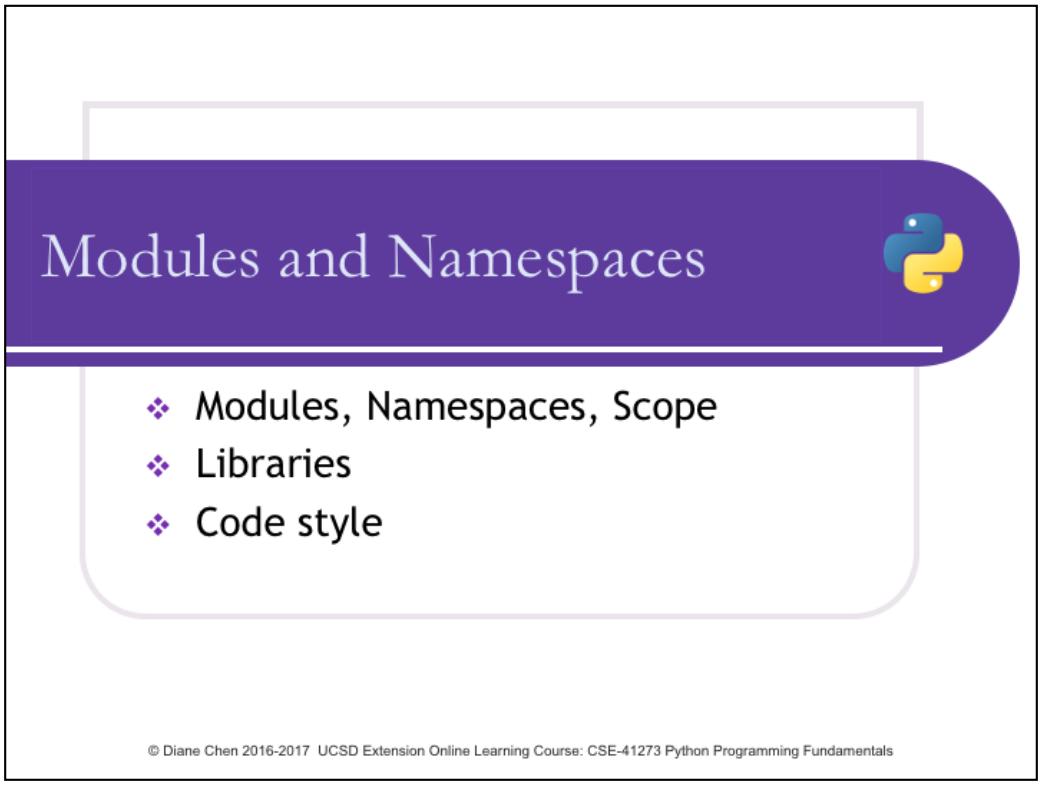
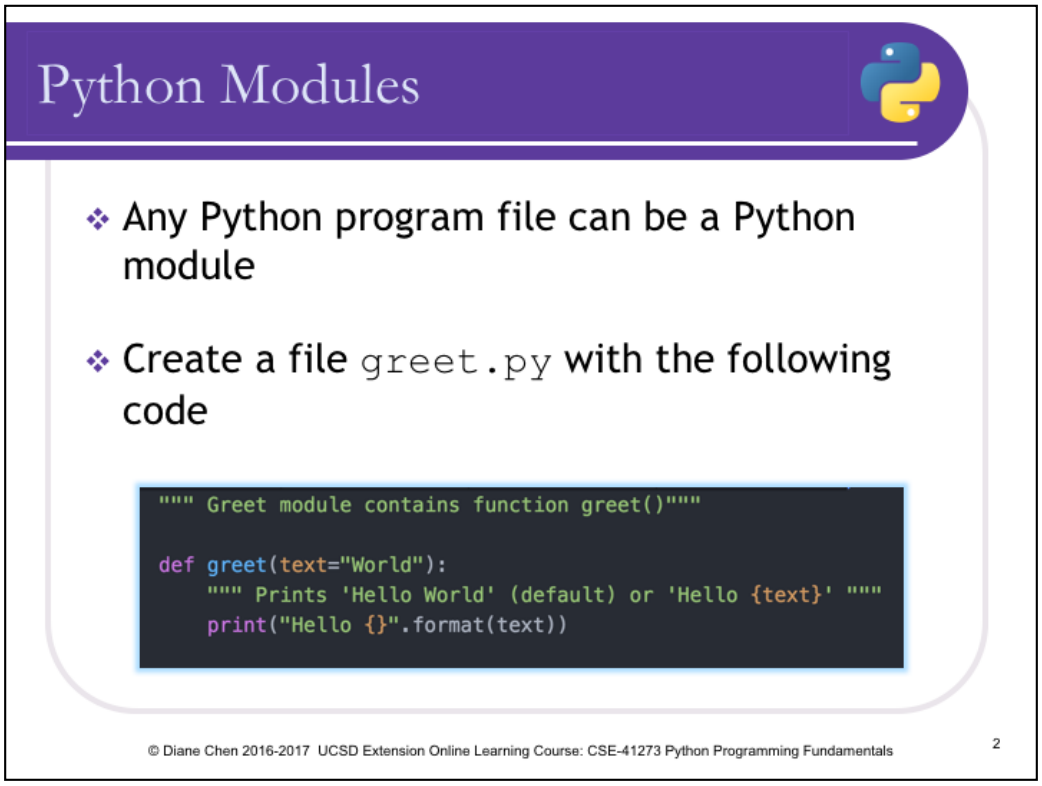
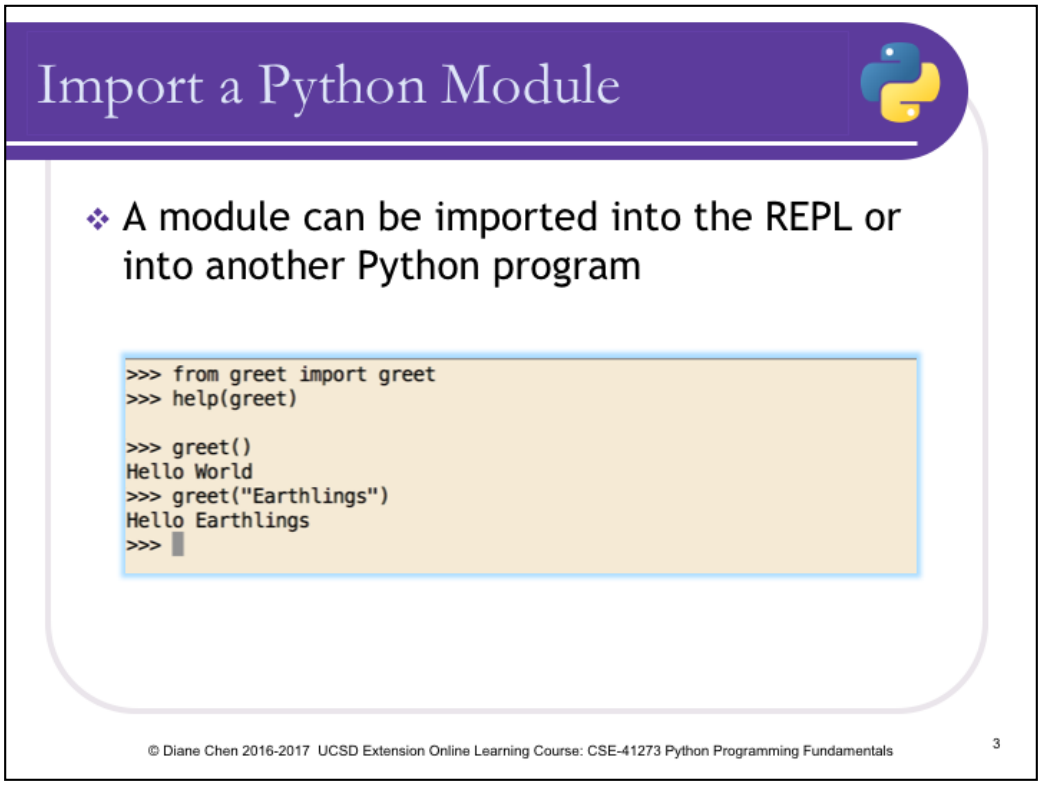


In this lesson I'll talk a little bit about modules, namespaces and variable scope. I'll also briefly introduce a couple of useful Python libraries we commonly need to add, and then talk about code style.



What is a Python Module? Any Python program file can be a Python module. Let's make a module greet.py with 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. 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.

Oh, and by the way, this color-coding is in the Atom editor using the built-in theme "One Dark". I like the way it highlights. Color-coding in the editor makes code so much easier to read and also it can be easier to spot typos since the color is often wrong.

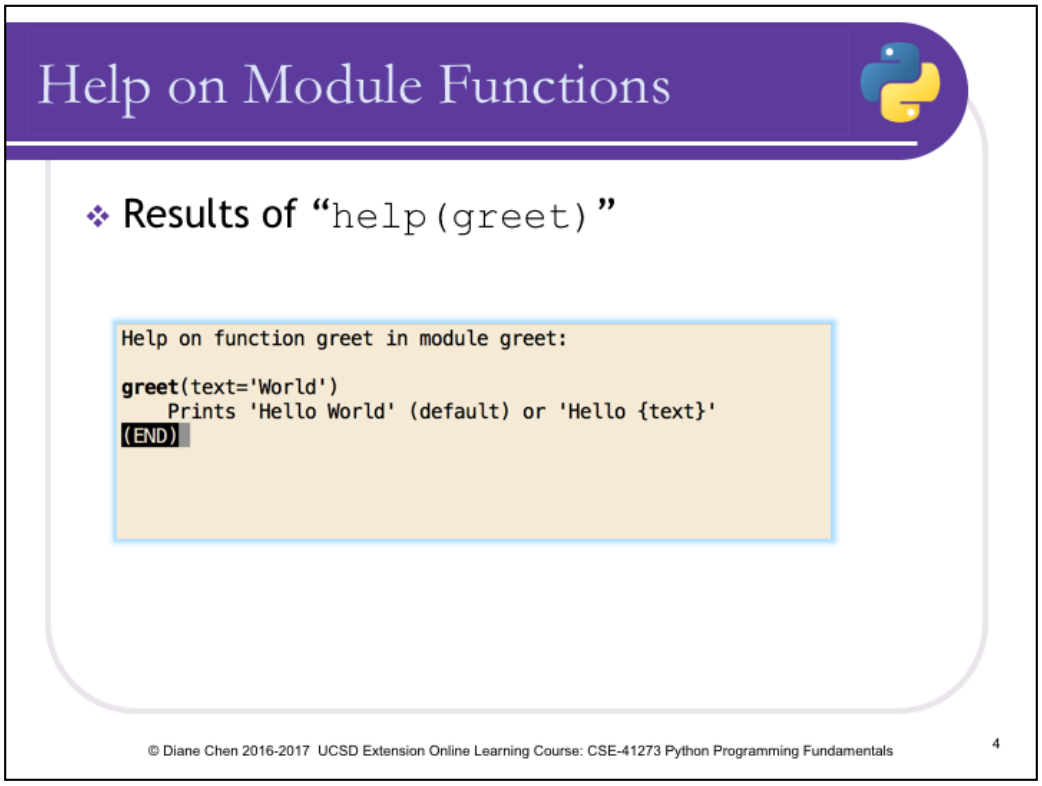


Python modules are libraries of code and you can import Python modules using the import statement. You can import into a program or from the REPL. Let's import our greet module into our REPL.

Enter "from greet import greet" into the REPL.

I'm sure you noticed that in our module, I put in docstrings; the string constants at the start of the file and at the beginning of the function greet(). Here I typed help(greet) and you see there is a little gap in the text where the help was displayed in an overlay. You can try it yourself; I show the results on the next slide.

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.

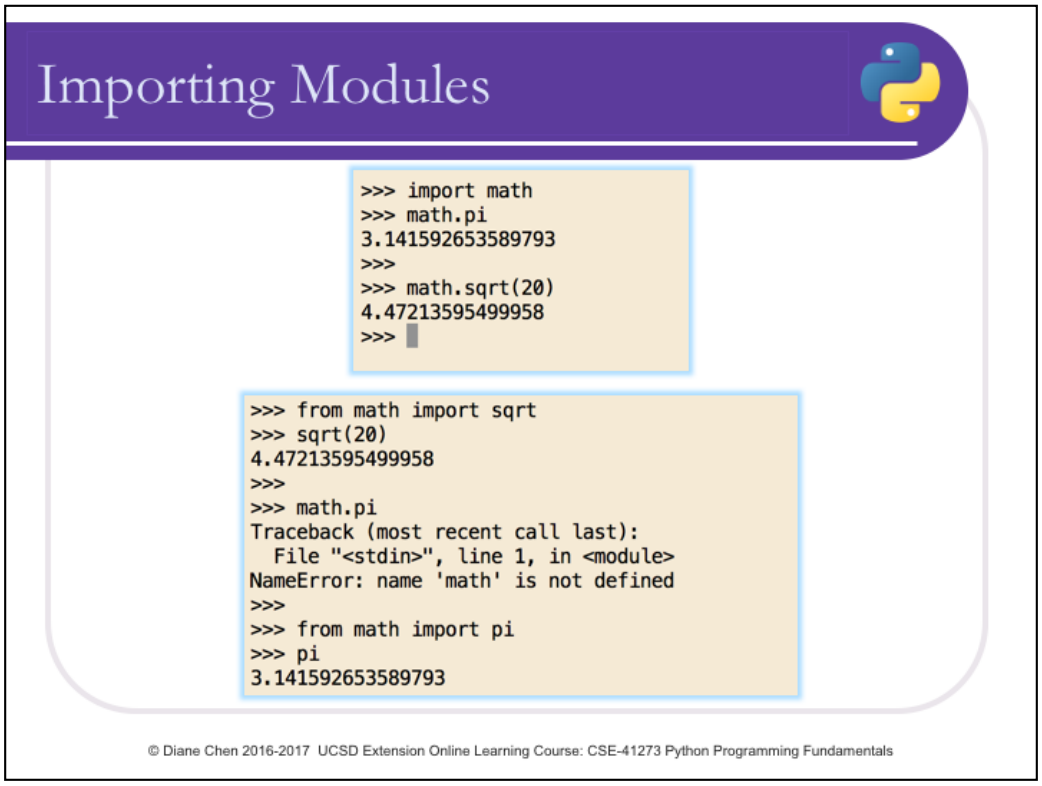


When we typed "help(greet)" in the REPL, it displays information in an overlay window as shown here. Note this information comes from the docstring from the function greet. You should be able to do this on any function or module.

When there is more than one screenful of help text, you can use the space bar or enter key to scroll through the help. The space bar scrolls one screenful, and the enter key scrolls just one line at a time.

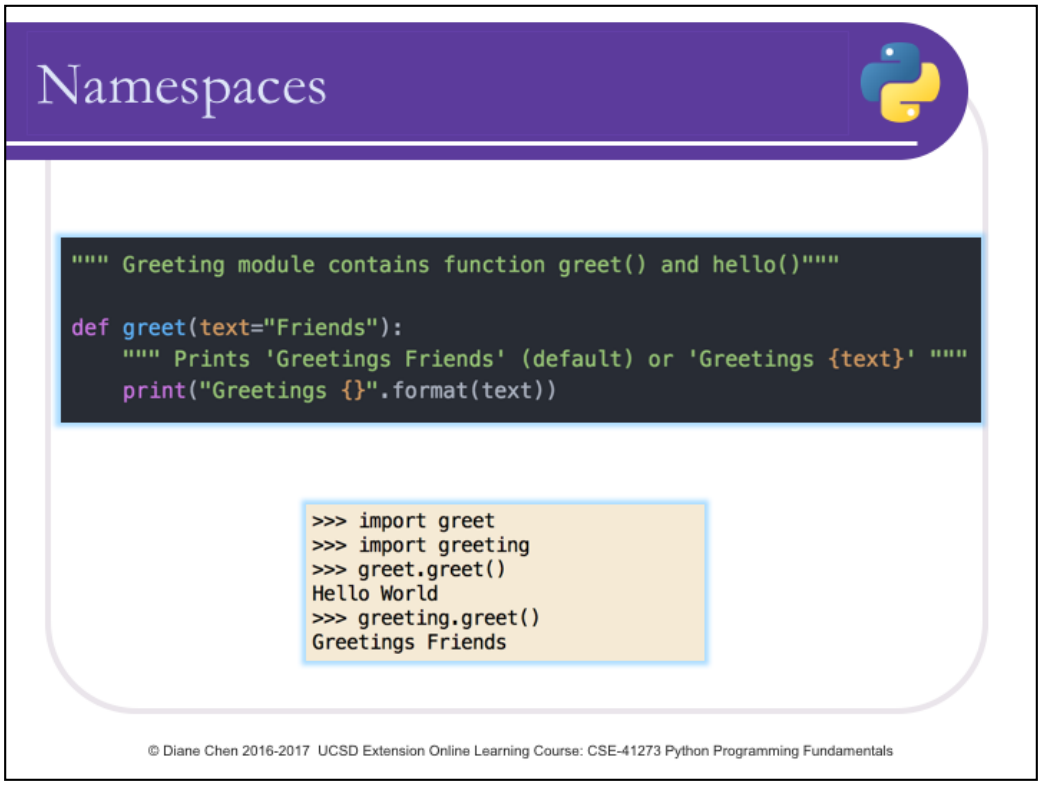
Type the letter "q" at any time to quit the help screen and go back to the REPL. So all we had to do to add help info to our module was to put a docstring on the function and Python automatically saves the docstring to use for help.

We'll be using this file in a future lesson, so don't delete it.



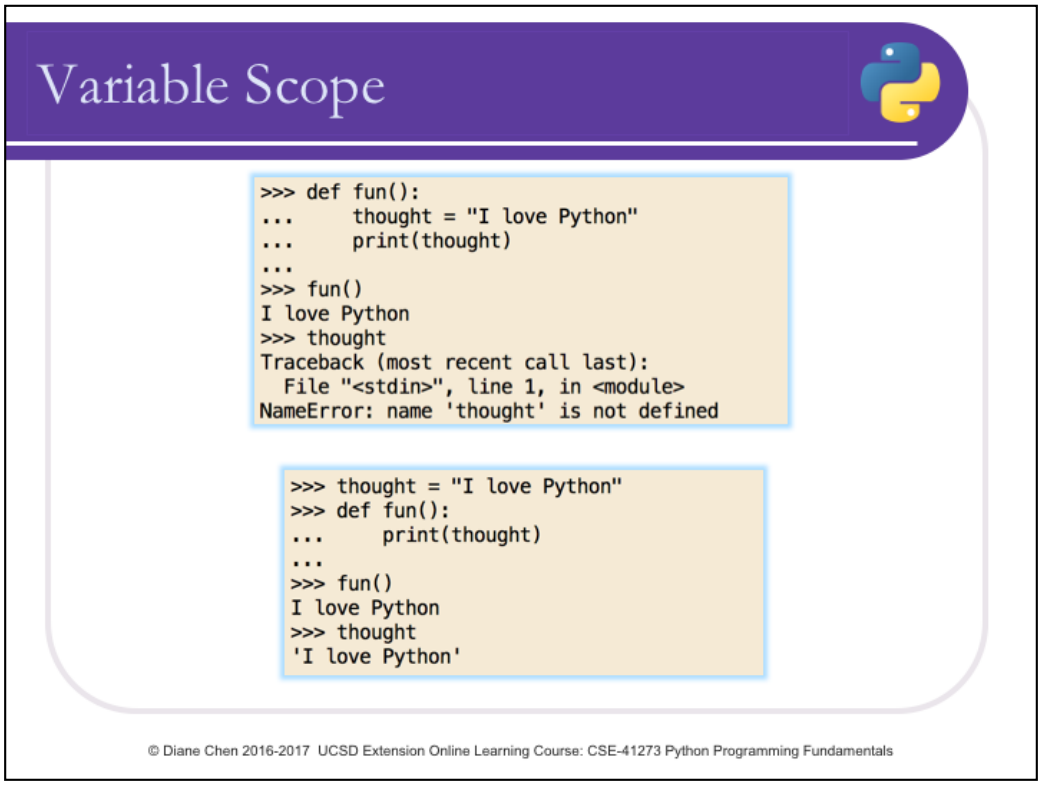
Let's start with a simple example of importing Python modules. We're going to import the math module by typing "import math". The module comes with a number of useful functions and variables. We're just going to demonstrate a couple of them. For example, if I wanted to use the value of pi, I can type math.pi and Python shows me the value of pi. It is always stored at the highest precision possible for the computer and Python version. The math module also comes with a number of useful functions. Let's try out the square root function. To use that, I type math.sqrt, give it the number and it returns the square root of the number. So in this case, I've asked Python to return the value of square root of 20.

This next example is in a fresh REPL. Now let’s imagine we are going to do some math-heavy calculations in our program and we will be using square roots often. We don’t want to have to type math.sqrt every time we want to do a square root calculation. How annoying! There’s another way to do it, by specifying just the object we want to import. Note I did say object, because functions are first-class objects in Python. We can type “from math import sqrt” and Python imports only the square root function, and I can use it without prefixing it with “math”. But notice that it doesn’t know what math.pi is. That’s because Python only remembered the sqrt function, based on our previous import statement. But if we need pi, we can import it the same way we imported sqrt.



Before we go to the next module, let's talk a little bit about namespaces and variable scope. What is a namespace? You can think of a namespace as a directory of names shared by objects that typically go together, and are in a module together. And its intention is to prevent naming conflicts, and allow us (or Python) to find the object (usually, but not always, a function) that we are referring to. This promotes code re-use by allowing programmers to find code that will help them get their job done, rather than writing everything from scratch. In the previous slide we saw that when we typed “import math”, we could use math.pi and math.sqrt in Python. Think of it as if the “import math” statement is importing a link to a table of contents of everything that is the math module. It doesn’t actually import the objects from the math module into the code, but Python knows that if it finds something prefixed by “math”, it can look it up and find what it needs. In this case, “math” is the namespace.

Here we see another example. We already have our program greet.py. Suppose we had another file greeting.py that also has a function greet() that prints out a different message. If we import both modules into the REPL, we have both of their namespaces available. There are 2 different greet() functions we can use. If we didn’t have namespaces to distinguish them, how would we (or Python) know which greet function we want to use? When we use imports like “from greet import greet” we have to be careful that we aren’t obscuring an existing function greet, or even a variable named greet. Similarly, we need to be careful of naming variables. For example, it’s not a good idea to name a variable “list”, because once you do that, you can no longer use the list constructor function.

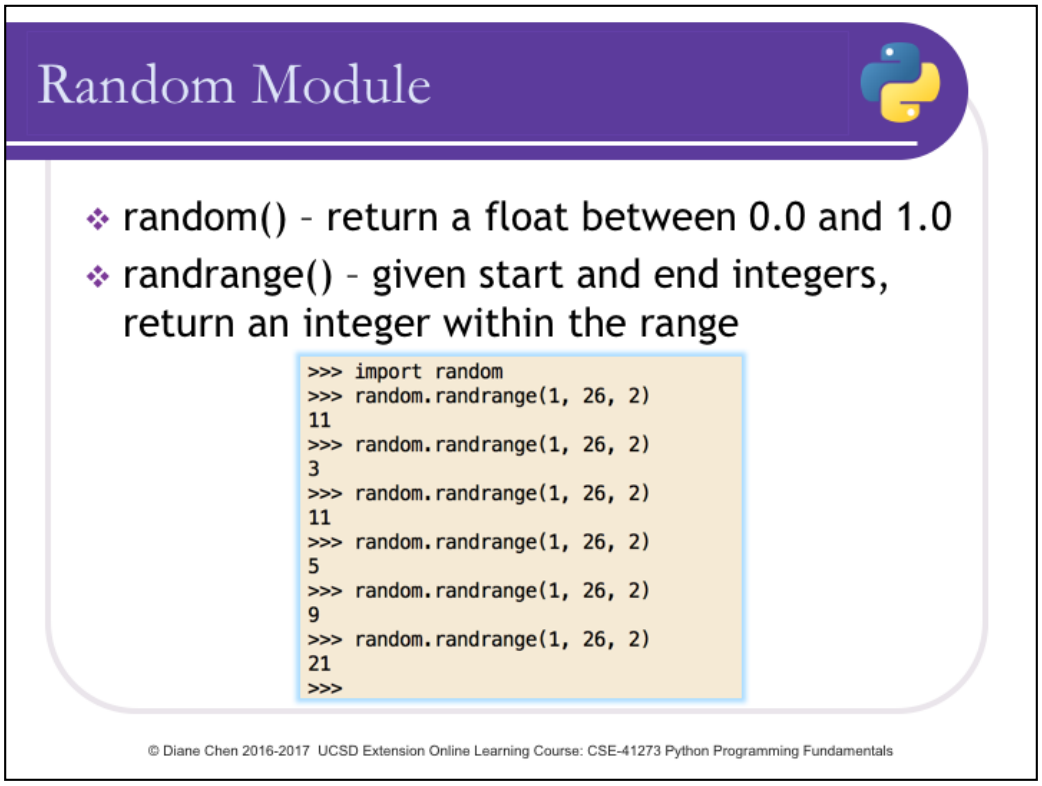


All programs have a global namespace, that contains any objects that have been defined globally. There is also the Builltin namespace with all of the standard Python objects like functions such as print() or len().

Local variables are those variables that are defined inside of functions – they are only accessible from inside the function they were defined in. Global variables are those that are defined outside of functions and are accessible from anywhere inside their enclosing scope. Here we see in the top example, the function fun(). Inside fun() there is a variable named thought, set to the string “I love Python”. But even after we execute the function and it prints out the string, if we try to access the variable outside of the function, we get an error, because it is now out of scope.

In our second example, we set the variable “thought” to “I love Python” before defining the function, making it a global variable. Inside the function we can print the variable thought with no problem. When Python executes the print statement, it looks in its local scope for the variable. Not finding it, it looks in the enclosing scope and finds it there. And after executing the function, thought is still the same, as we would expect.

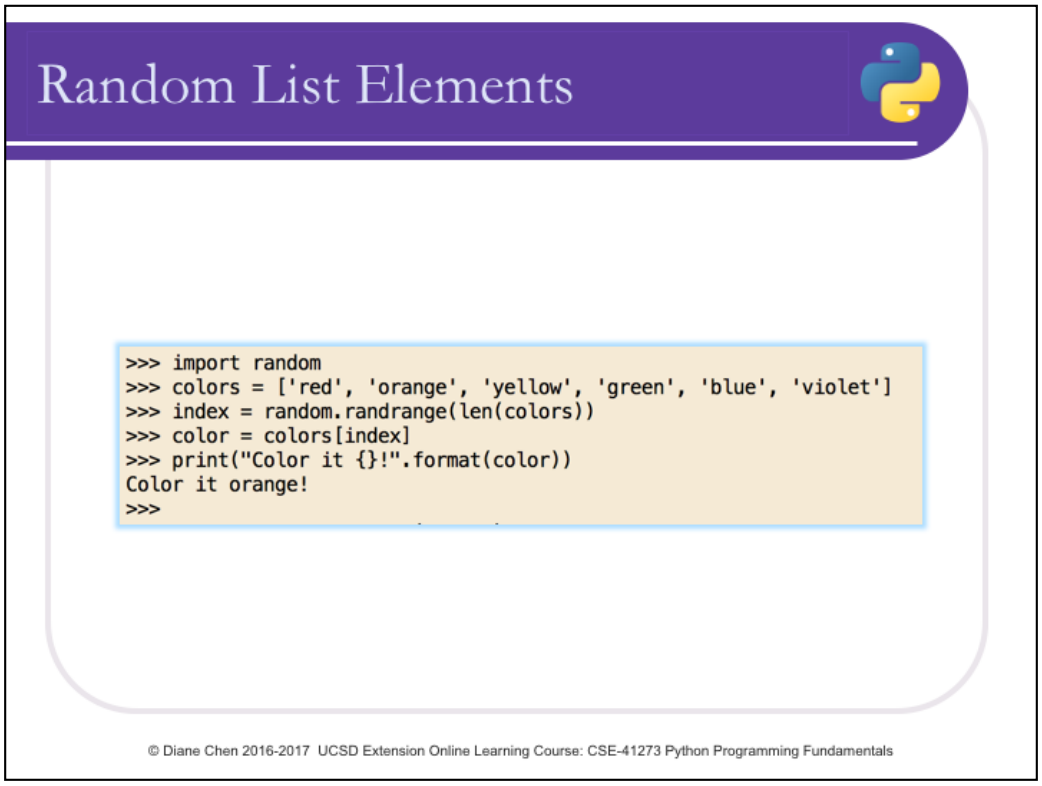
Whenever you assign a variable inside a function, it creates a local variable, even if there is a global variable already with that name. So let's see an example of this. Here we set the global variable thought to be the same "I love Python" string. Inside the function, we set the variable thought to the string "Django is fun too". Look what happens after running the function. You might expect that the variable thought will now contain "Django is fun too", but it does not, because inside the function, Python created a new local variable named "thought" with the new value. Once the function has exited, that local variable goes away.



We saw the math module a bit, here's another useful module, random. Often we would like to have some type of randomness in our code. The random module contains a number of useful functions to use for randomness. Note that it is actually a pseudo-random package and not meant for security or cryptographic purposes. For that, there is another module named secrets that is specially designed for security purposes.

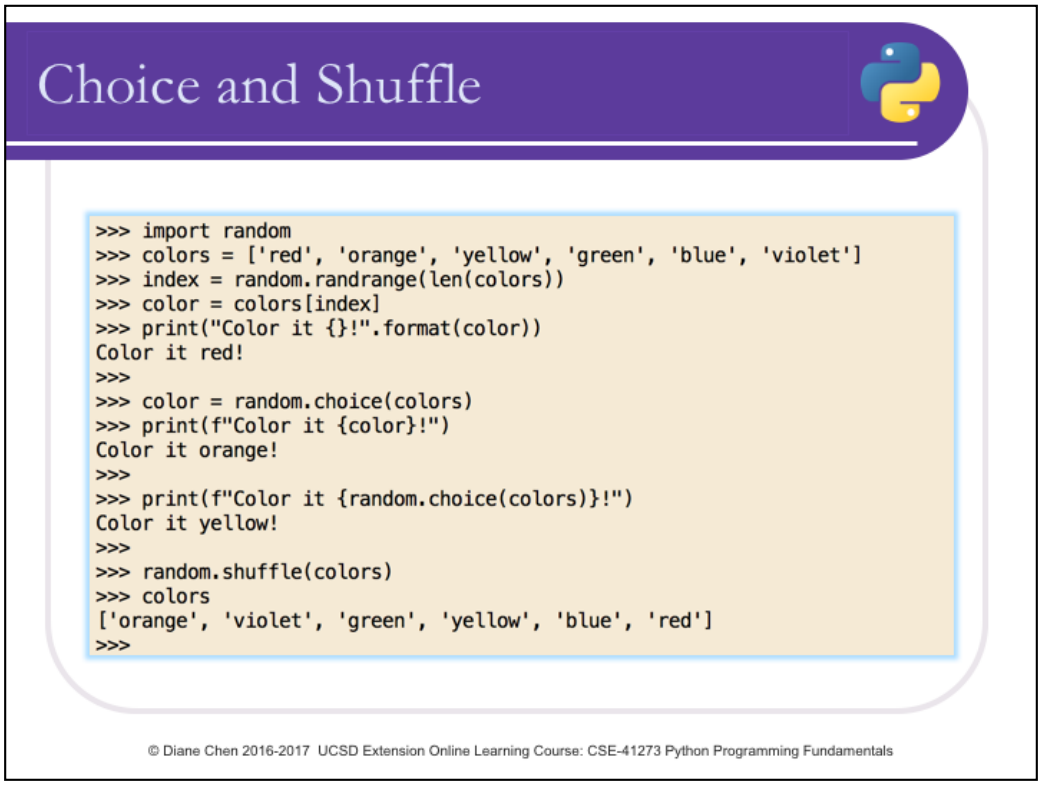
Here we see an example of using randrange from the random module. randrange takes as input a start and end value and an optional step value that defaults to 1.

This example here will give us a random odd number between 1 and 25.



Suppose we have a list of colors and we want a random color from the list. As we see here, we could get the length of the color list, and then use randrange to get a random integer in the range of the list, then use that to index into the list.

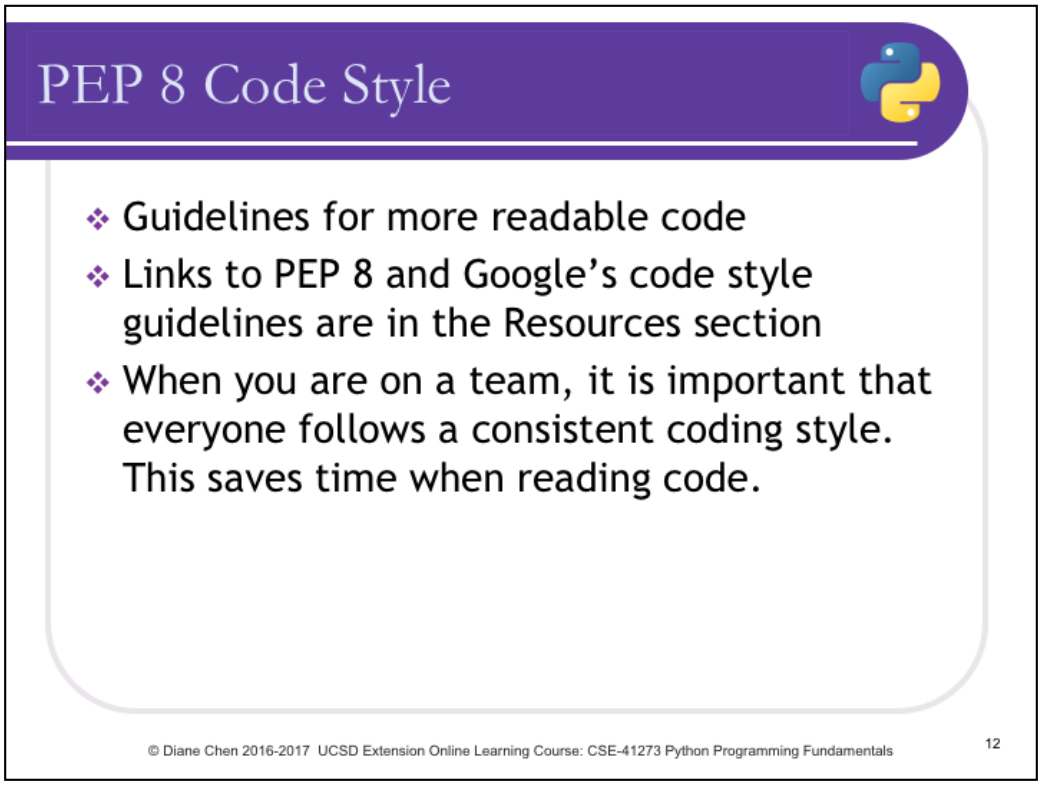
But that's kind of ugly code, cluttered with bookkeeping. Just like we saw with looping, we have a better way to do this.



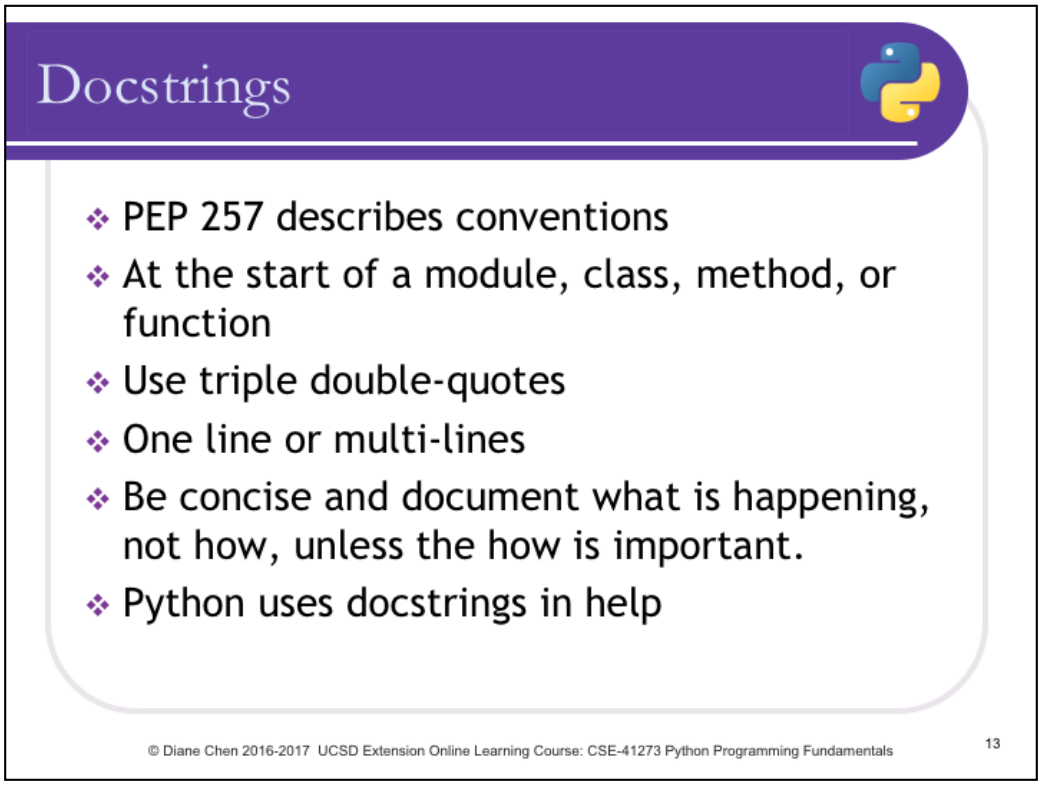
The Pythonic way to get a random element of a list is to use random.choice. Given the list of colors, random.choice() will choose one of the elements at random.

Notice the second time I print the color, I use the Python 3.6 f-string for much less typing. And the third time, I really take advantage of the new f-strings by inserting the call to random.choice inside the f-string.

Another useful function from random is shuffle – an in-place re-ordering of the elements. We only use it on lists, because other standard sequence types are immutable.



What is code style? It is the way you format your code – including many optional things like the amount of spacing you use and things like that. We follow some basic guidelines in order to make our code more readable. I have included some links in the Resources file to PEP 8, the original Python style guide, and a style guide from Google, along with some videos about code style. The "PEP" of PEP 8 stands for "Python Enhancement Proposal", but the system contains a lot of "meta" documents such as the PEP 8 style guide. One of the goals of Python was to make a programming language that humans can understand easily. It's especially important if you are on a team, for everyone on the team to follow the same code style guidelines. If you have to look at code that was written with a completely different code style, it just takes a little extra time to understand how the code works. Even if you're writing code all by yourself, it's a good idea to follow a consistent code style. If you eventually want to make your software open source, and want people to contribute to your project, they are not going to be happy if the style is very different from what most people are used to.



Docstrings are special strings that you put at the start of a module, function, class, or method to document what is happening in the code. PEP 257 describes conventions for docstrings. We always use triple-quoted multi-line string format, even if it is only one line, because it makes it more obvious and easier to make a single-line docstring into a multi-line one. Be concise in the docstring and describe *what* happens, not *how* it happens, unless the *how* is truly significant to the user of the software. Later we will see how Python uses docstrings in its help functionality.