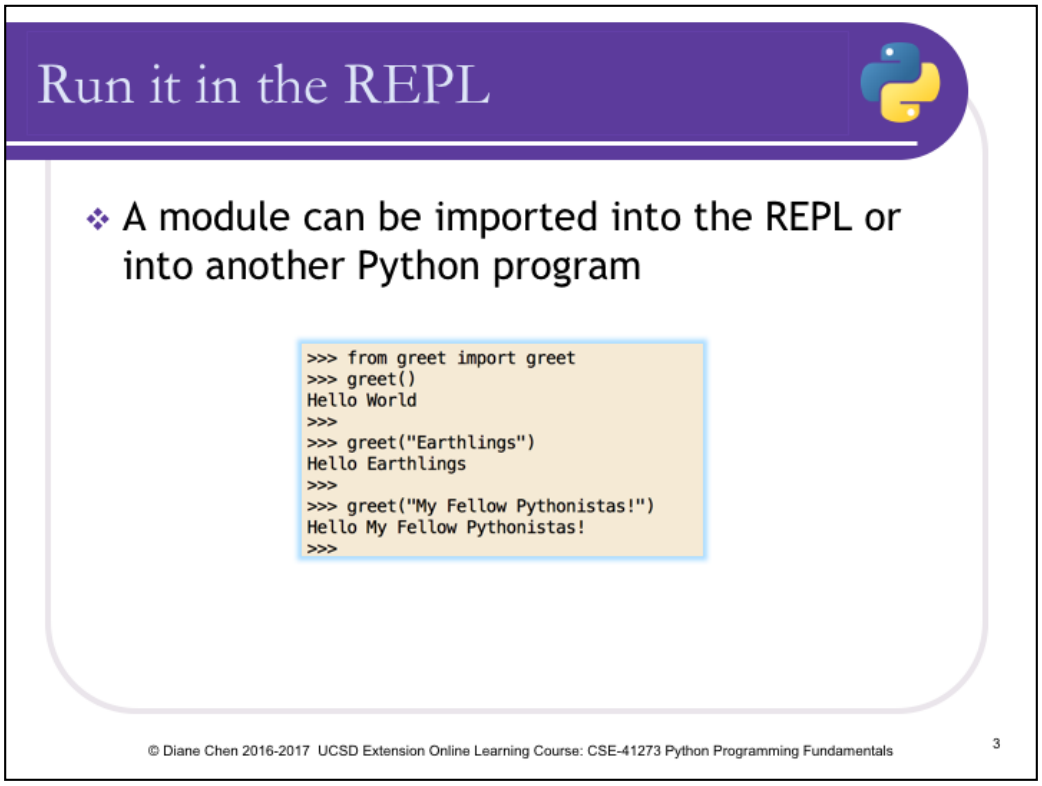
Finally, we'll also see more about the concepts of Iterables and Iterators; what they are, what are their differences and how to use them.

Our Python Module

- Create a file `greet.py` with the following code

```
""" Greet module contains function greet()"""

def greet(text="World"):
    """ Prints 'Hello World' (default) or 'Hello {text}' """
    print("Hello {}".format(text))
```

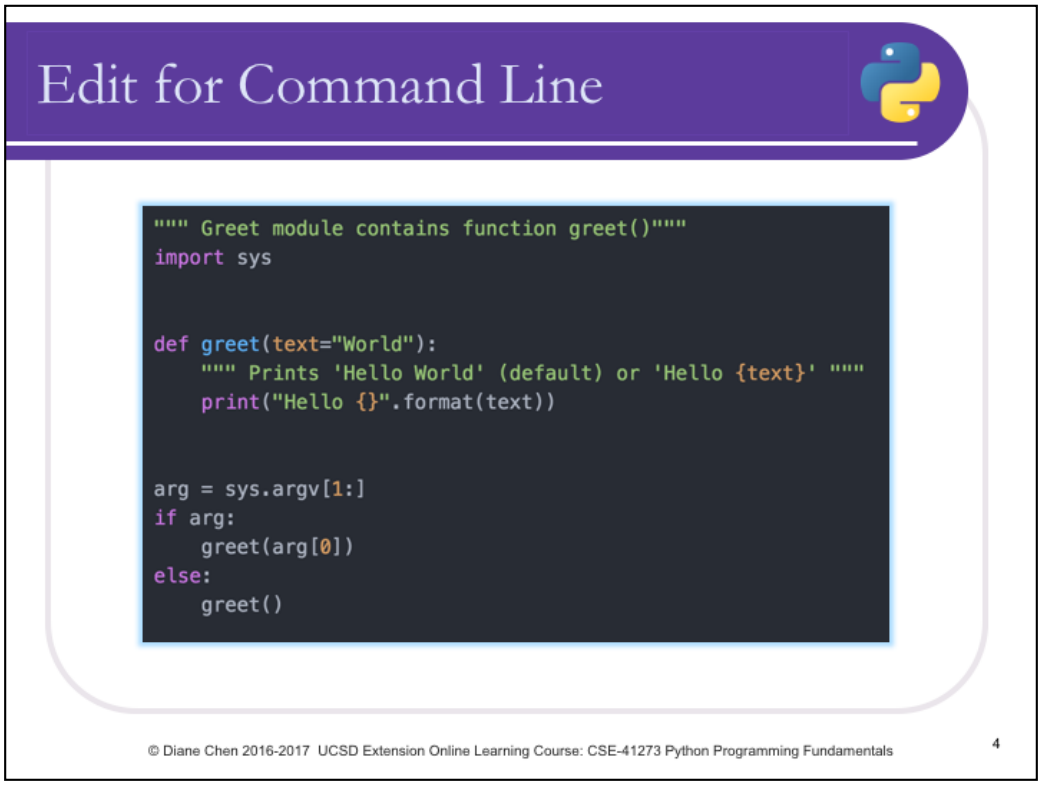

Let's start with the module greet.py that we created last lesson. It has one function, greet, that takes one optional argument of text to print following the word "hello". If no argument is given then it prints Hello world. If you didn't create the file already, you can pause the video to go to your editor and create this file by typing in these lines of code. Be sure to save the file to your python class folder. Don't forget to activate your virtual environment before running Python.



Let's import the greet module into the REPL. Again, be sure to be in your python_class project folder with your virtual environment activated before you start Python. I'm nagging you now so you'll remember in the future!

Enter "from greet import greet" into the REPL.

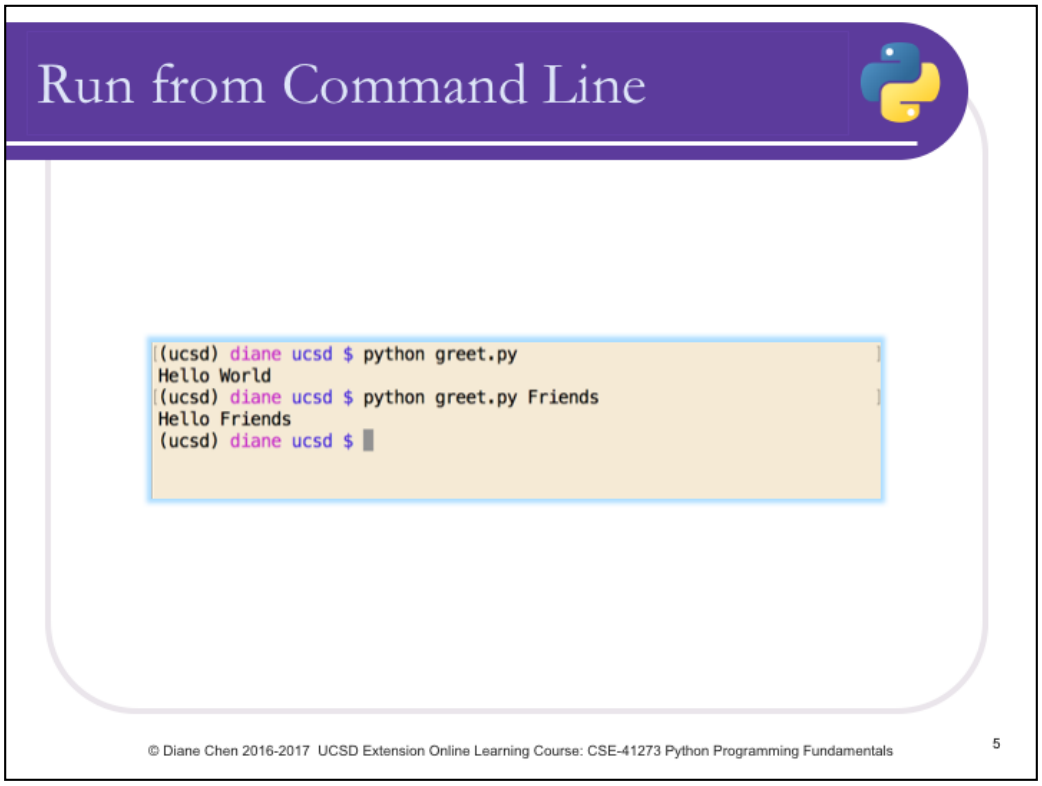
Typing greet() with no arguments causes the function to use the default of "World" and print Hello World. If an argument is given, then that is displayed after the "hello" in the greeting. The argument need only be a string, not just one word.



Now let's change greet.py so we can run it from the command line with an optional argument. We have to add "import sys" to import the system module in order to access the command line arguments.

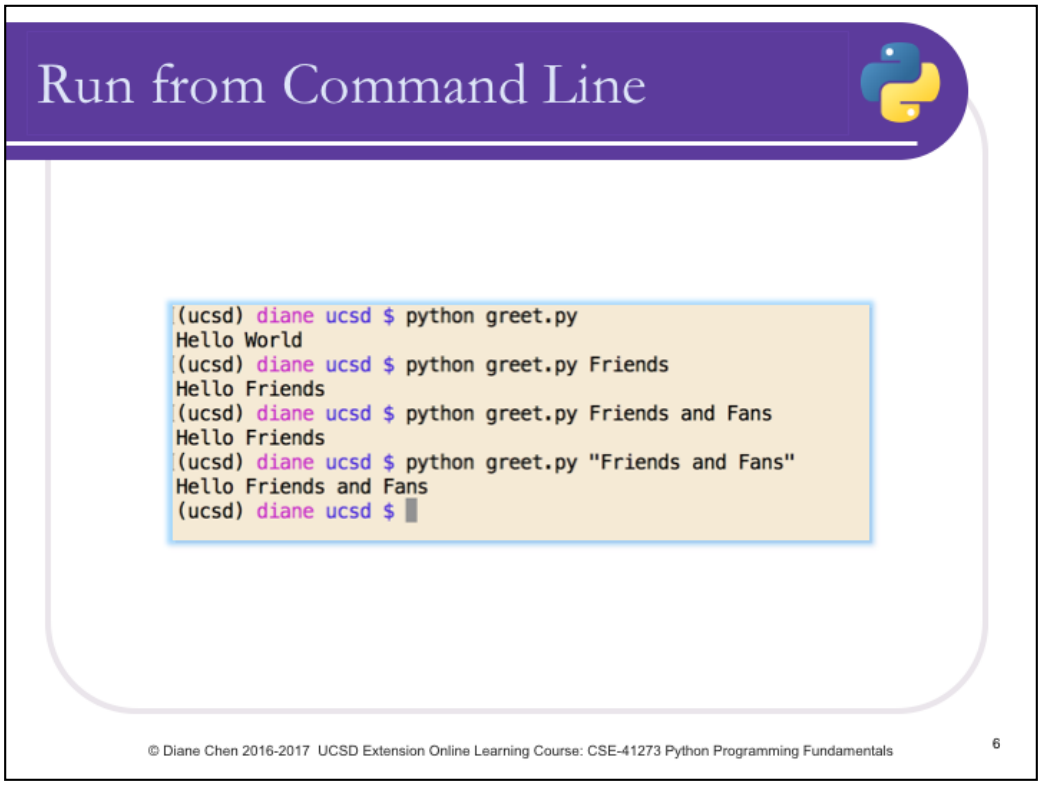
Sys.argv is a list of the command line arguments. The first element, index zero, is the name of the program, so we don't care about it. We can take advantage of one of the cool things about slices, and assign to the variable arg the slice of sys.argv starting at one and going to the end. In this use of a slice of sys.argv, if there is no value at index 1, this does not generate an error. We just get back an empty list. Very handy!

So now, if there is an argument, we use it as the parameter to the greet function, otherwise we just call greet with no argument and let it print out Hello World.

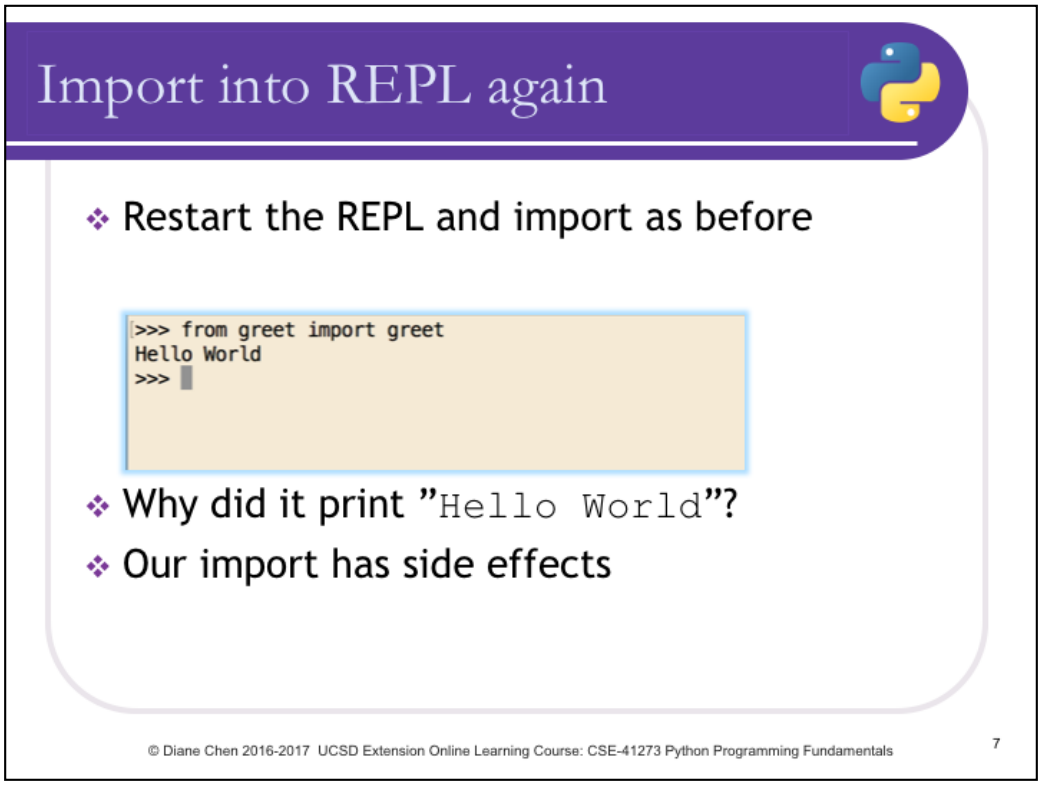


Exit the REPL and run the script on the command line. Here we see that running it from the command line produces the output we would expect. When we run it without any arguments, it prints “hello world”, and when we give it an argument, it prints the argument after the Hello.

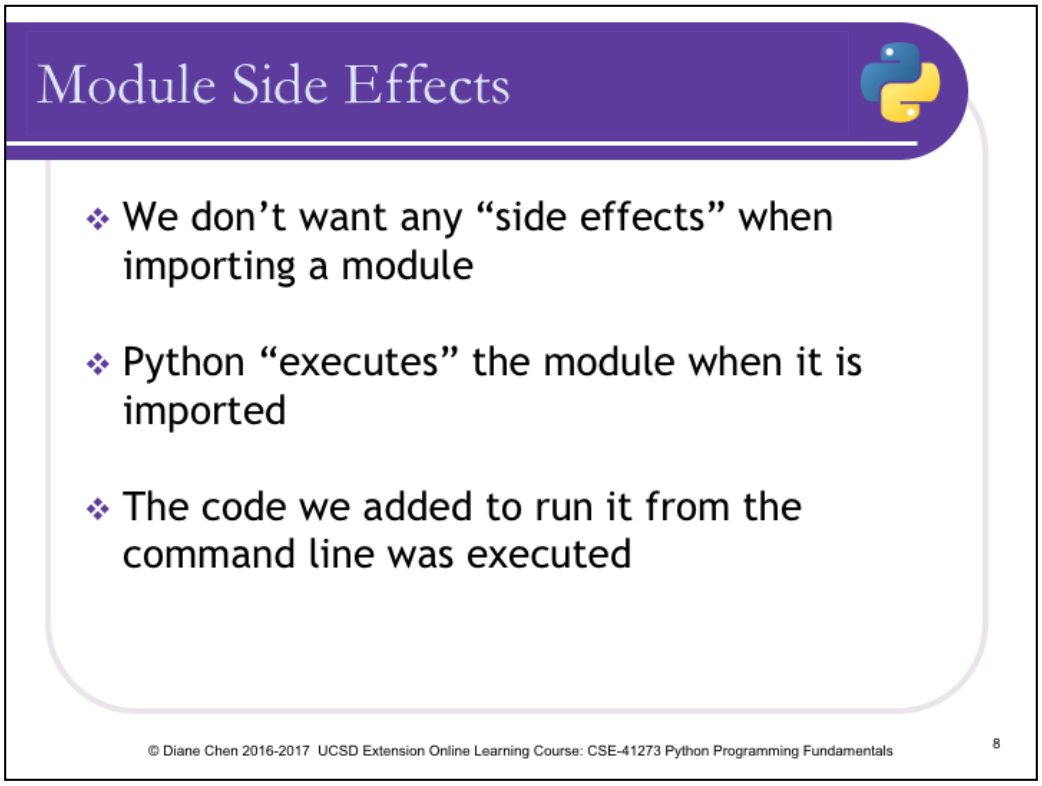
As an aside, I’m using a Mac computer and I have my Terminal windows customized to show my username “diane” in dark pink and the current folder in purple, named ucsd. I also named my virtualenv “ucsd”. I modified the terminal window to have a tan background to make it easier to distinguish where the images come from.



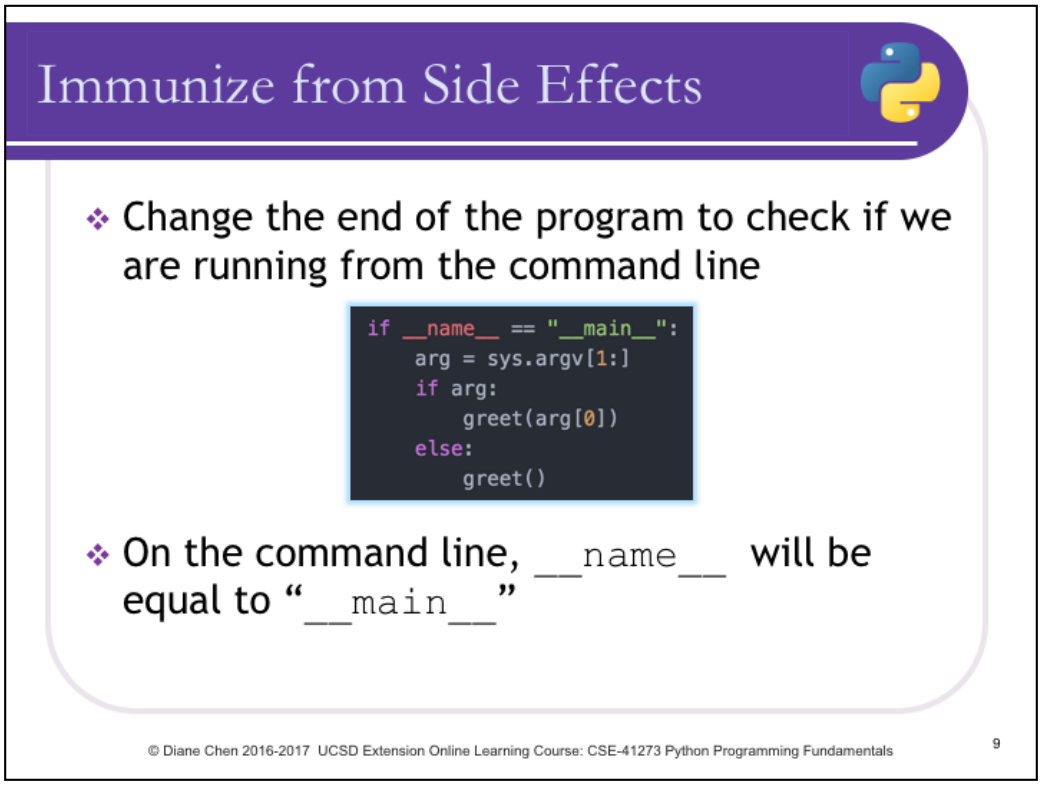
What happens if we give it more than one argument? Our program only looks at the first argument and ignores everything after that. It doesn't cause an error because we didn't put in any code to check for this and raise an error for too many arguments. If you enclose the phrase “Friends and Fans” inside quotes, then Python will consider it one string and therefore one argument.



Now let's test our greet function again in the REPL, since we have made changes to the module and we should always test after changes. We import it in the same way we imported it before. Oh no! It prints out “Hello World” even though we didn’t tell it to! What’s happening here? You might have thought that since we didn't change the function greet(), we don't need to test in the REPL. That kind of thinking leads to doom and despair, as we can see here. The problem is that we didn’t protect our script from side effects of being imported.



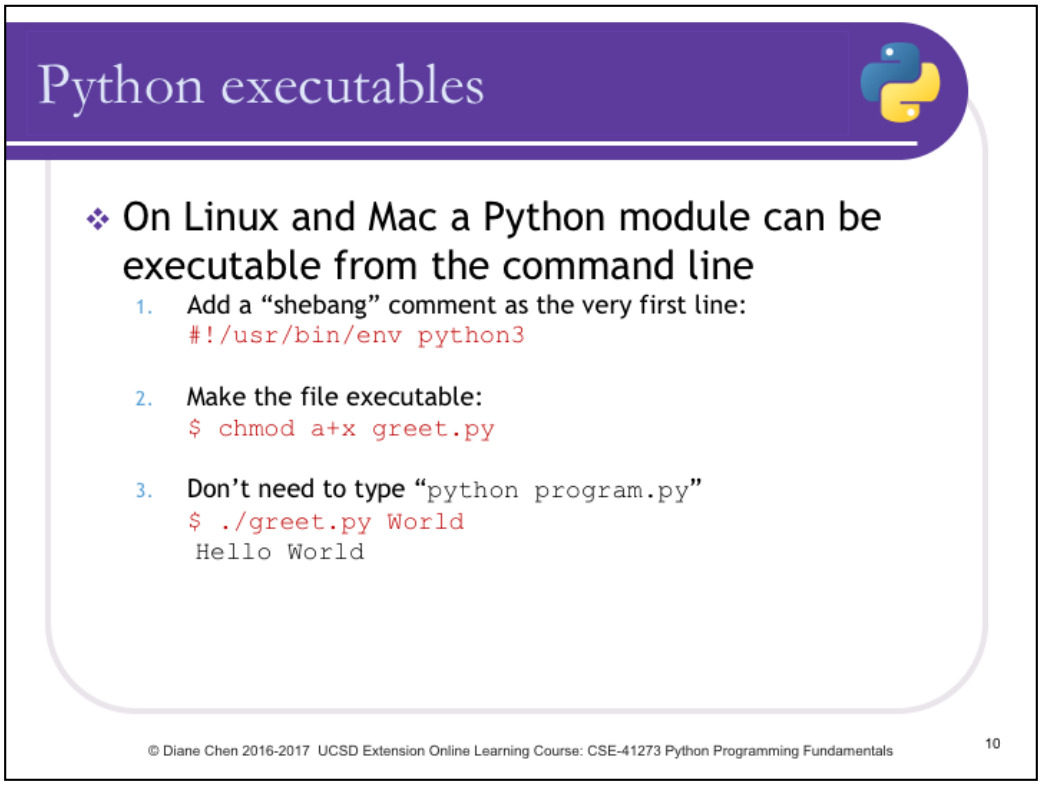
This is a Very Bad Thing. We never want side effects to happen when we import a module. So what is going on? The problem is that when a module is imported, the Python interpreter reads and executes the whole file. It needs to do that to store the definitions of the functions and other things inside the file that we might need to access after importing it. So the code that we added in order to make it a command-line script got executed during the import, even though it was not being called from the command line. How do we handle this seemingly contradictory situation, where we want to be able to import the module and use the function greet, but also be able to run it from the command line?



We need to make another change to the program to protect it from having side effects. Add this code to the end of the file. What are those funny things there with all the underscores? The variable is “name” with two underscores on both sides. We pronounce it as “dunder name” where “dunder” is an abbreviation for “double underscore”, because saying “double underscore name double underscore” is just too much trouble!

When something is named with double underscores, it means it is a special object in Python. Dunder name is an object that exists in every program. When a program is running from the command line, the value of dunder name is the string dunder main, so our block of code here gets executed. When a module is imported into a program or into the REPL and executed by the Python interpreter, its dunder name will be the name of the module instead of dunder main, so the code in the if block will not get executed. We will see lots of dunder variables and things later on in the class. They are very common in Python.

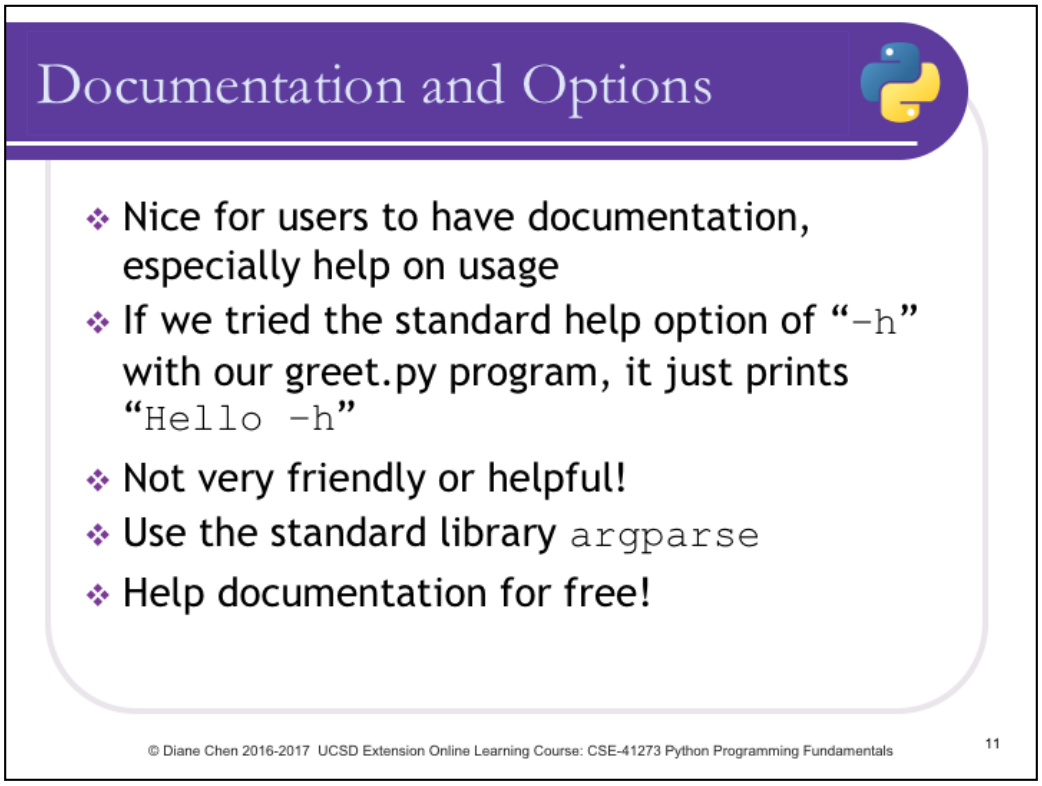
In any case, now we have protected our module from import side effects.



When working on a UNIX-like system such as Linux or Mac, you can make Python programs into executable scripts.

We can do this by adding a special comment to the first line of our file. This is usually referred to as a “shebang” and starts with the comment symbol pound sign, hash symbol, or its true name octothorpe, followed by an exclamation mark and information about the location of the Python interpreter to use. This example will run our code with the default Python 3 when we execute it as a script from the command-line. In order to actually execute our code as a script, we need to make the file executable with the chmod or "change mode" command.

Once that is set up it can be run just with ./greet.py, so it saves a bit of typing. Putting the ./ before the script name forces it to be executed by the system. Without it, the system thinks it is a shell command, rather than a script and it gives an error. I'm not going to use it in this class becaue it doesn't work on Windows, but I wanted to show you that it is possible to do.

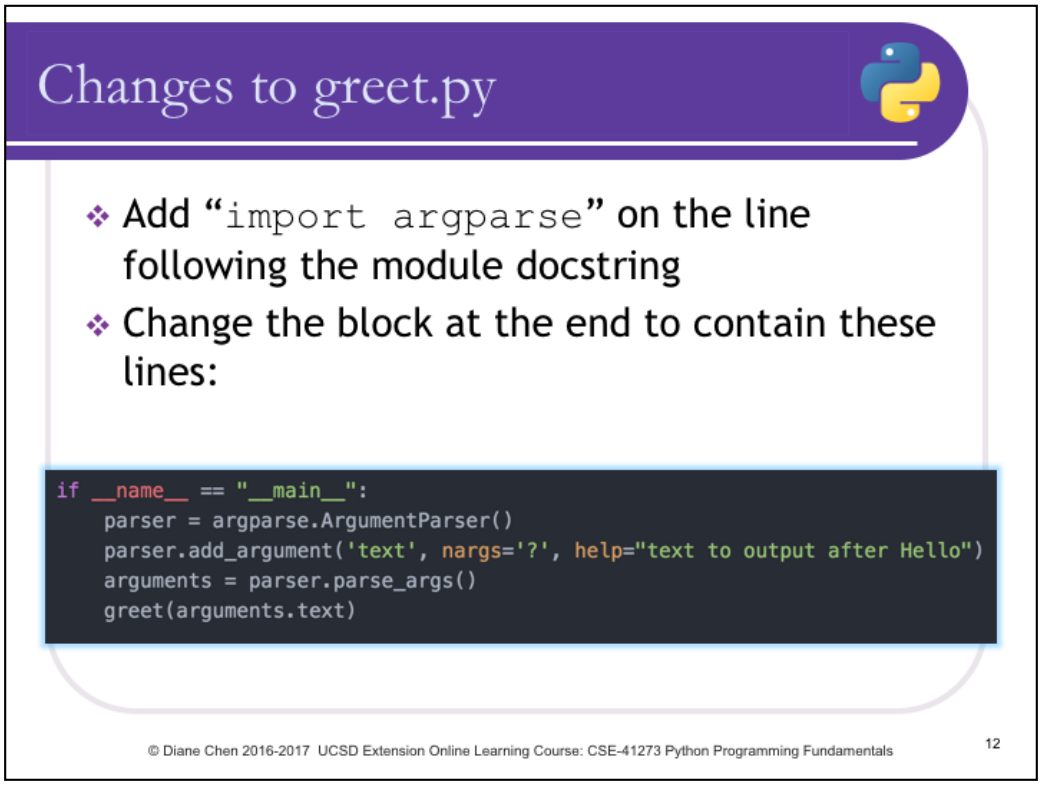


So now we have a program module that can be run from the command line. What about documentation? Wouldn't it be nice for users to be able to ask for help on our program? Maybe they don't remember what the arguments are.

It's pretty standard to be able to type a program name as if you're going to tun it, but give it the "-h" option and it prints out help information. Unfortunately, that won't work with our program because we don't check for "-h" in the argument list. If we type "python greet.py -h" on the command line, it would just print "Hello -h". Not very friendly is it?

We could implement a check to see if the argument passed is '-h' but that's pretty annoying to have to do that for every program. Lots of duplicated code and it would be error prone and easy to introduce copy-paste errors and typos.

Fortunately there is a standard library called argparse, that provides great capabilities for this. You tell it the arguments you want and it makes the docs for you and manages the command line arguments. It automatically compiles help information to display, which is really nice. There are other similar libraries such as docopt and getopt, but we are just going to talk about argparse.



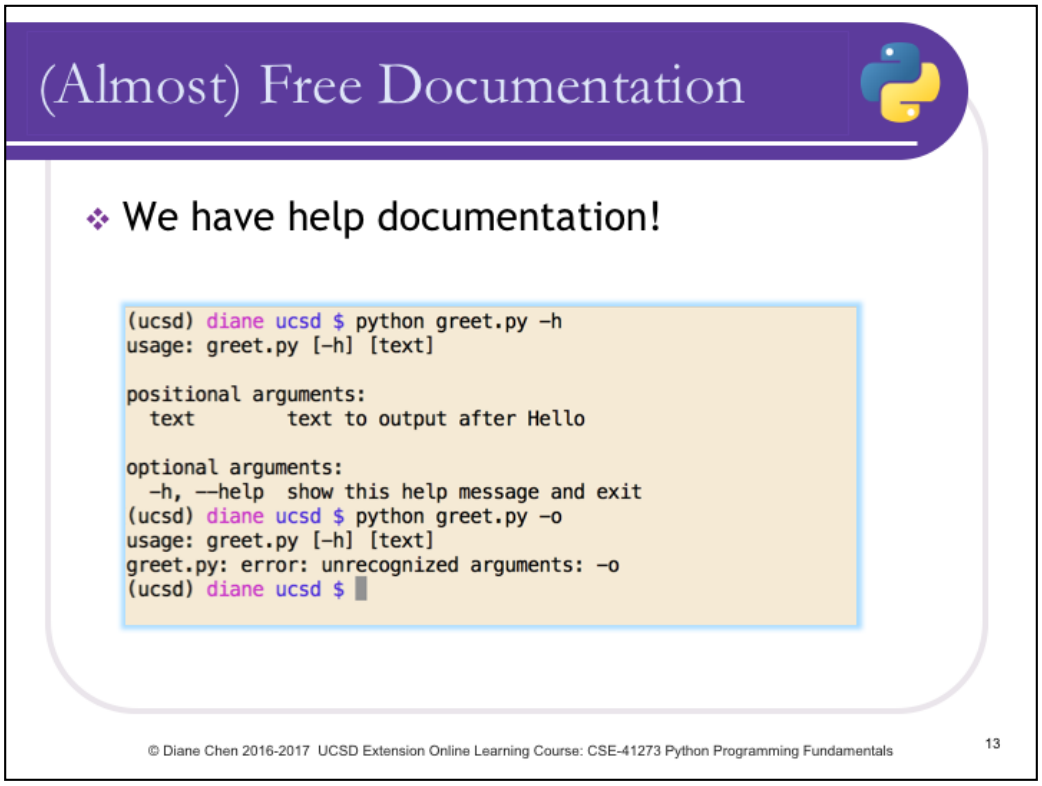
So, let's give argparse a try! Add "import argparse" to the module right after the top docstring. We can also remove the "import sys" line, but it's not critical at this point.

The changes to the block that is executed when it is called from the command line are shown here. The first line of the block is setting the variable "parser" to an ArgumentParser object that is created by argparse.

Then we call add_argument which is a method on the ArgumentParser object. A method is a function that only applies to certain kinds of objects. We will talk more about methods when we talk about classes next week. The parameters to add_argument consist of the string "text", which is the name of the variable that will contain this argument. "nargs" is the number of "text" arguments; the question mark means that we allow 0 or 1 only.

The help parameter is of course so it can build the help docs. This method "registers" the text argument with the ArgumentParser object that we created as "parser". In the next line we call parser.parse_args(). This method gets the command line arguments and processes them according to the instructions we gave in the add_argument call.

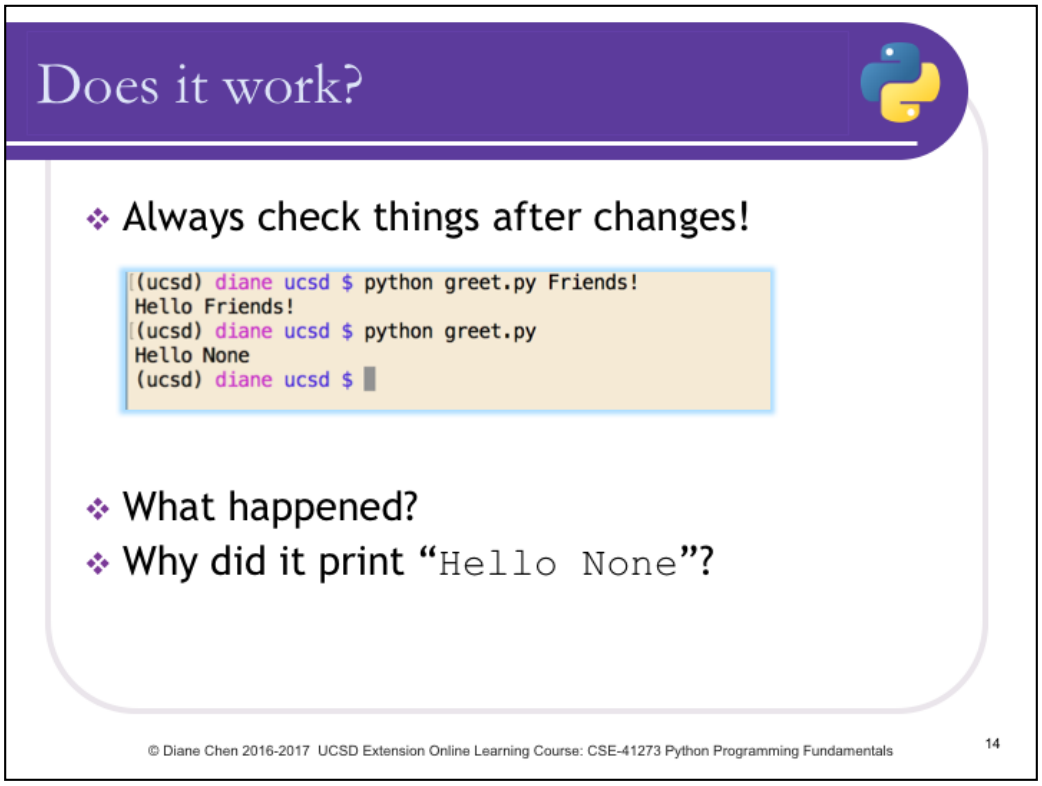
The result is an object that contains the command line argument(s) in object attributes. Because we told it the argument name is "text", the object "arguments" has an attribute named "text" that contains the input. So we just pass arguments.text to the greet function.



Now we have help documentation for our program! With the standard command line option –h, we get usage and help documentation. That's pretty cool!

Note we also get error handling for invalid options. If there is an error it displays the usage line before the error message.

It's practically free – we didn't have to write any special code to handle the help message other than the help parameter "text to output after Hello" when we registered the argument with add_argument. It's very easy and clean, and we get a lot of value for the small effort of using the argparse module.



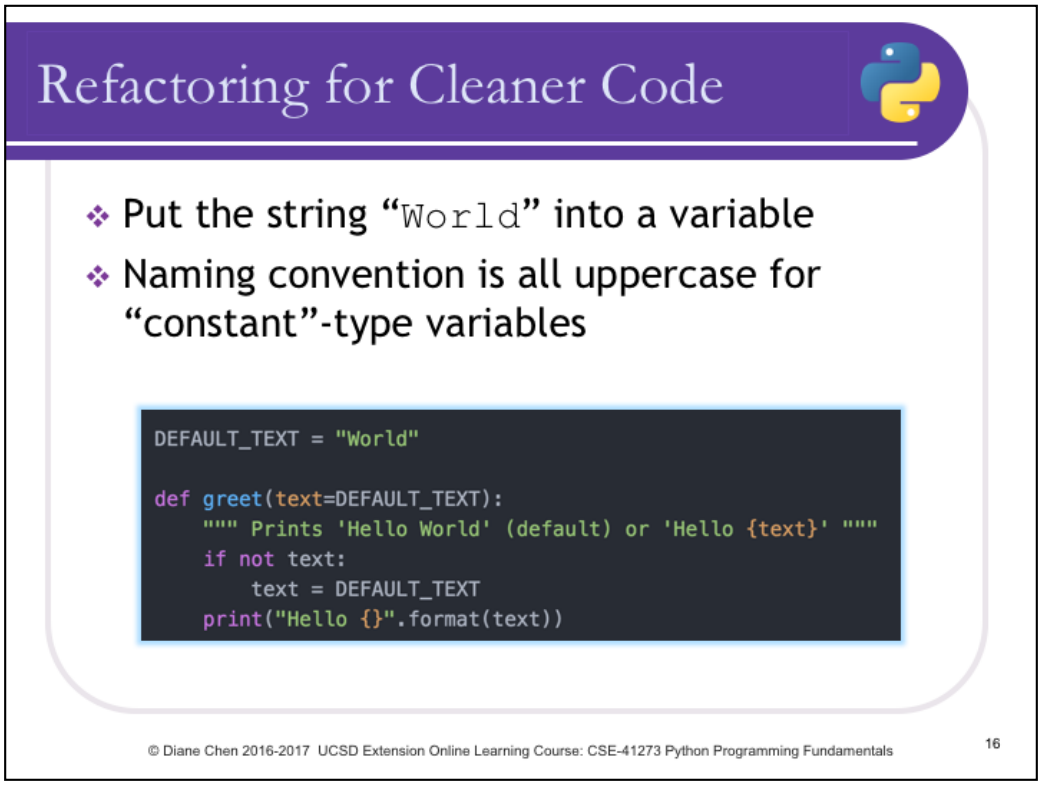
Running on the command line, our program works as expected when we put in "friends", even with an exclamation point. This is because Python treats command line arguments as strings and uses whitespace as the separator, so the exclamation point is just part of the argument string. But look what happened when we ran it with no arguments. Instead of printing Hello World, it printed Hello None. What's up with that?

The problem is because when there is no argument entered on the command line, the "text" variable from the argument list from argparse is the special Python value "None". Now that we switched to using argparse, we are always passing an argument to greet(), so the default is never applied.

We could test in the code before we call greet(), and call with no arguments if arguments.text is None. However, let's think about making the greet() function more robust so it can handle this and still use the default string.

We add a line of code to the greet() function to check the variable "text" for truthiness, or rather, for falseyness. In this way, we will not only catch when the variable passed is None, but also if the variable passed – maybe from some other program importing our module – is the empty string. Try it now; it should work as we would expect.

But there is something I don't like here. We have the string "World" in two places now. This is not clean coding practice. We like to follow the DRY principle: Don't Repeat Yourself.



Let's refactor our code to make it cleaner and easier to change in the future. We create a new variable to contain the string "World". Python doesn't have "constant variables" like some other languages, but we have the convention where naming a variable all uppercase means that we never expect to change. Then we use that variable, "default text", in both places where we need to set the text to a default value.

Another advantage to this is that if we want to change the default to be something else, like "Pythonistas", we only need to change it in one place. If this was a large program with several of these kinds of strings scattered all over, we can modify them all by just making one change to the code.

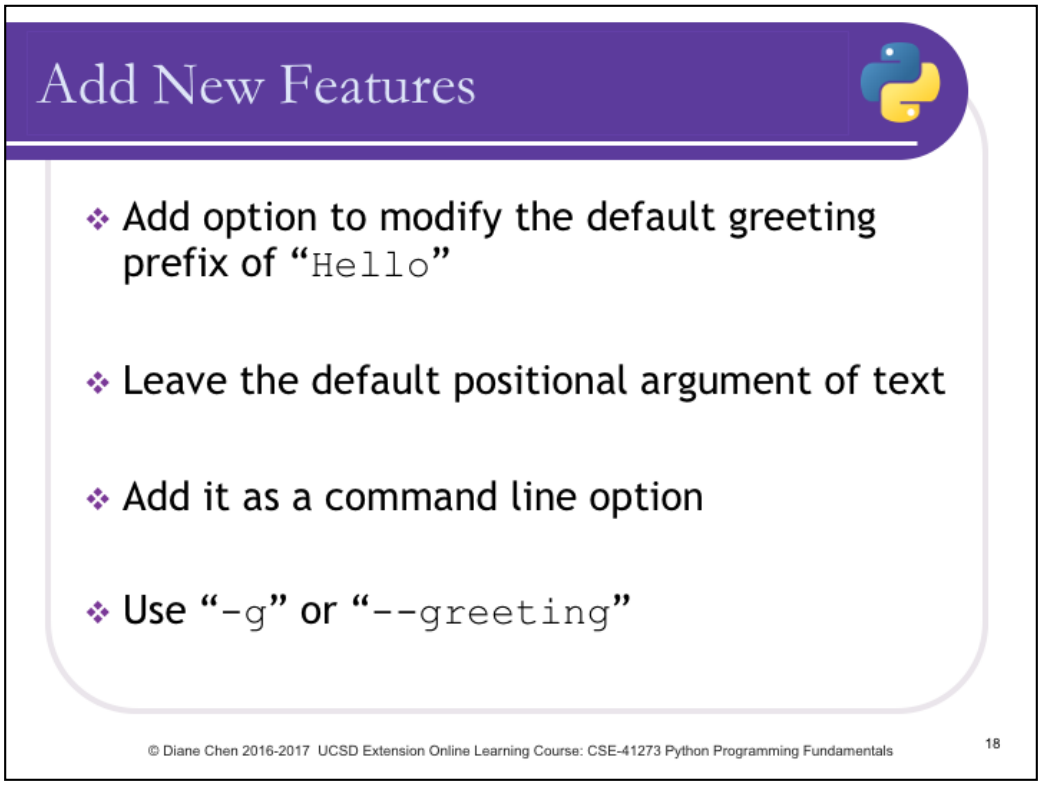
Good coding practice also says to use these "constant" variables for numbers that might be used. Say you have a program that stores some list of data, and you decide you're going to limit the list to 25 items. If you use a "constant" variable from the start, then if you decide later to allow 50 items, you can change it in one place. You don't have to wonder, if you find a number 25 in the code somewhere, is the number there because it is the size limit, or does it mean something else? Giving them meaningful names is very important too.

Back to the code! Be sure to test the different options again. Also open a new REPL and import it as before. Does it do what we expect? Try passing the greet() function an empty string. What happens?

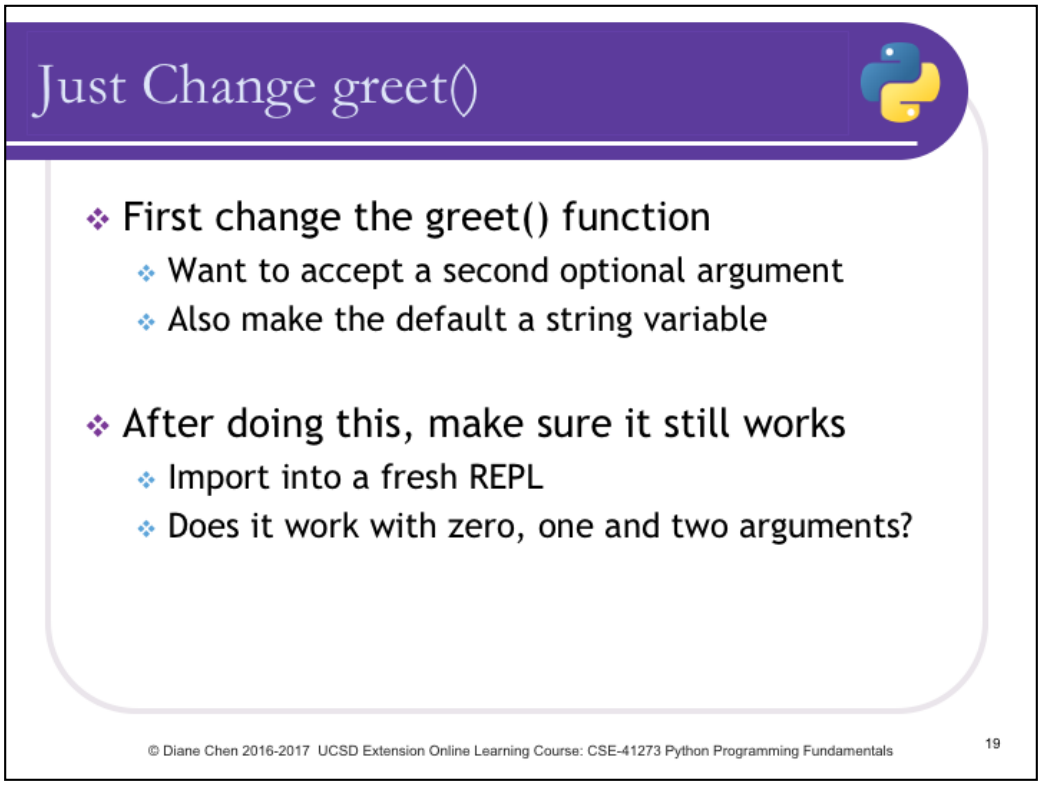
Our Finished Module

```
""" Greet module contains function greet()"""
import argparse


DEFAULT_TEXT = "World"

def greet(text=DEFAULT_TEXT):
    """ Prints 'Hello World' (default) or 'Hello {text}' """
    if not text:
        text = DEFAULT_TEXT
    print("Hello {}".format(text))


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('text', nargs='?', help="text to output after Hello")
    arguments = parser.parse_args()
    greet(arguments.text)
```


So here is our complete greet module. Isn't it great? Doesn't it feel wonderful that it's finished?

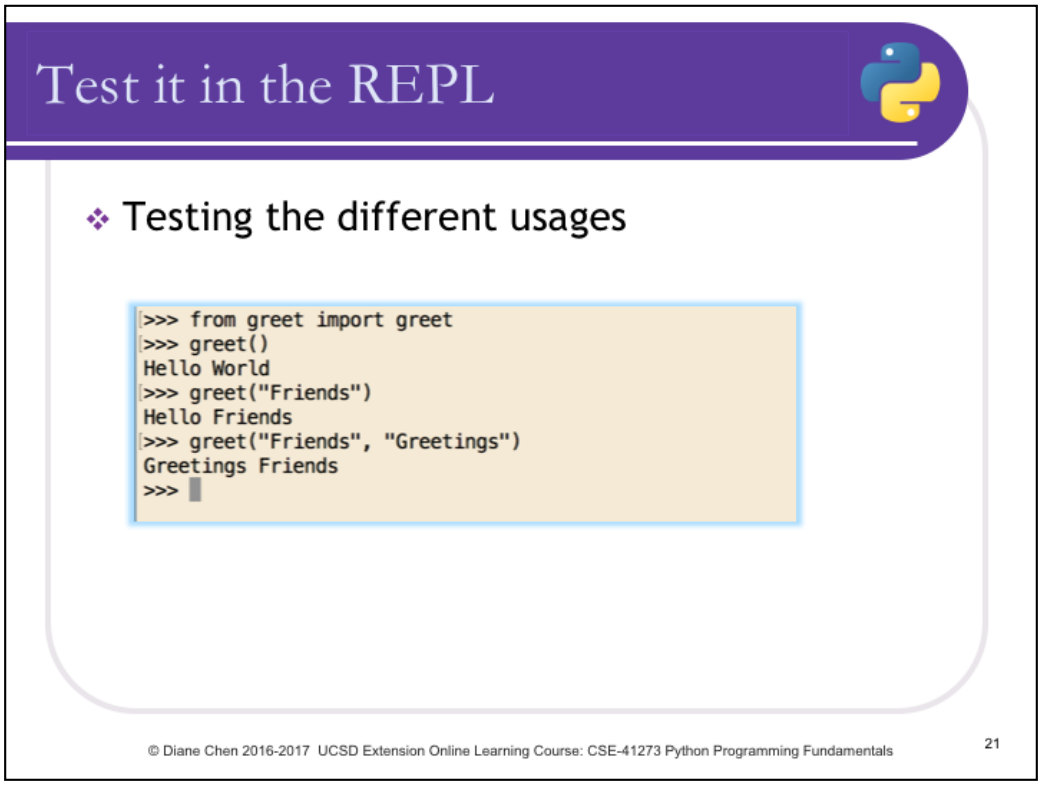


Now that we have a nice little program, let's add more features! In a project, this is sometimes called "scope creep"; when the requirements for being "done" change and new things are added to a project's definition. We want to add an option to change the default greeting prefix of "Hello". We don't want to affect the default positional argument, so we want it to be a command line option using "-g" or "—greeting". Let's think about what we need to do to implement this.



We want to limit our potential for errors, so the first thing we do is change only the greet() function to take an extra optional argument. If we put this argument second, then we won't affect any place where this module may have been imported and the greet() function used. If only one argument is passed to greet() then it will be assumed to be the first argument and things will work as before. Our second optional argument will default to "Hello" using a string variable. Try making these changes yourself. Then check it in a new REPL and try it out with zero, one or two arguments. After importing, it should work as expected with zero, one or two arguments. Go ahead, put the video on pause; I'll wait.

Here is the new version of the greet() function. We added a new "constant" string variable containing the default greeting of "Hello". A new parameter named greeting is added to the function and given a default of DEFAULT_GREETING. In the same way that we checked the "text" parameter, we check the greeting parameter and if it is falsey, we set it to the default greeting. Then the print statement is modified to print from the variables. I also change the docstring for the greet() function.



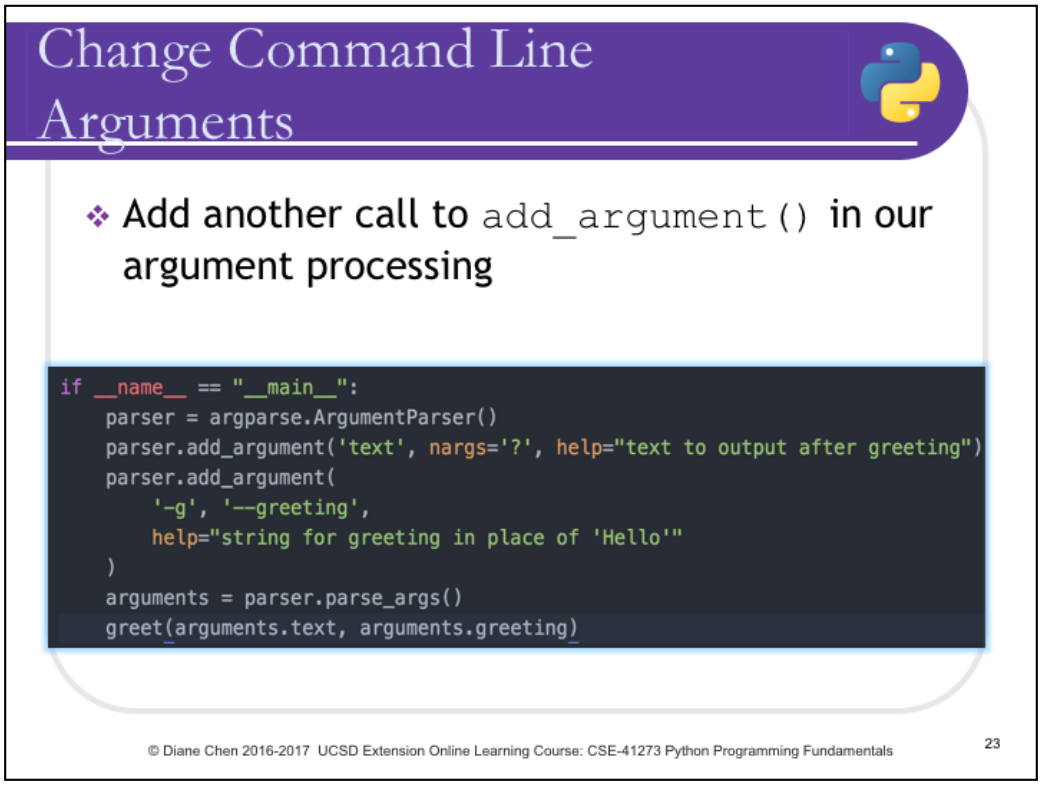
Let's test it in the REPL. Here we see that it works as expected for zero, one or two input parameters.

Using Keyword Arguments

- What if we only want to change the greeting?

```
>>> greet(greeting="Greetings")
Greetings World
>>>
>>> greet(greeting="Greetings", "Friends")
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>>
>>> greet(greeting="Greetings", text="Friends")
Greetings Friends
>>>
```


If we only want to change the greeting and not the text, we can use the named keyword argument as in the first example here. If we use the keyword argument and follow it with an unnamed argument, we will get an error. This is because keyword arguments can be in any order, so once it finds a keyword argument, everything after that also has to be keyword arguments. When we don't use the keyword name when we call a function (as in the examples in the previous slide), they are considered positional arguments. Don't forget to test with None and empty string arguments too.



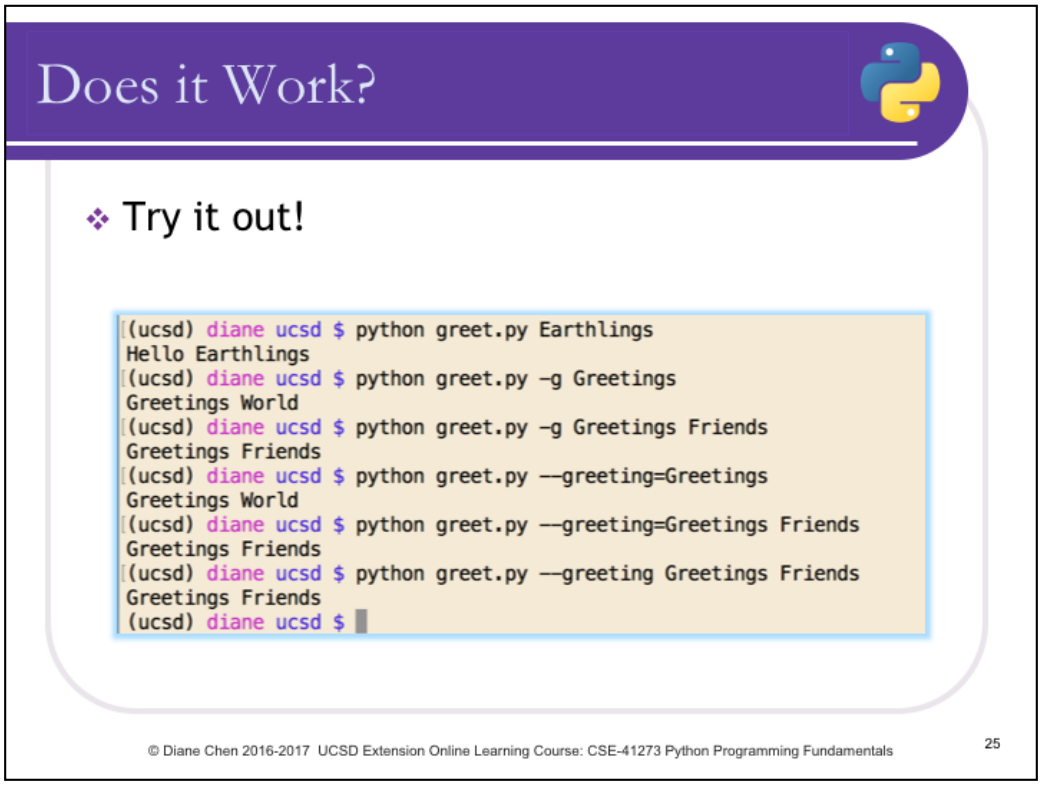
Now we need to change our argument parsing to handle the new option we want to support when the program is called from the command line. Here I've shown the changes to the argument parsing. We have another call to add_argument; putting in the abbreviated option of –g and also the extended option of minus-minus-greeting, along with our help messge. Because this line ends up being extra long, I have split it up into multiple lines for clarity. The new argument from parge_args is called "greeting" from our "minus-minus-greeting". Because it is an option, we don't need to specify "nargs". We call the greet function with both "text" and "greeting".

Help Updated Automatically

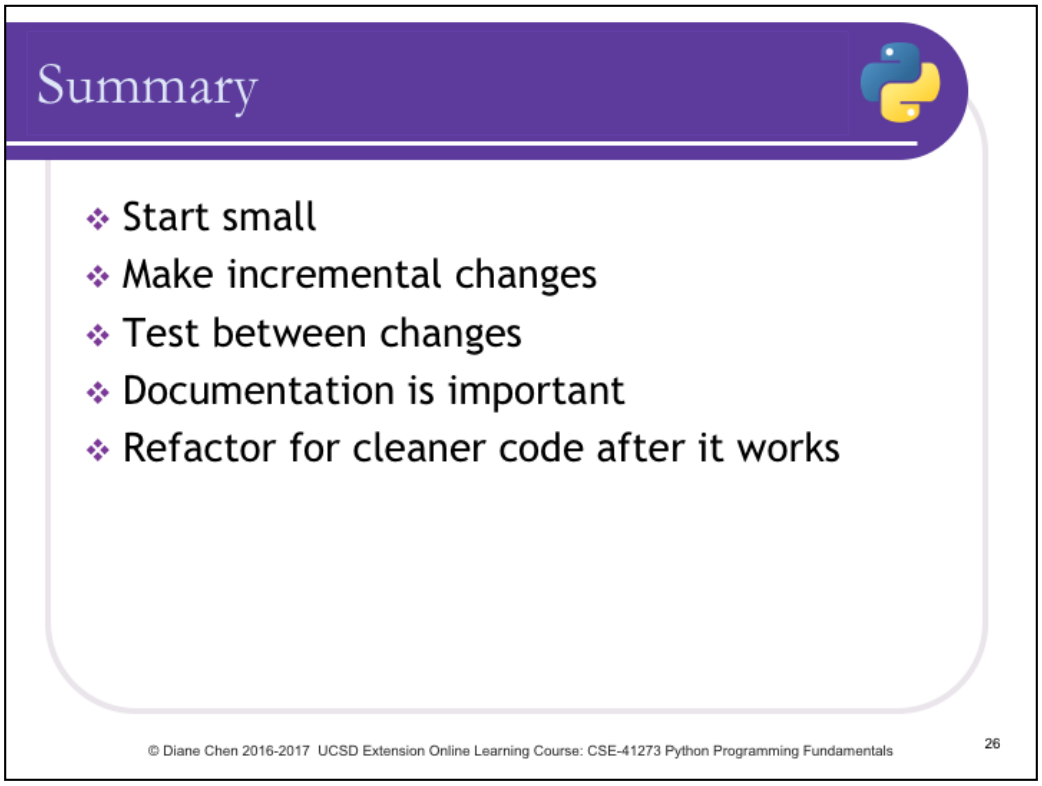
- Now our help documentation shows the new option

```
(ucsd) diane ucsd $ python greet.py -h
usage: greet.py [-h] [-g GREETING] [text]

positional arguments:
  text                  text to output after greeting

optional arguments:
  -h, --help            show this help message and exit
  -g GREETING, --greeting GREETING
                        string for greeting in place of 'Hello'
(ucsd) diane ucsd $
```


We only needed to make the one new call to add_argument and we get the new information in the help message automatically. Pretty nifty!



Let's try it out! Does it work? Yes! Running it on the command line we can see that it works for different variations of input. It's a nice feeling to have it finished, isn't it?



I hope you learned something from this example of creating a program with a command line interface. Even though it's a very simple program, you get an idea of the coding process and how the command line works. When you are writing a program, if you follow these general guidelines, things should go much smoother. When you start, you might think "Oh I know how to do that!" and write a whole bunch of code before testing anything. Then if there's a problem, even a little typo, it's hard to find the cause. So try to start small, with bits that can be tested and verified. When you make changes, make small incremental changes to the code and test things at each step of the way. Documentation is important, so don't forget it. If you write a package you want others to use, they aren't going to use it if they can't figure out how to use it or what it does exactly, from the documentation. In this case, once we switched to using argparse, we got the help documentation practically for free; but don't forget docstrings and comments in the code. Also, if it's a package that others might use, you're going to want to have user documentation too, not just code documentation.

When the program is working, look it over with a critical eye to see if things can be made cleaner or clearer or more "Pythonic". As we go along, you'll see more examples of "better" ways of doing things that make a program more readable for other programmers. We will also learn how to make test programs so that we can easily make sure that we do not cause new errors when we make a change.