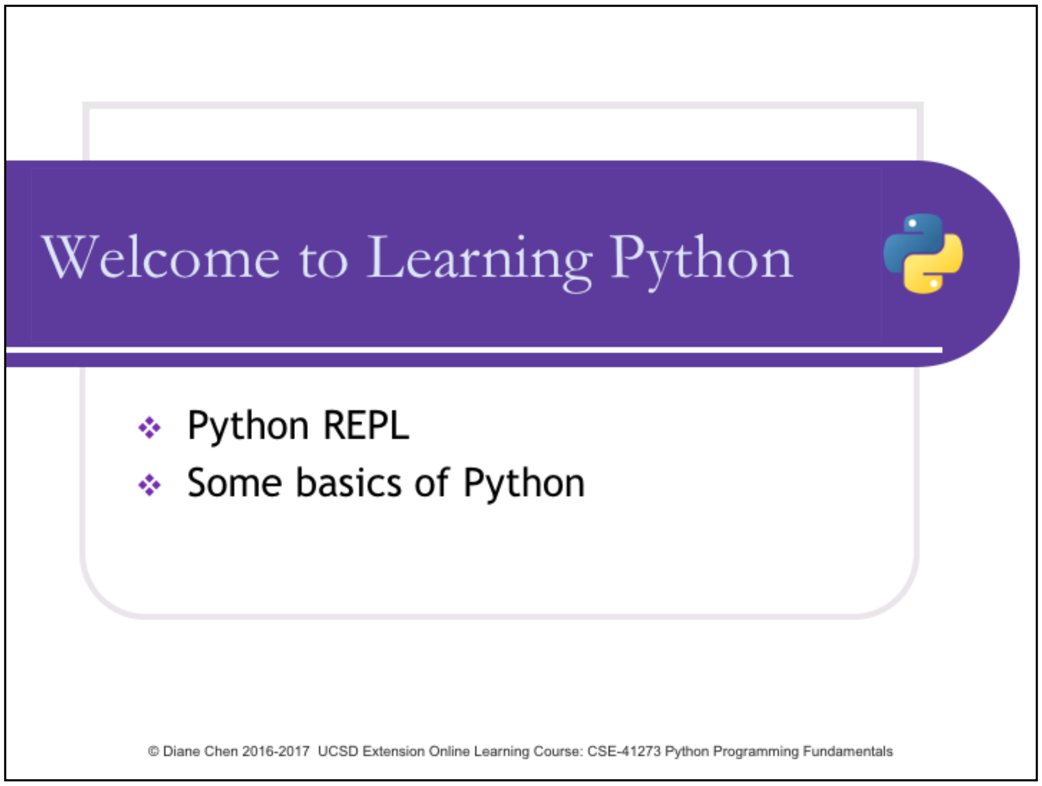


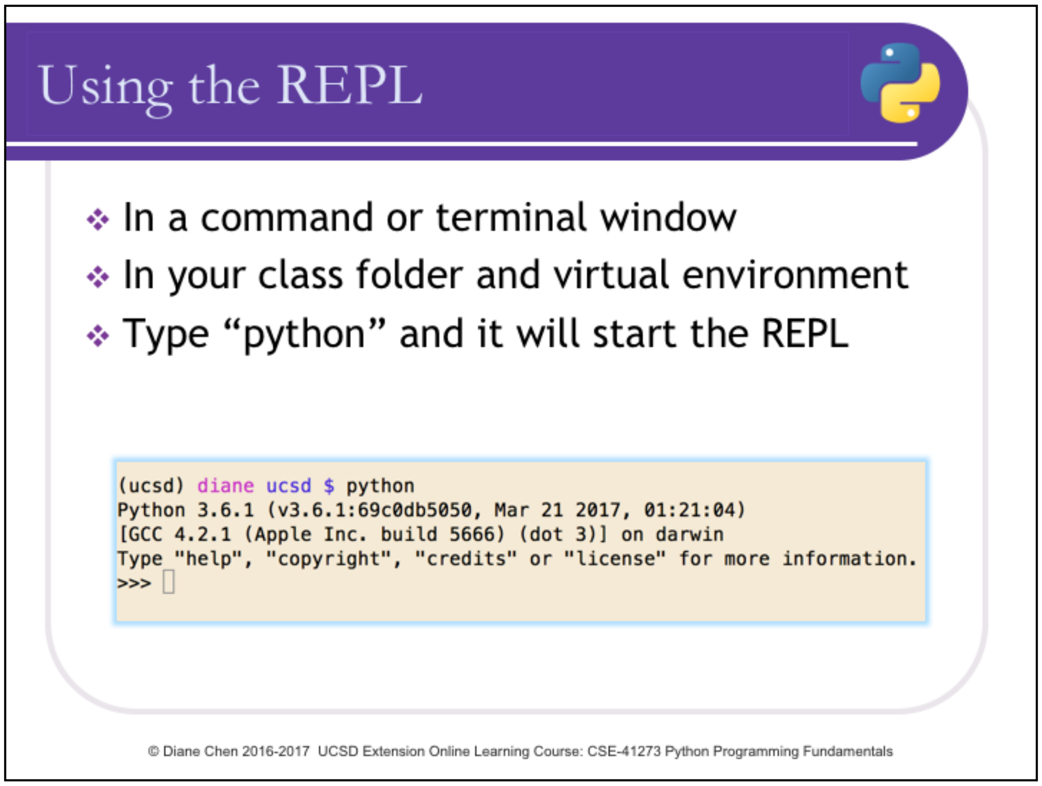
Welcome to Learning Python

- Python REPL
- Some basics of Python

Welcome to week 1 of Python Programming fundamentals. Please get your Python environment set up, if you haven't already done it. If you have any problems with it, be sure to let me know. The first steps of getting things set up can sometimes be frustrating; things should smooth out once that is set up.

I'm going to talk about some of the basics of Python. This will be review for many of you, but I am including it because I've found that a number of the students come to this class with programming experience, but no previous Python experience. If you have no programming experience at all, you will find this class may move too fast for you, although the textbook is aimed at new programmers. We will be going way beyond what is in the textbook in this class.

Python is an interpreted language, which means that there is no compile-link step between writing and running Python. That means that we can run Python interactively. Most of what you will see in these slides, at least for the early lessons, will be from the Python REPL. What is that? REPL stands for Read-Execute-Print-Loop, where we can interact with Python. The REPL is sometimes also referred to as the Python shell.



Using the REPL

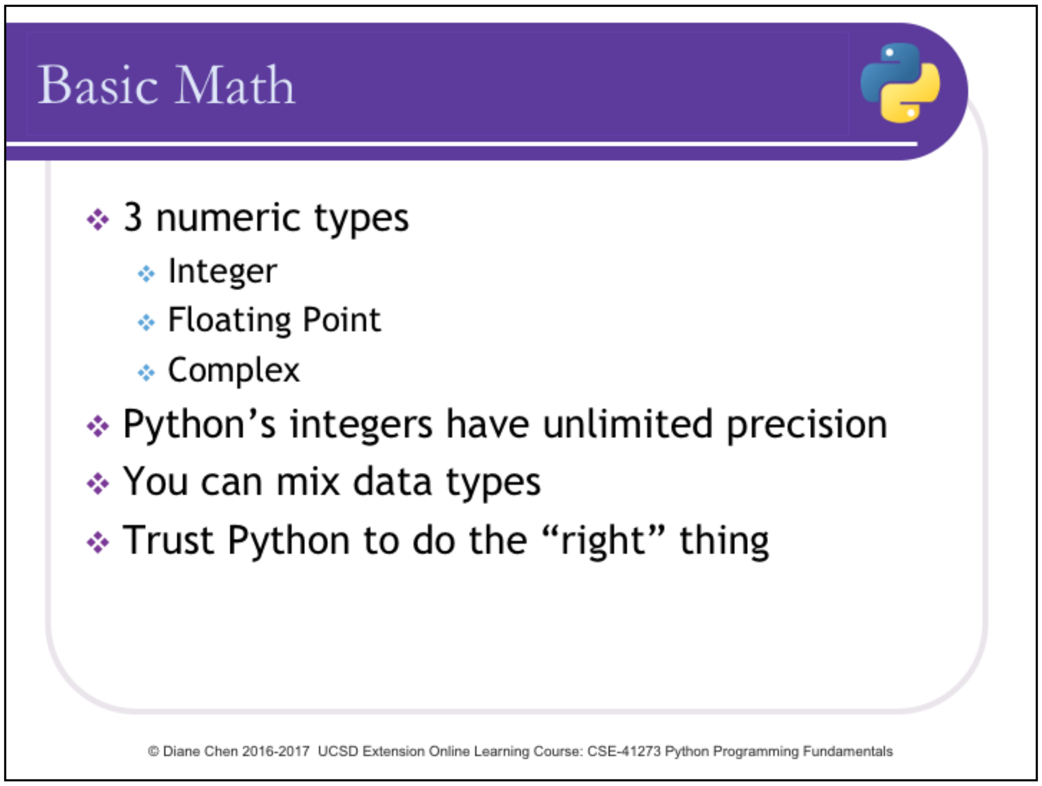
- In a command or terminal window
- In your class folder and virtual environment
- Type “python” and it will start the REPL

```
(ucsd) diane ucsd $ python
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

To use the Python REPL, you first need to be in a command or terminal window, in your class folder with your virtual environment activated. Whenever you do anything for the class, you should begin this way. Now type ”python” and it will start the Python shell or REPL. Python prints out the version information for you, and then is ready for your input.

Note the 3 greater than symbols. This is the prompt for the REPL. At this point you can start typing Python statements. Before I continue, I want to point out that once you have activated your virtual environment in your class folder, it will continue to be active in that window, so if you want to run Python from a different folder, it will work OK. You aren't stuck living in only one folder for your virtual environment.

Now, let’s try out the REPL.



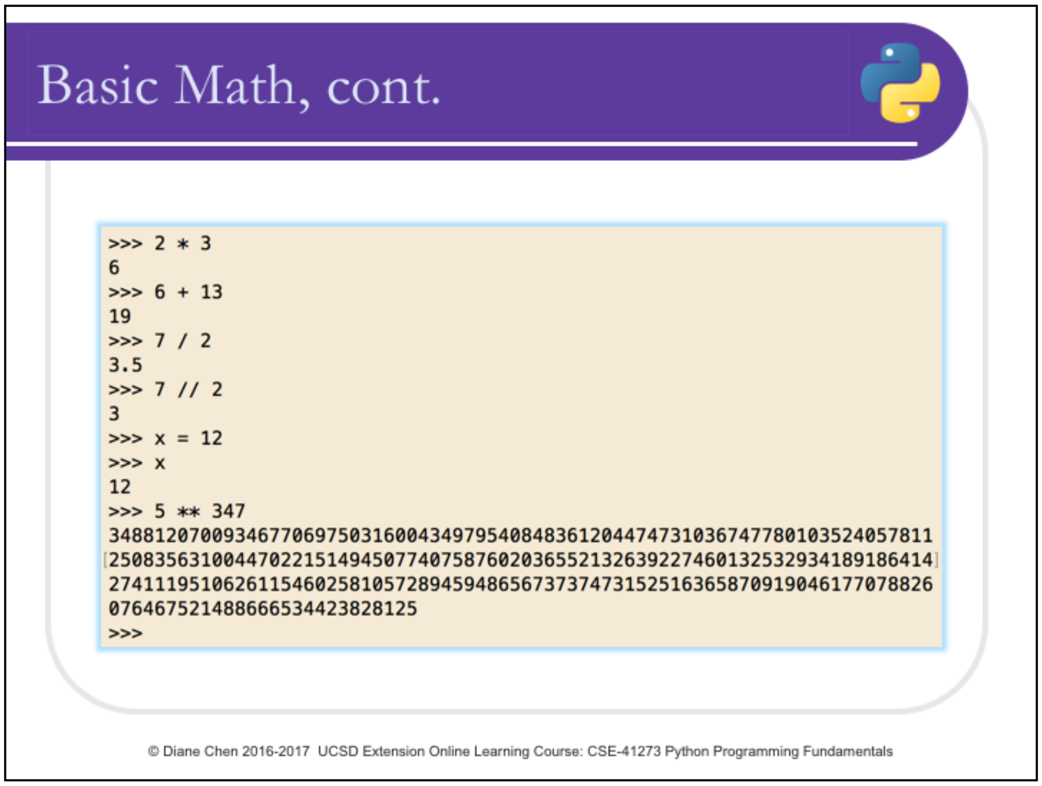
Basic Math

- 3 numeric types
  - Integer
  - Floating Point
  - Complex
- Python's integers have unlimited precision
- You can mix data types
- Trust Python to do the "right" thing

Python provides three basic numeric types, integers, floating point numbers, and complex numbers. We will only concern ourself with integers and floating point numbers in this class.

A unique characteristic of Python integers is that they have unlimited precision. That means your integer will never be too long to fit into Python's integer type, so there is no such thing as integer overflow. This is very different from most other programming languages!

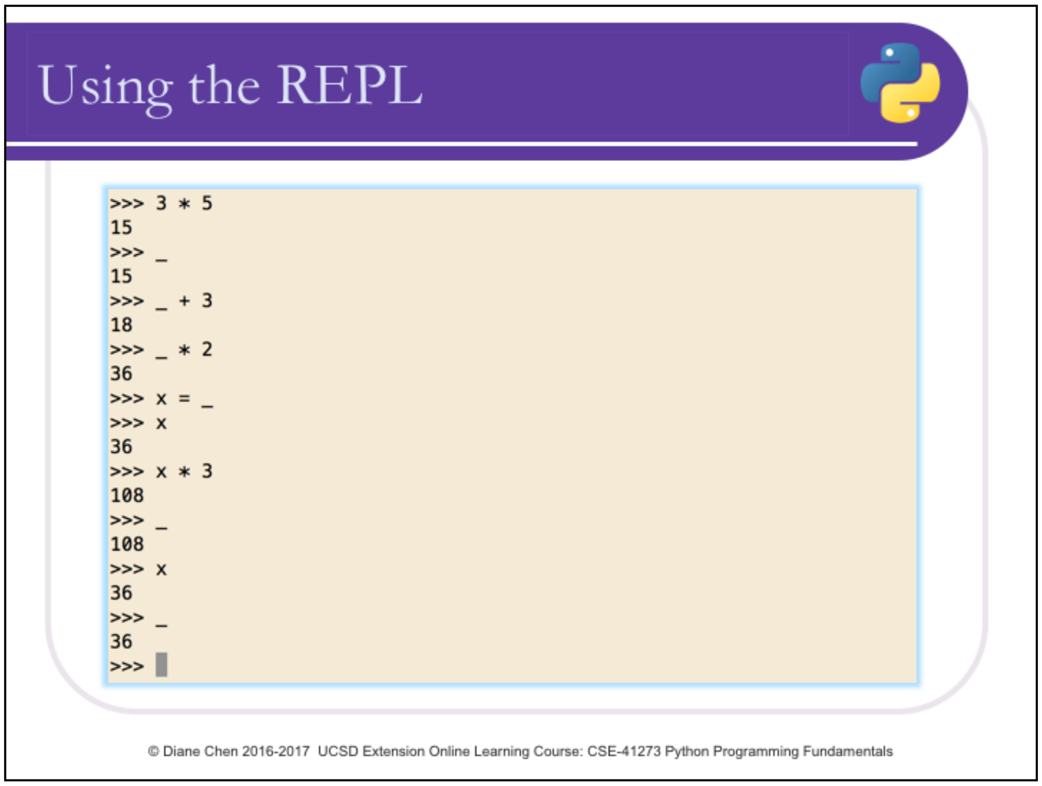
Another aspect to realize about numbers is that you can freely mix different numeric types. You can trust Python to do the right thing when there are mixed types in a statement.



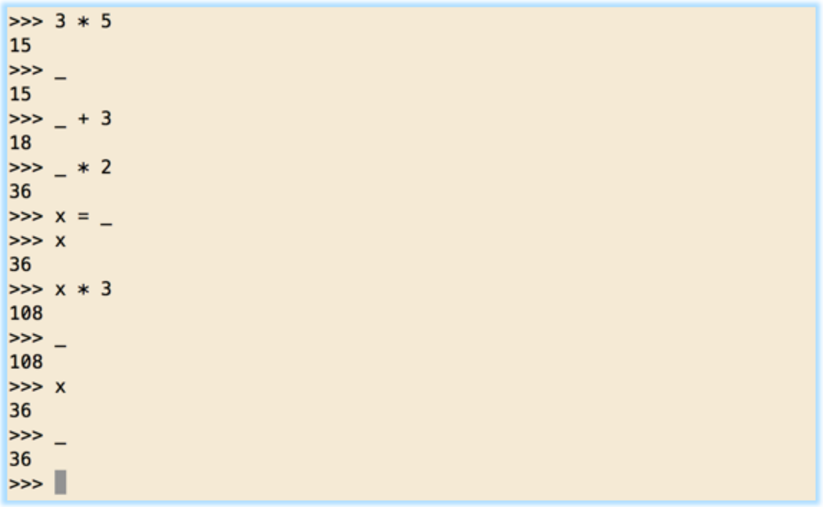
Basic Math, cont.

```
>>> 2 * 3
6
>>> 6 + 13
19
>>> 7 / 2
3.5
>>> 7 // 2
3
>>> x = 12
>>> x
12
>>> 5 ** 347
348812070093467706975031600434979540848361204474731036747780103524057811
250835631004470221514945077407587602036552132639227460132532934189186414
274111951062611546025810572894594865673737473152516365870919046177078826
076467521488666534423828125
>>>
```

In the REPL, we can do basic calculations as you would expect. We can multiply 2 times 3 and python prints the result, 6. Add 6 plus 13 to get 19. Divide 7 by 2 (using integers) and we get a floating point 3.5. You might expect that because we used integers, we should get an integer back. In python 2, that was the case; but in Python 3 they realized that it makes more sense to give the real answer when integers are divided. If you want integer division, also called Floor division, you can use the double-slash. Note that Python is printing the result of the calculations – this is the way the interactive shell, or REPL operates. Note if we say “x = 12”, Python doesn’t print anything. That’s because there wasn’t a result from the statement since the expression '12' is assigned to the variable x. If we want to see what is the value of the variable x, we just type “x” and Python gives us the value. Now we see an example of how Python can handle very large integers. I don’t know that I would ever need a number that big, but we can see that we don’t have to worry about “integer overflow” like we would in most other languages.

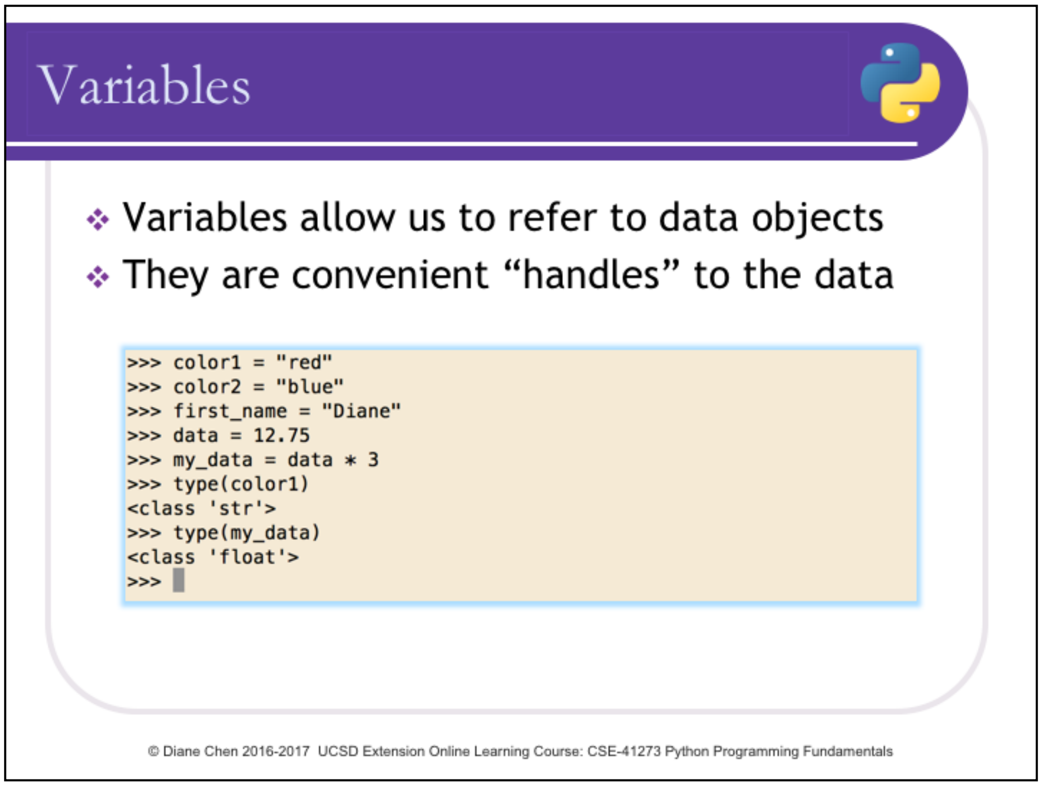


Using the REPL



```
>>> 3 * 5
15
>>> _
15
>>> _ + 3
18
>>> _ * 2
36
>>> x = _
>>> x
36
>>> x * 3
108
>>> _
108
>>> x
36
>>> _
36
>>> 
```

One cool thing about the REPL is that it uses the underscore character as a special variable. This underscore character contains the value of the last expression that was printed in the REPL. So if we do some calculation, we can chain calculations using the underscore. We can use it to assign a value to a variable. Note that when we do x times 3, while the underscore variable changes, the value of x doesn't change because we didn't assign any value to x. However, note that when we just print the value of x, it also changes the value of the underscore variable because something was printed as a result of the previous expression.



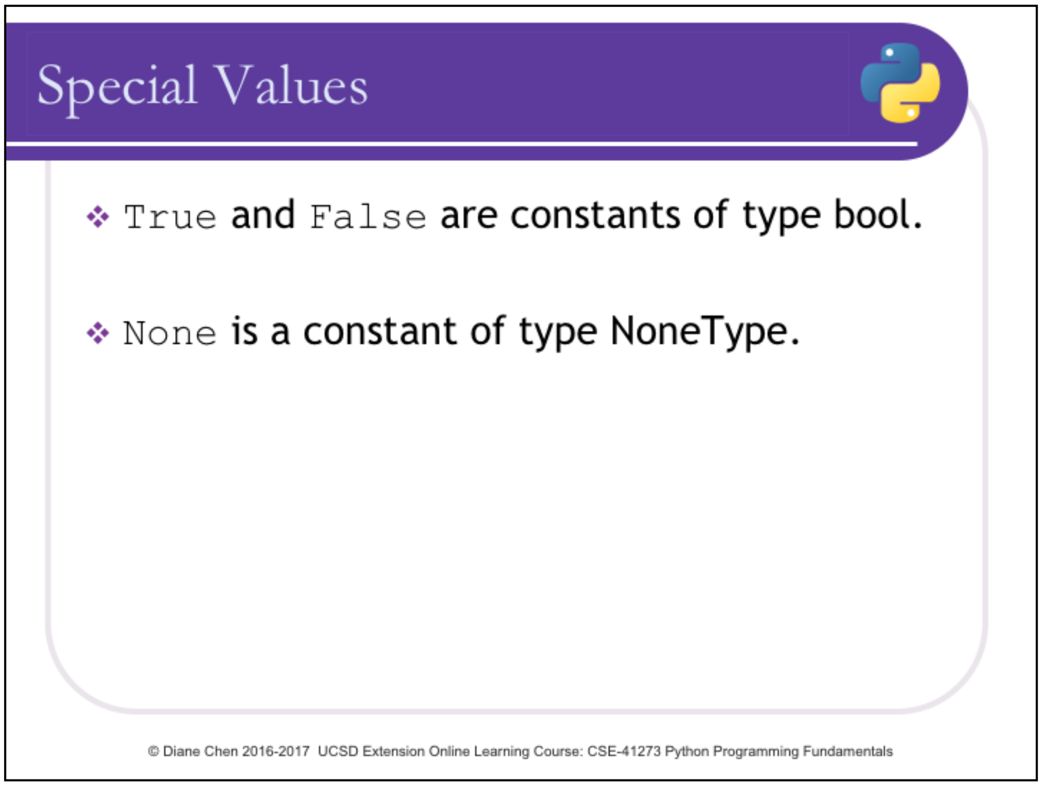
Variables

- Variables allow us to refer to data objects
- They are convenient “handles” to the data

```
>>> color1 = "red"
>>> color2 = "blue"
>>> first_name = "Diane"
>>> data = 12.75
>>> my_data = data * 3
>>> type(color1)
<class 'str'>
>>> type(my_data)
<class 'float'>
>>> 
```

Variables in Python are references to data objects. What does this mean? Everything in Python is an object of some kind, and variables allow us to have a convenient way of referring to the objects we are using.

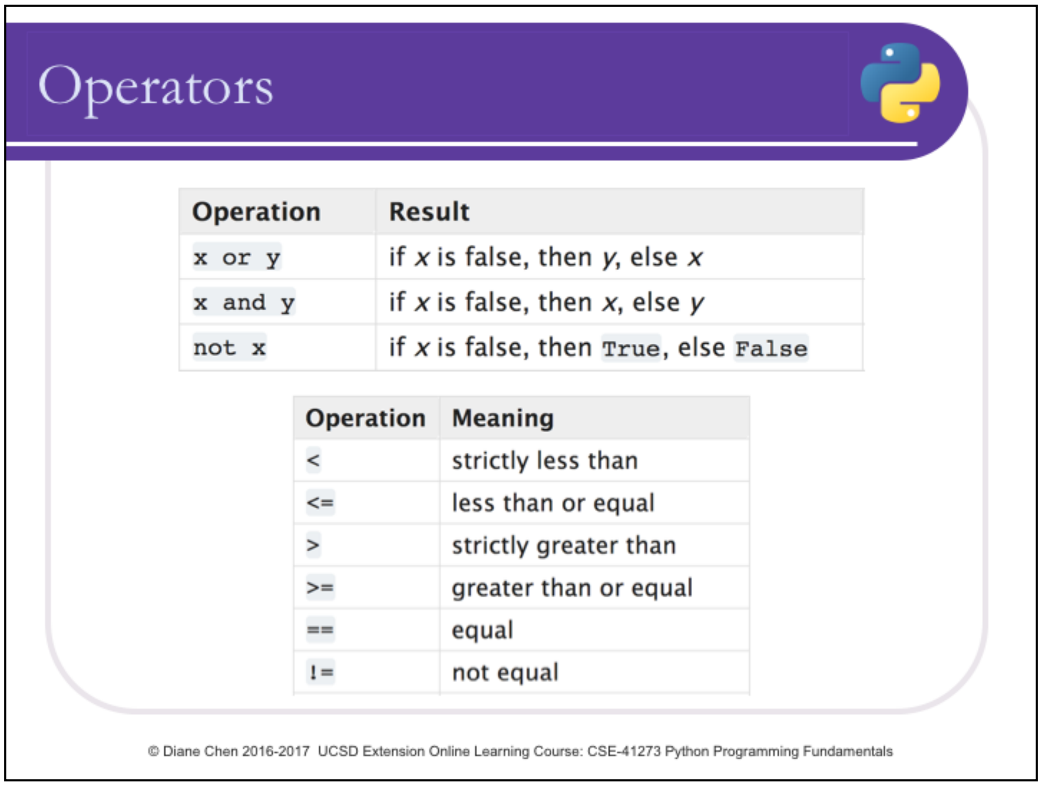
Note that everything has a type. We will talk more about data types later, but here you can see two types , “str” which is string, and ‘float’ which is a number. Here I am using the function “type” with input of a variable. Python looks at the object referred to by the variable and returns the type of that object.



Special Values

- `True` and `False` are constants of type bool.
- `None` is a constant of type NoneType.

Python has 3 special built-in constants: True, False, and None. True and False are constants of type bool or boolean. The constant None is very special. It represents emptiness or the absence of a value. We will see more of the None constant in the future.



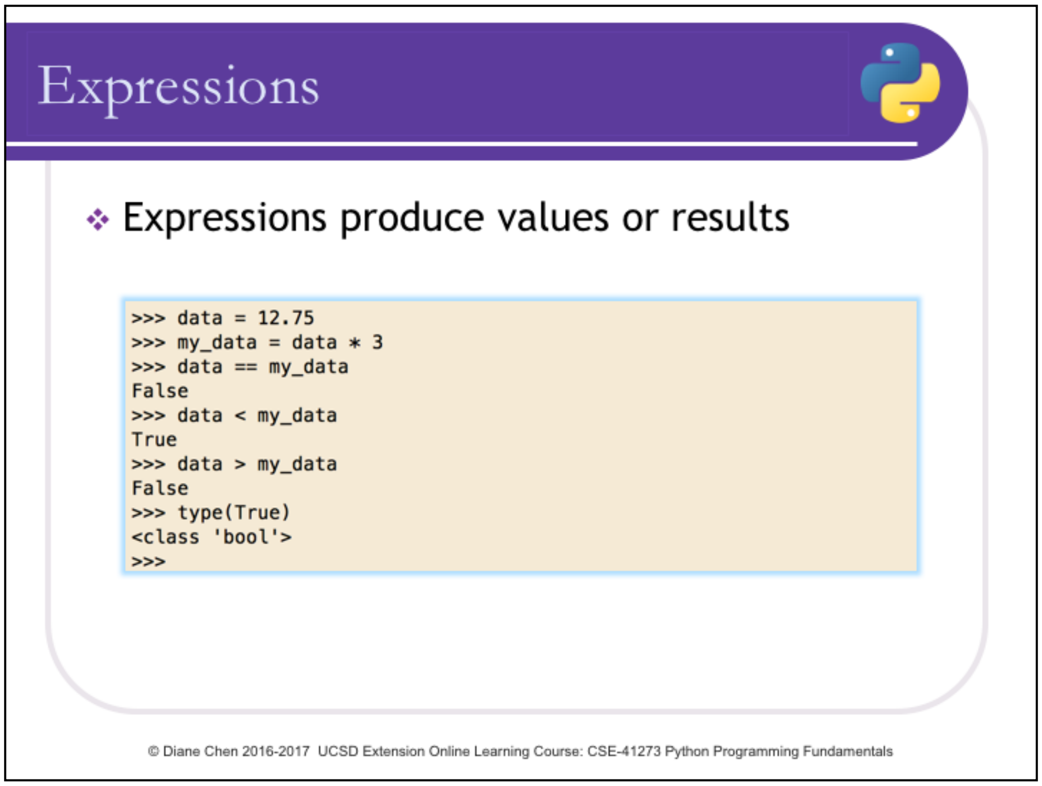
Operators

Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal

We’ve seen some of the standard operators already, like the arithmetic ones. Boolean operations are "and", "or", and "not". Here are some truth tables to describe how "and" and "or" work. If you have two boolean variables x and y, the result of "x and y" is True only when both of them are True. The result of "x or y" is true if either of them is True. Of course, "not" reverses True or False. We also often need to compare objects in Python. Usually they are numbers, but we can also compare other types of objects. We'll see later in the class how we can create our own objects and implement our own comparisons. Our comparison operators are "is equal", "not equal", "greater than", "less than", "greater than or equal", and "less than or equal". When we use them on numbers the result is obvious. We will see later on that is is not always obvious when we do comparisons on other data types.

For integers, Python also supports the usual bitwise operators of and, or, exclusive or, etc.



Expressions

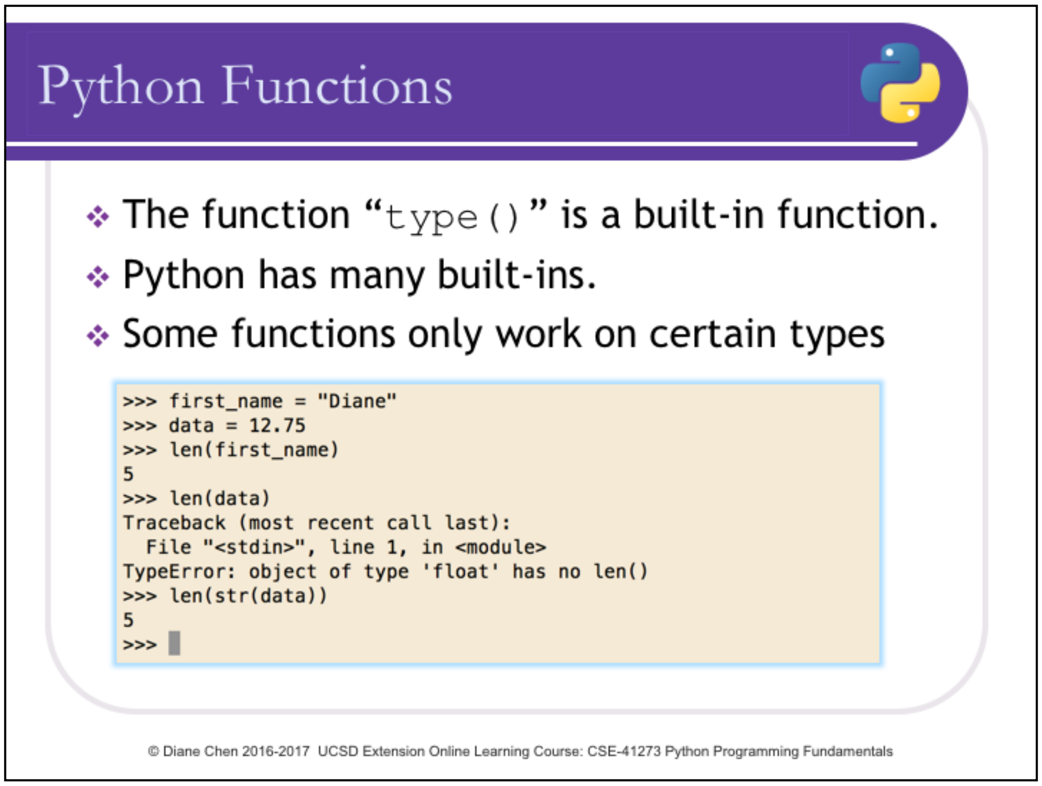
- Expressions produce values or results

```
>>> data = 12.75
>>> my_data = data * 3
>>> data == my_data
False
>>> data < my_data
True
>>> data > my_data
False
>>> type(True)
<class 'bool'>
>>>
```

Let's talk about expressions. What are they? An expression is a single unit of objects and operators that combine to produce a value. They are often arithmetical. Sometimes they are formed in order to produce a yes or no answer, which in Python is represented by the boolean values True and False. Note that these are always capitalized in Python. If you are using them and leave off the capitalization, Python will think it is a variable with that name. What is the type of True or False? They are typed "bool" which stands for boolean.

In our examples here, the first statement assigns the value of 12.75 to the variable data. In the next statement, the expression “data * 3” is evaluated and the result is assigned to the variable my_data. The next 3 lines are expressions representing a comparison between the value of data and the value of my_data.

See chapter 2 of the textbook for more information on expressions, operators, and how they all work together.

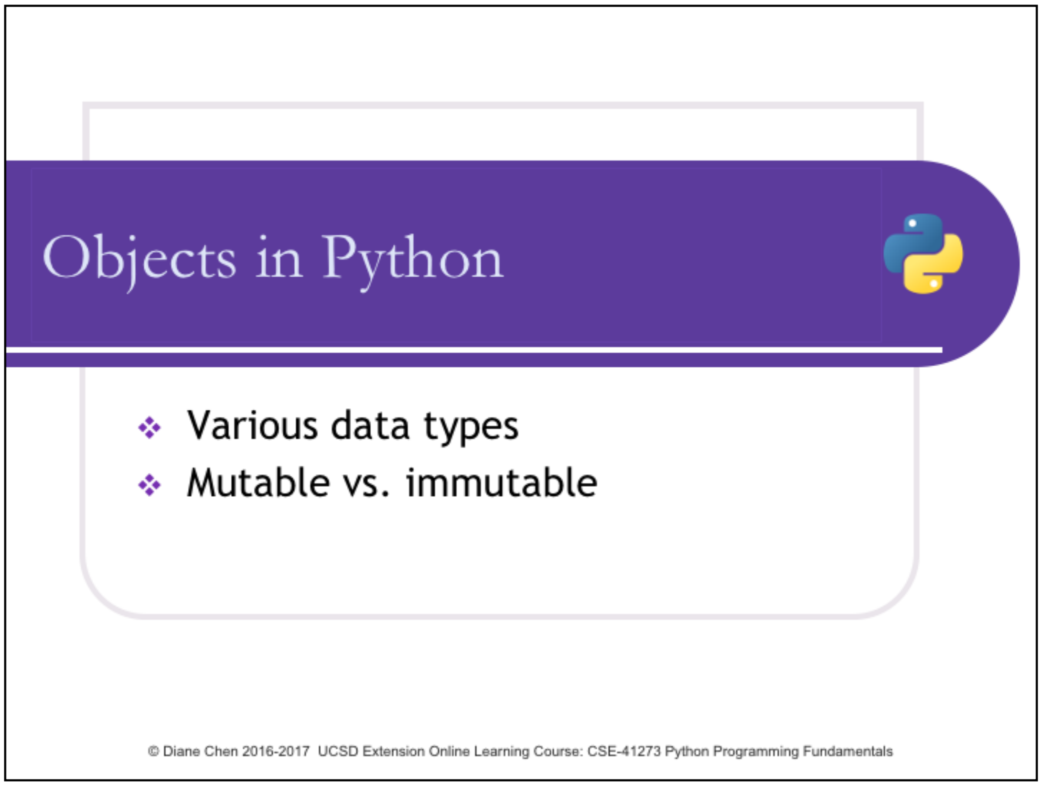


Python Functions

- The function "`type()`" is a built-in function.
- Python has many built-ins.
- Some functions only work on certain types

```
>>> first_name = "Diane"
>>> data = 12.75
>>> len(first_name)
5
>>> len(data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'float' has no len()
>>> len(str(data))
5
>>> 
```

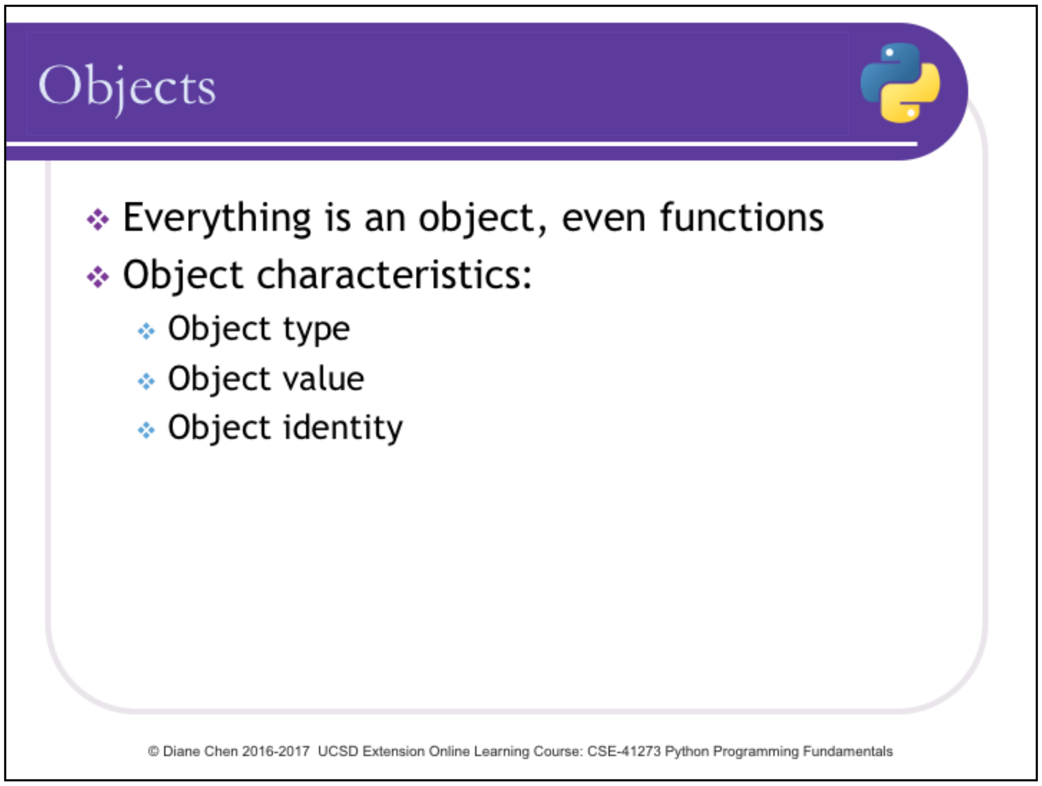
Python has many different built-in functions. We've seen the type() function already. There is another useful function, the len() function. It will tell us the number of characters in a string. But look what happens when we try it on a floating point number! Python gives us an error. The len() function doesn't work because it doesn't make any sense to ask the length of a floating point number. If we want to know how many *characters* are in the representation of the floating point number in 'data', we can use the str() function to convert the number to a string and then we can get the length. Later I will go into some of the more common built-in functions, and we will see others as the class continues.



Objects in Python

- Various data types
- Mutable vs. immutable

Now I want to talk about Python's object types that are used to hold and represent different kinds of data. Some data types can be modified, and others cannot. Objects that can be modified are called mutable types, and the ones that cannot be modified are called immutable types. As the class goes on, we will see how this difference affects the way we handle variables and their data.

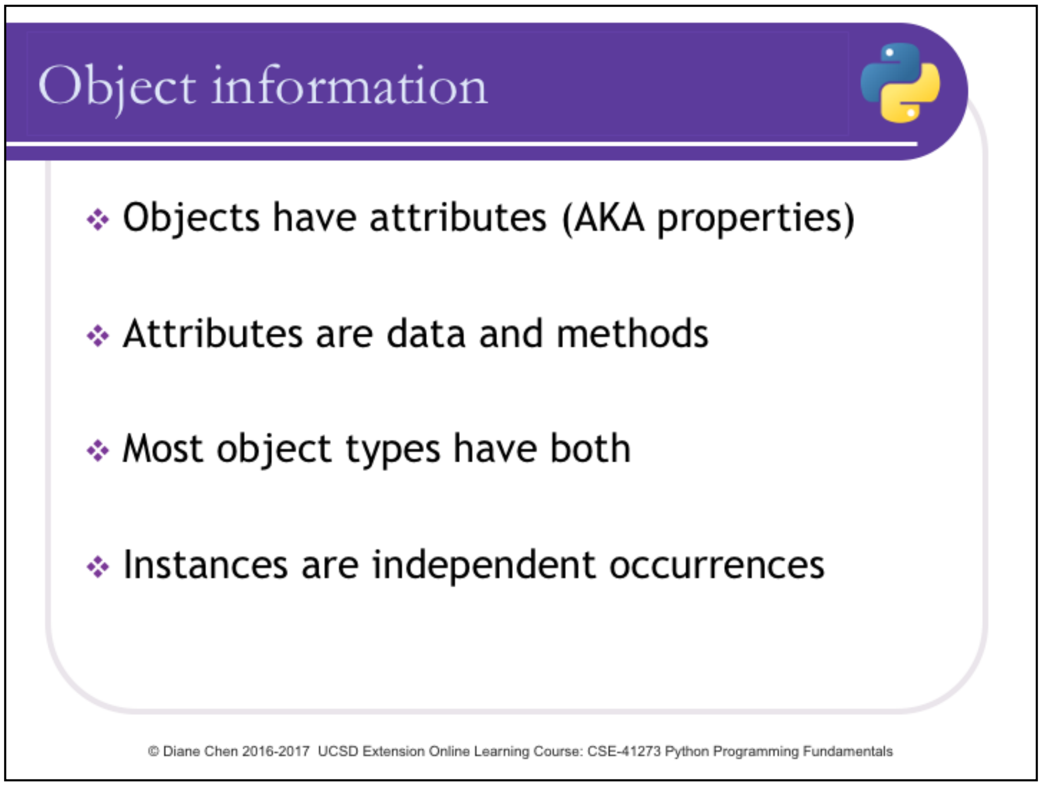


Objects

- Everything is an object, even functions
- Object characteristics:
  - Object type
  - Object value
  - Object identity

Python contains many data types as part of the core language. It's important to understand that pretty much everything in Python is represented by objects and by relationships between objects. As I mentioned, there are both mutable types and immutable types of objects. We will see later how this works for the variables of these different types. Even functions are objects that can be referenced by variables and passed around just like other variables and other objects.

Each object in Python has three characteristics: object type, object value, and object identity. The Object type tells Python what kind of an object it's dealing with. The type could be a number, or a string, or a list, or something else. The Object value is the data value that is contained by the object. And you can think of object identity as an identity code for the object. Variables in Python reference the identities of the objects. This identity is the memory location of the object, and *doesn't change for the object*. Note that this is not the same thing as a variable, since variables simply reference the objects, like handles of the bucket of the object data.

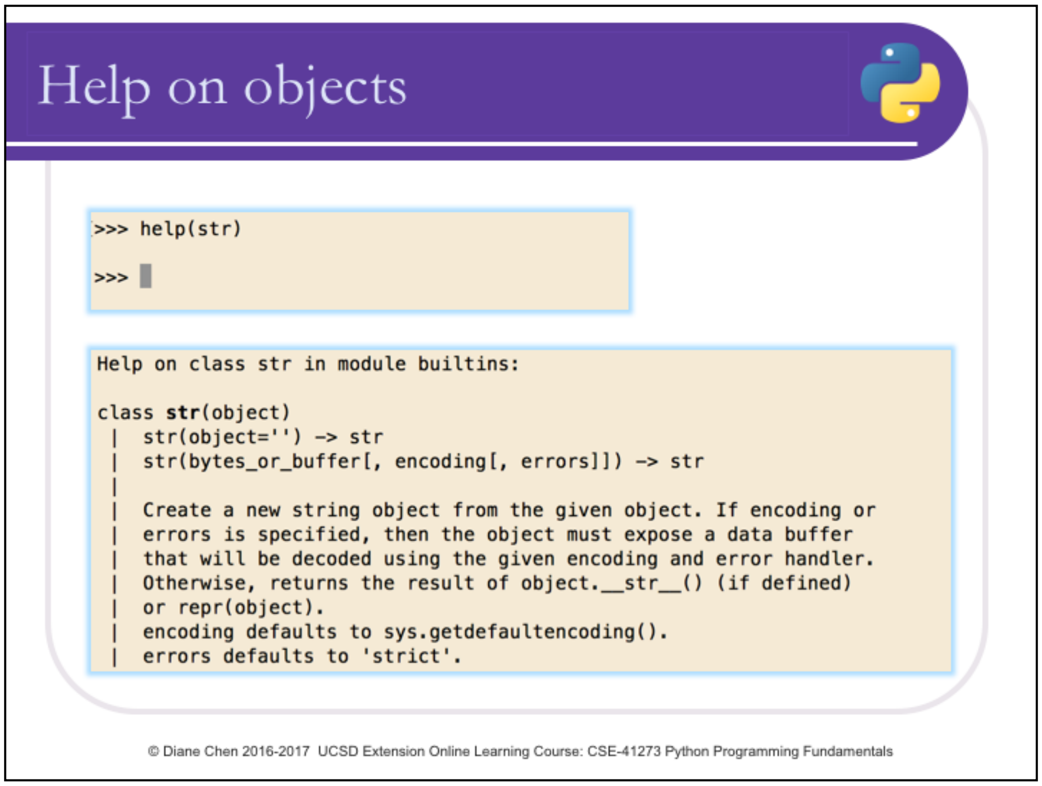


Object information

- Objects have attributes (AKA properties)
- Attributes are data and methods
- Most object types have both
- Instances are independent occurrences

Most Python objects have either data or functions (that we call methods) or both associated with them. These are known as attributes of the object. The way we refer to them is by the variable name of the object, followed by a dot, followed by the name of the attribute. The two types of attributes are data attributes or methods. A data attribute is a piece of data that is attached to the specific object. A method is a function that is attached to an object and usually performs some function or operation on that object. The object type always determines the methods that are available for the object. Most of the time when someone refers to the attribute of an object, they are really talking about data attributes, however, the methods technically are also attributes. Data attributes are also called properties.

We often refer to “instances”; an instance is one separate occurrence of an object. So two instances of the same object type could have different data values, but they would have the same methods available to operate on them. For example, if we have “x = 14” and we also have ”y = 42”, these variables are referencing two distinct instances of type ”int”.



Help on objects

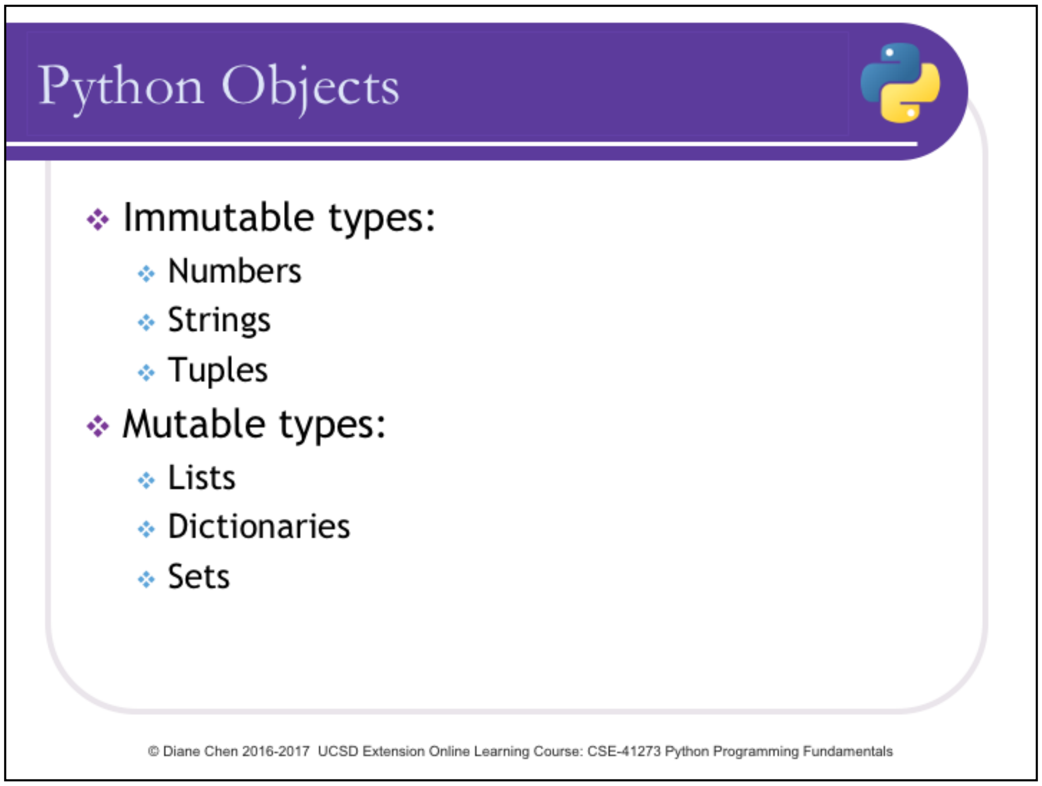
```
>>> help(str)

>>> 
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
```

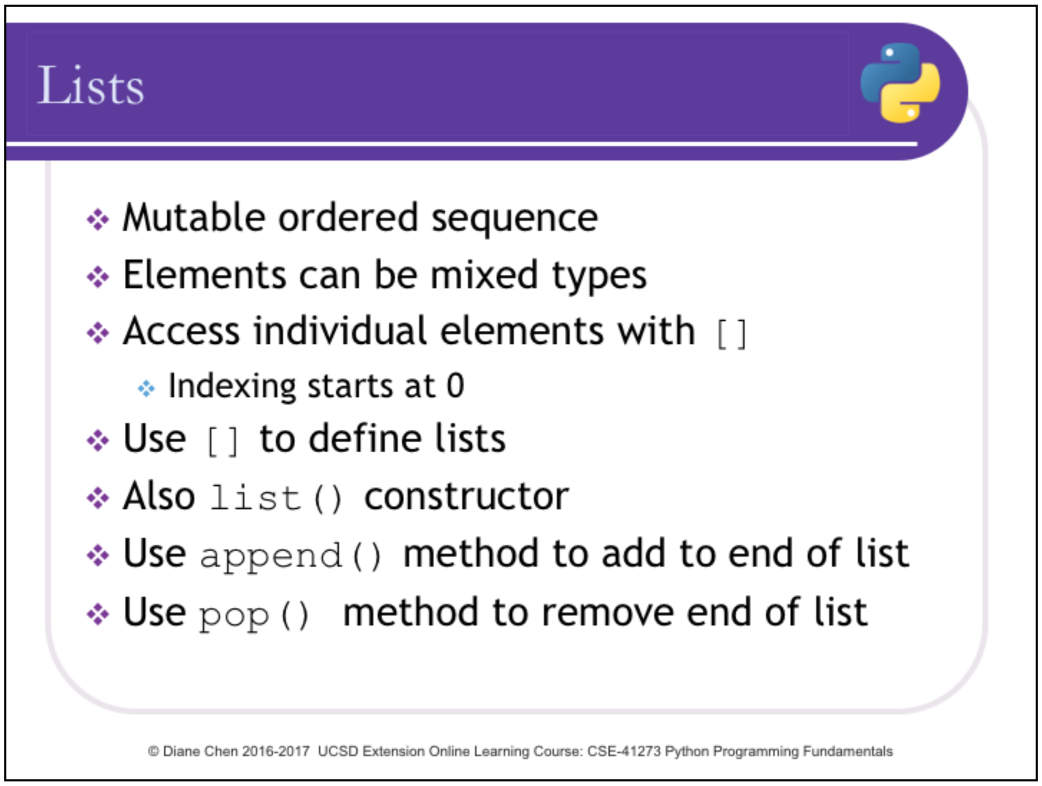
You can use the help() function in the REPL to get information on objects, including any attributes and methods that exist for that type. You can type help(variable) or help(type) to get the information. You can ignore all the methods it shows that start and end with two underscore characters. These are what we call “dunder” methods, where “dunder” stands for “double underscore”. Because it is just too much trouble to say something like “double underscore str double underscore”! It is much easier to say “dunder str”. We will see more of dunder methods in the lessons about classes. You should get used to using the help() function in the REPL to find information about behaviors of objects. Note that when you do help(object) it displays the information in an overlay, one page at a time. Use the space bar to advance to the next page, or the Enter key will advance one line at a time. To exit the help, type the letter Q.



Python Objects

- Immutable types:
  - Numbers
  - Strings
  - Tuples
- Mutable types:
  - Lists
  - Dictionaries
  - Sets

This is a list of the main basic Python object types that I'll be talking about. We've already seen numbers and strings. Lists are what you would think, namely, a list of values collected together. What is a tuple? A tuple is just like a list, but they are immutable. It may seem like a useless data type, but we'll go into how they can be very useful for certain things. Dictionaries are unordered collections of key-value pairs. Sets are unordered collections of unique items.



Lists

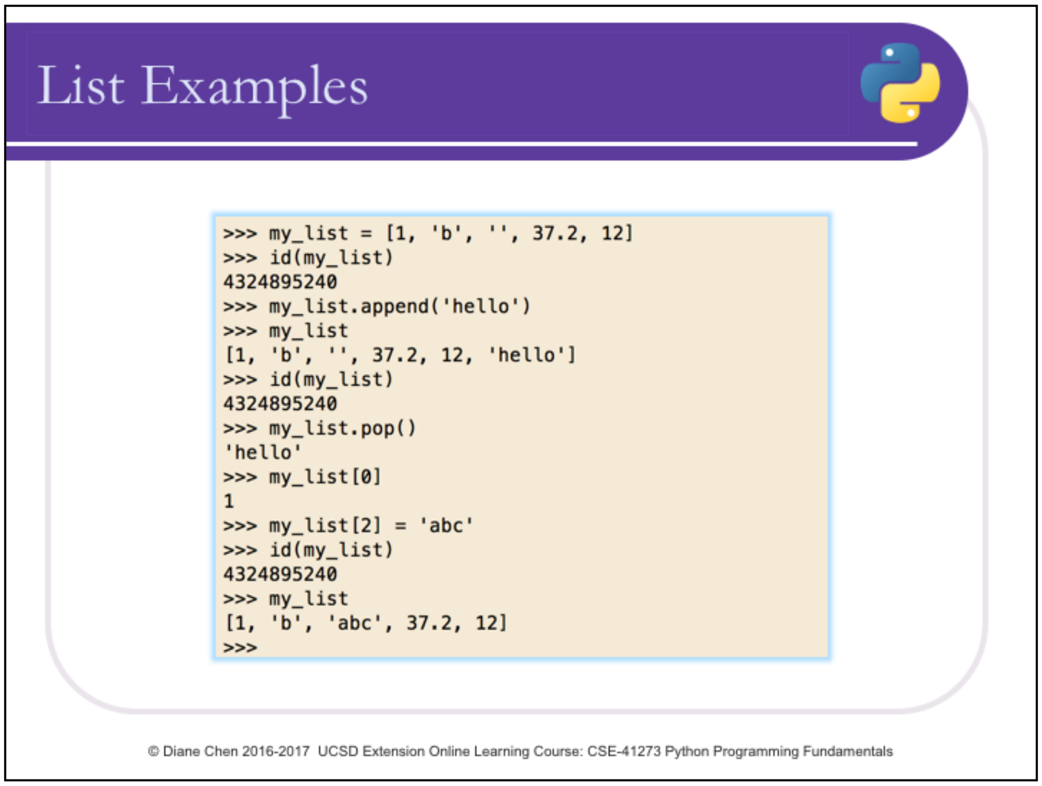
- Mutable ordered sequence
- Elements can be mixed types
- Access individual elements with `[]`
  - Indexing starts at 0
- Use `[]` to define lists
- Also `list()` constructor
- Use `append()` method to add to end of list
- Use `pop()` method to remove end of list

Let's talk about lists first, since they are a very common data type. A list is an ordered sequence of objects. The elements that make up the list can be of any type. We can access individual elements of the list by indexing using the square brackets, with indexing always beginning at 0.

Square brackets are also used to define a list. You can also use the list constructor to make a new list from other objects. Most of the time, the square brackets are the easiest to use.

Use the append() method to add items to the end of a list. Likewise, the pop() method removes the last item in a list. The pop() method returns the value of the item removed, so the REPL prints it for you. There are a number of other methods that operate on lists; you can use help() in the REPL or take a look at the online docs to see them all.

Lists are mutable, which means that the individual members of the list can be modified.



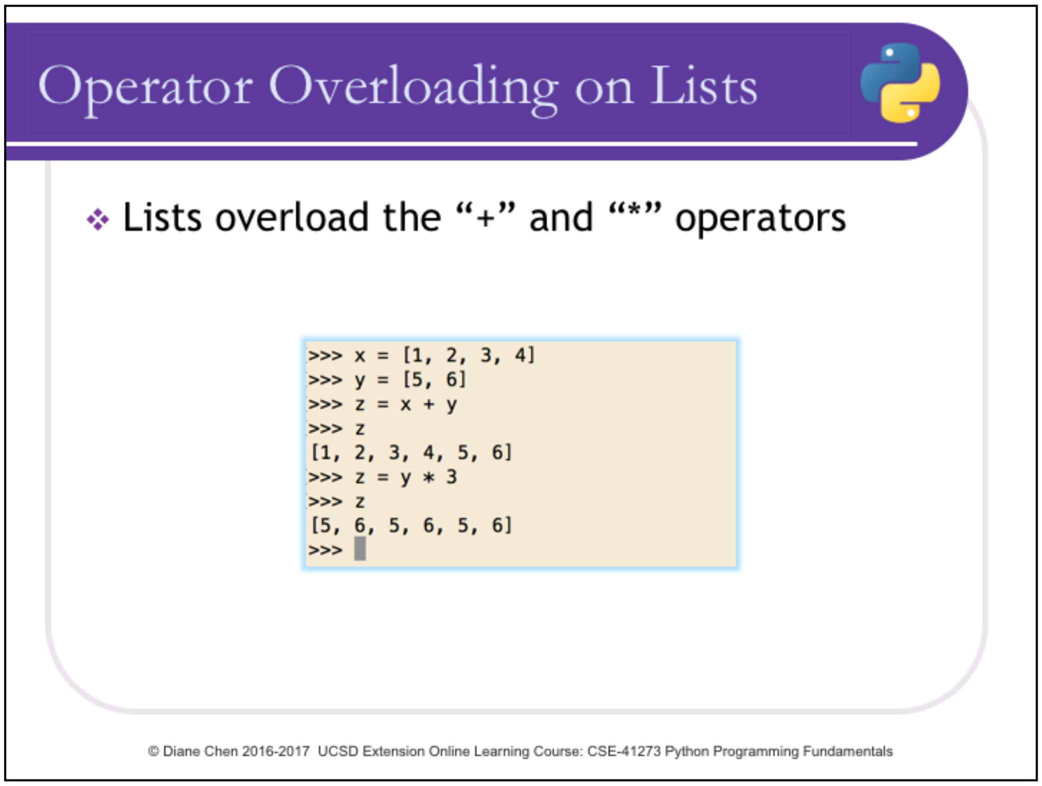
List Examples

```
>>> my_list = [1, 'b', '', 37.2, 12]
>>> id(my_list)
4324895240
>>> my_list.append('hello')
>>> my_list
[1, 'b', '', 37.2, 12, 'hello']
>>> id(my_list)
4324895240
>>> my_list.pop()
'hello'
>>> my_list[0]
1
>>> my_list[2] = 'abc'
>>> id(my_list)
4324895240
>>> my_list
[1, 'b', 'abc', 37.2, 12]
>>>
```

Here I have a list called my_list containing a variety of items in it. I can append to the list, and when I use the method pop, it removes the item. The value of the item removed is returned by the pop() method, so the REPL prints it for us here.

To reference a specific element of the list we use the square bracket notation. We can display them or change them individually.

There is a function id() in Python, that tells you the id of an object. It is essentially the memory location of the object that the variable is referencing. Note that I show the id of the list and id doesn't change even when we make changes to the list. This is because the list is a mutable object, which means we can change any part of it at any time. We'll see this function again later.



Operator Overloading on Lists

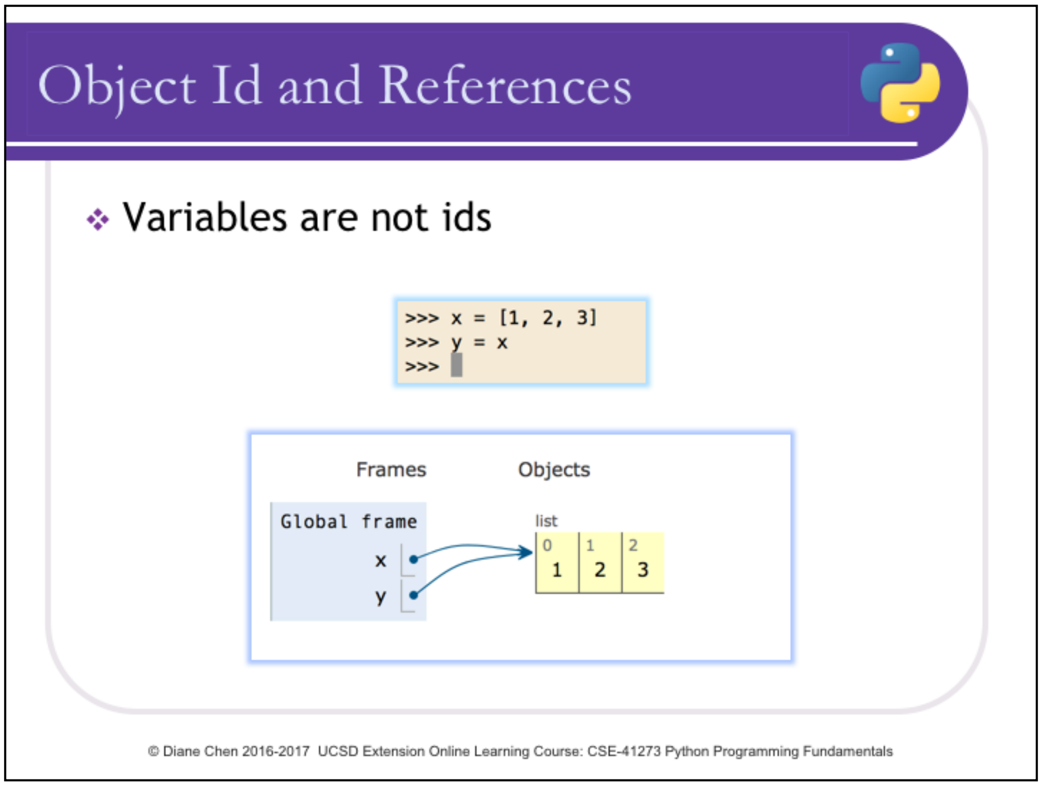
- Lists overload the "+" and "*" operators

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6]
>>> z = x + y
>>> z
[1, 2, 3, 4, 5, 6]
>>> z = y * 3
>>> z
[5, 6, 5, 6, 5, 6]
>>> 
```

You may have heard the term operator overloading before. What does it mean? When we take an operator, for example the plus sign for numerical addition, and make it operate on objects that are not numerical, this is operator overloading. We have overloaded the meaning of the "addition" operator to include the other objects.

Lists have the addition and multiplication operators overloaded. You can take 2 lists, and "add" them together with the plus sign, and the result is a new list containing the elements of the first list, followed by the elements of the second list.

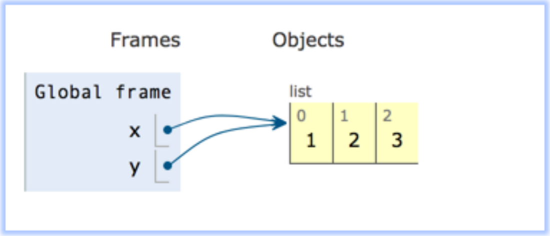
Similarly, you can "multiply" a list with an integer value, and the result is a new list containing the members of the original list repeated the number of times specified.



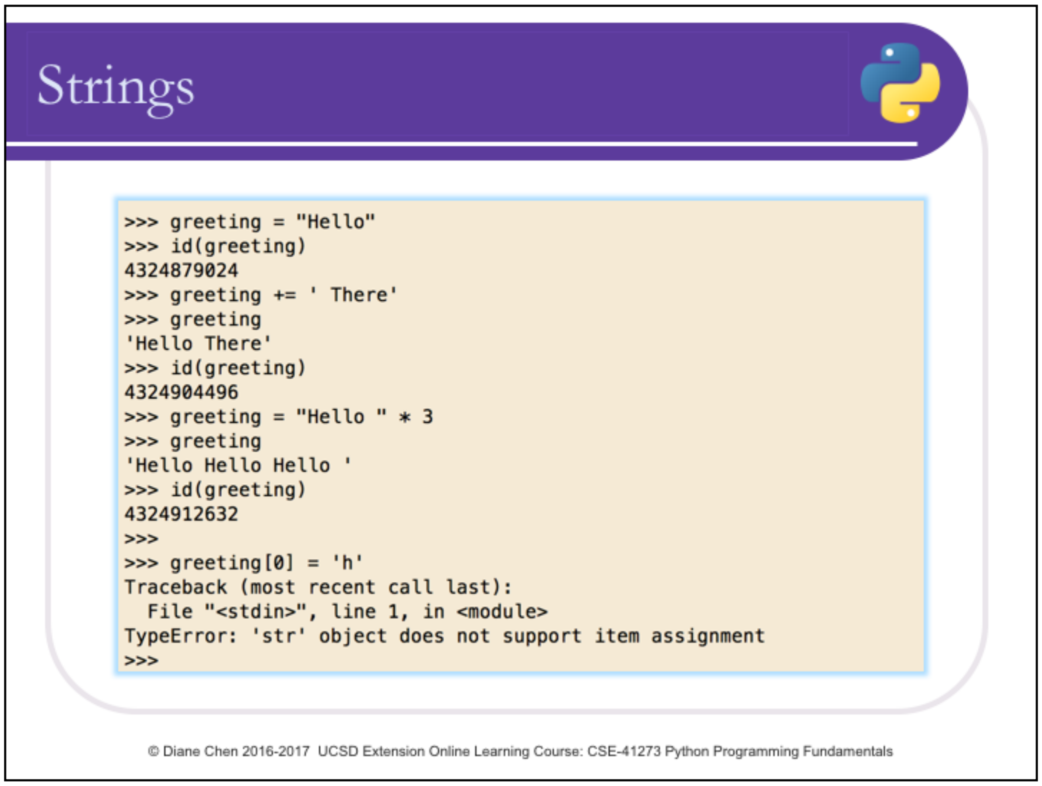
Object Id and References

- Variables are not ids

```
>>> x = [1, 2, 3]
>>> y = x
>>>
```



This is as a good place as any to take a minute to talk about references and object Ids. Every object has an id, as I mentioned before. I want to emphasize that variables are not ids. Variables are references to the object. When we have a list variable, the variable points to the location in memory where the list exists. If you take a new variable and make it equal to the first list variable, you are not copying the list! This is very important to understand. Here is a code snippet and a picture from pythontutor.com showing what happens when the code is executed. The list object exists in memory at some location. After the code executes, the variables x and y BOTH point to the SAME list. This is one of the most important things to remember. I wrote a blog post about this issue; it is linked in the Notes for the week. Pease read it after you have finished the lectures for this week.



Strings

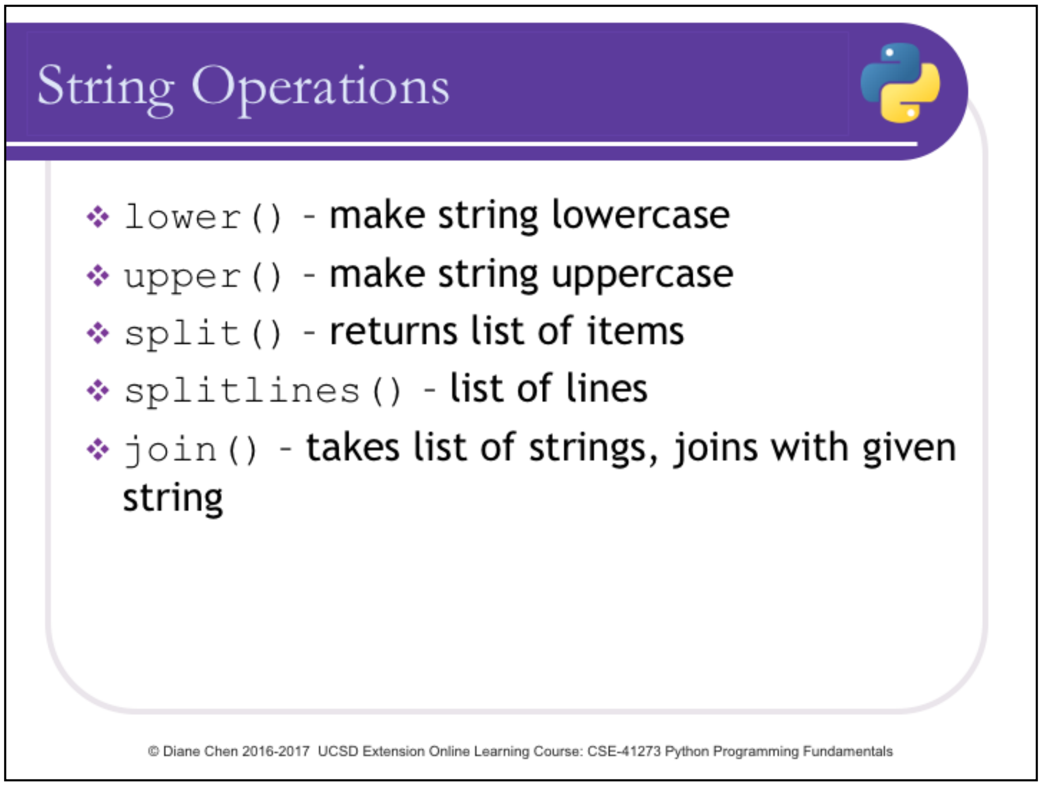
```
>>> greeting = "Hello"
>>> id(greeting)
4324879024
>>> greeting += ' There'
>>> greeting
'Hello There'
>>> id(greeting)
4324904496
>>> greeting = "Hello " * 3
>>> greeting
'Hello Hello Hello '
>>> id(greeting)
4324912632
>>>
>>> greeting[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

We've seen strings just a little so far. Internally, strings are like lists of characters. We can add strings together and multiply them with integers, just like lists.

However, note that string objects are immutable; that is, they cannot change. Whenever you *think* you do something to change a string variable, you can be assured that what is happening behind the scenes is that a new string object is created (with a new id) and the variable is changed to point to the new object.

We can index individual characters of a string in the same manner as a list. But only to refer to or read the values – if we try to modify an element of the string, we get an error because we are trying to modify an immutable object.

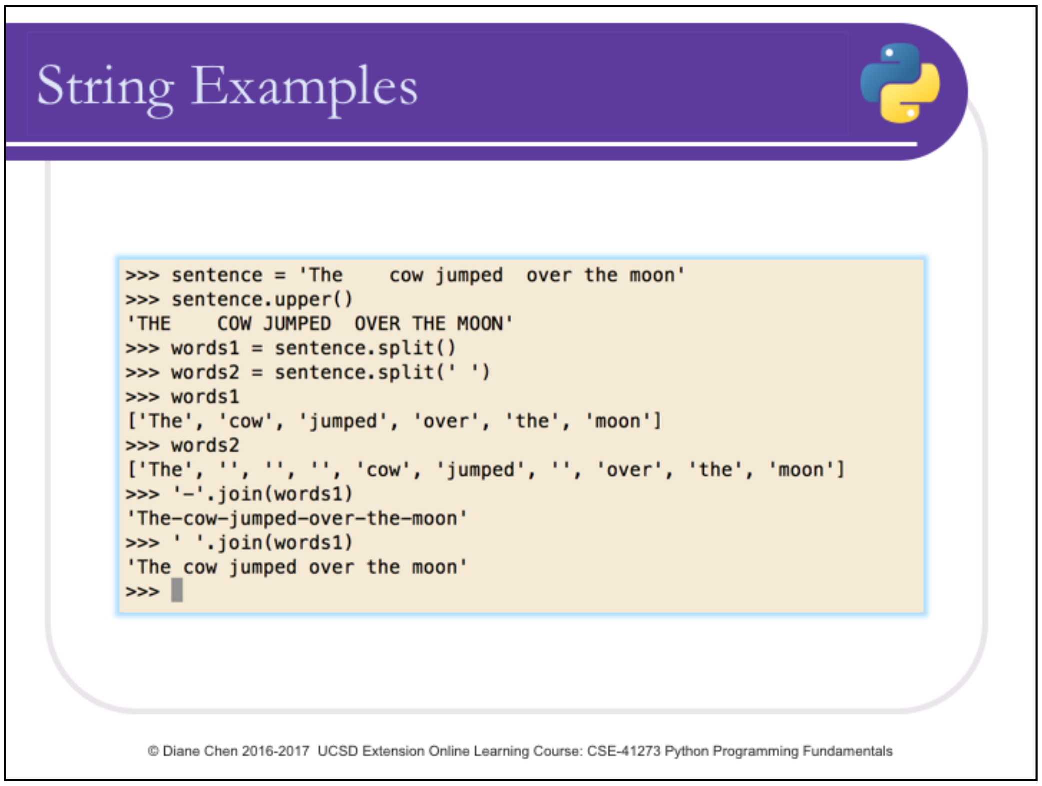
So again, be assured that you can change string variables as needed, but what happens is your variable references a new string object after the "change".



String Operations

- `lower()` - make string lowercase
- `upper()` - make string uppercase
- `split()` - returns list of items
- `splitlines()` - list of lines
- `join()` - takes list of strings, joins with given string

There are many special methods that operate on strings. You can see them by using the help() function in the REPL, or you can go to the documentation page for methods on strings. A link is provided to the documentation in the Notes and Resources page. Here are a few that are pretty common. The lower and upper methods are pretty obvious. The split method takes the string and returns a list of strings split apart by the given separator. If no separator is given it defaults to whitespace, which is defined as spaces, tabs and line breaks. Splitlines returns a list of strings based on splitting on line terminator characters. Join is a very handy method, but is a strange one – it seems counterintuitive the way it works. It takes the original string and uses it like a separator to join together all the strings given as input.



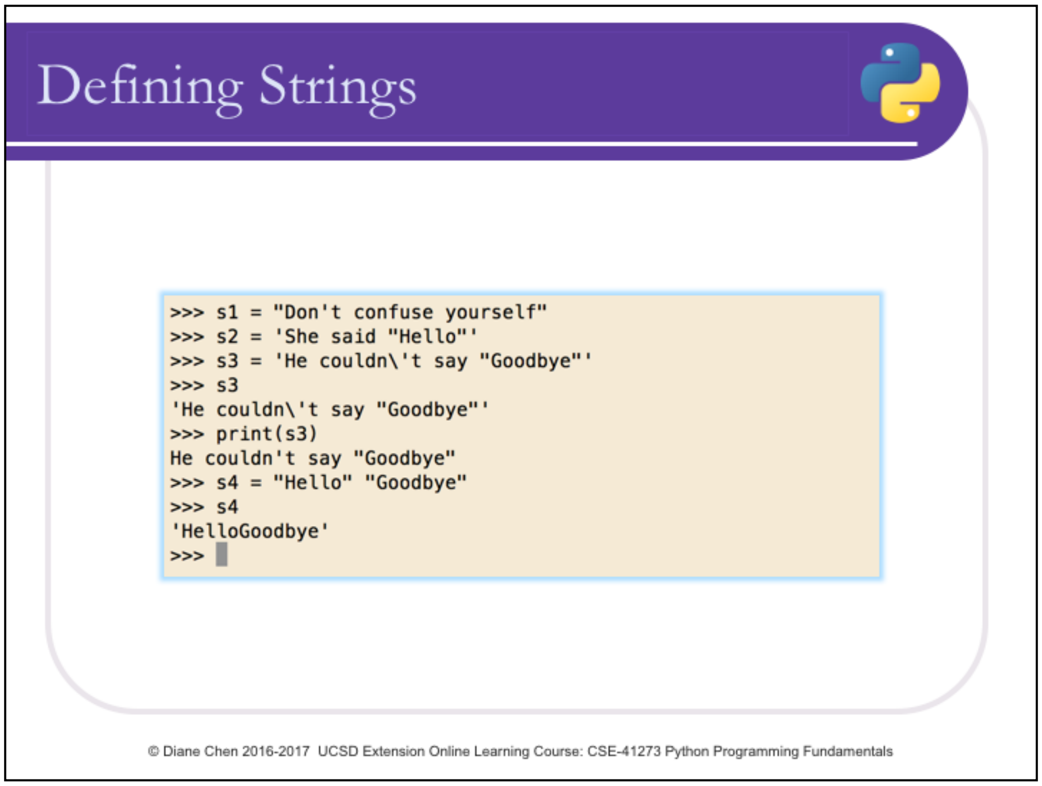
String Examples

```
>>> sentence = 'The    cow jumped  over the moon'
>>> sentence.upper()
'THE    COW JUMPED  OVER THE MOON'
>>> words1 = sentence.split()
>>> words2 = sentence.split(' ')
>>> words1
['The', 'cow', 'jumped', 'over', 'the', 'moon']
>>> words2
['The', '', '', '', 'cow', 'jumped', '', 'over', 'the', 'moon']
>>> '-'.join(words1)
'The-cow-jumped-over-the-moon'
>>> ' '.join(words1)
'The cow jumped over the moon'
>>> 
```

Here are some examples of using string methods. We have a sentence, the cow jumped over the moon. Note that there are some extra spaces in there. Sentence.upper() returns a new string with all the letters uppsercase. Take a look at the two different usages of split(). The words1 list is created using the default split(), while words2 is created by splitting on the space character. When we use the default, the system treats runs of whitespace as one unit of whitespace, so to speak. But if we split on the space character, then we get extra empty strings in our list result.

Using split() is also a good way of making a list of strings. If we know we want each element of the list to be a separate string, we could just type one string with all the words with spaces in between them and then use split to get our list. It looks a little cleaner and is much easier to type.

Here are examples of using join. The reason that the method works that way is that it has to be a method on strings, because that's what we are joining. So the string that we use is the separator and it joins the list of strings together.

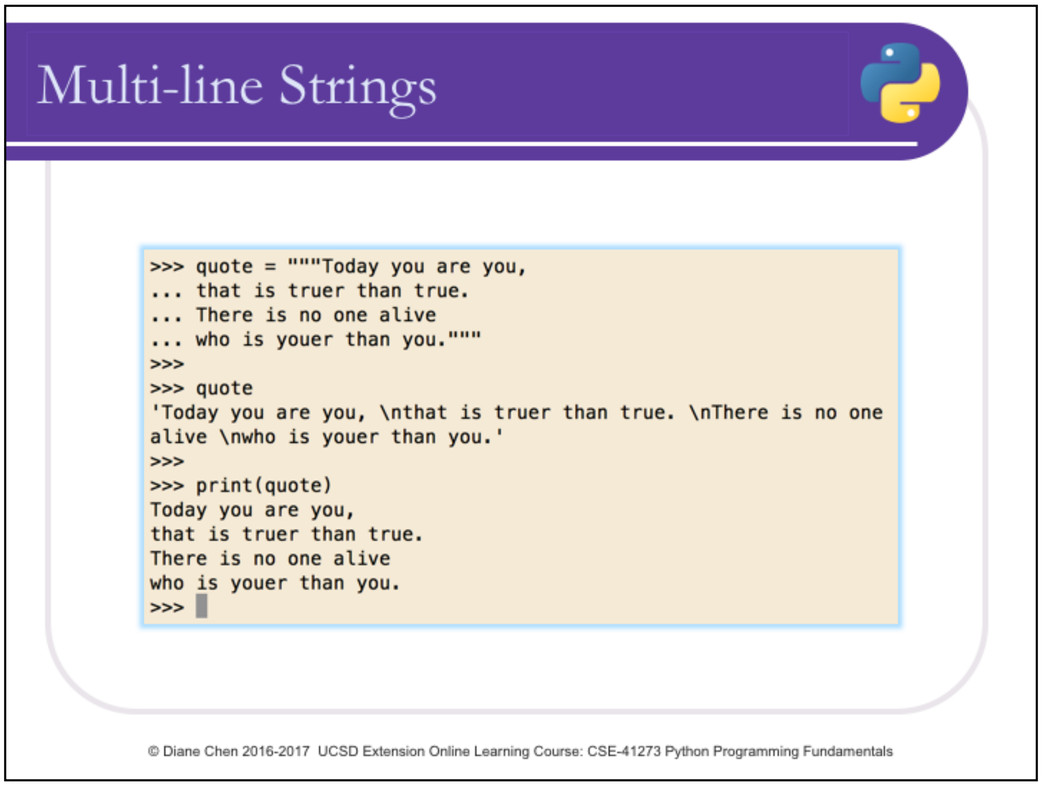


Defining Strings

```
>>> s1 = "Don't confuse yourself"
>>> s2 = 'She said "Hello"'
>>> s3 = 'He couldn\'t say "Goodbye"'
>>> s3
'He couldn\'t say "Goodbye"'
>>> print(s3)
He couldn't say "Goodbye"
>>> s4 = "Hello" "Goodbye"
>>> s4
'HelloGoodbye'
>>> 
```

You may have noted that I sometimes used a single quote and sometimes use a double quote when I define strings. Python doesn't care one way or another, unlike some other languages. As long as the single or double quotes match up, Python treats them as equal. This allows us to include single or double quotes inside strings easily, without needing to escape them. Of course, if you want both of them in one string, you will need to escape it. Note that when we just display the variable s3 in the REPL, we see the escape character, but when we print it with the print function, it prints as we expect it to. This is one of the differences between printing things at the REPL and with the print function. Think of the REPL as displaying a developer-friendly version of the variable data, while the print function produces a "people-friendly" version.

Another quirk of defining strings is that if you have 2 strings together, without anything between them, Python treats them as one string. Sometimes this is useful, but if you're not careful you can mess things up. I was defining a list of strings once and missed a comma in between two of the strings. Python didn't complain, but my list didn't look like it was supposed to!

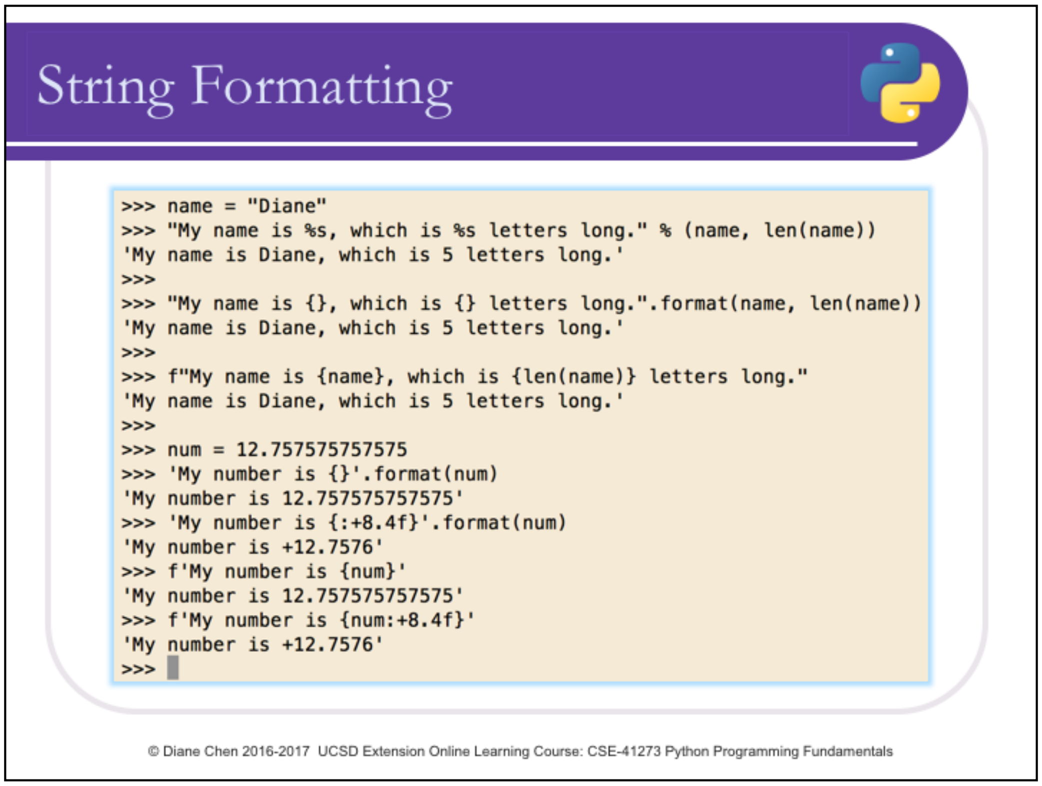


Multi-line Strings

```
>>> quote = """Today you are you,
... that is truer than true.
... There is no one alive
... who is youer than you."""
>>>
>>> quote
'Today you are you, \nthat is truer than true. \nThere is no one alive \nwho is youer than you.'
>>>
>>> print(quote)
Today you are you,
that is truer than true.
There is no one alive
who is youer than you.
>>> 
```

Python also has multi-line strings. These use three quotes to define the start and end. You can use three single quotes or three double quotes. I prefer using 3 double-quote characters, because then there is no confusion of whether you are seeing 3 single quotes or a double quote and one single quote.

Let's see an example with this Dr. Seuss quote. Once Python finds a triple quote, it keeps reading until it gets to another set of triple quotes. In the Python REPL, when it knows there should be more to come, it changes the prompt to be 3 dots instead of the usual greater than symbol. Like I showed on the last slide, when you display the variable from the REPL, it shows all the characters, so you see the newline character represented by the backslash n. But when we print it with the print function, then all the lines are displayed separately as we would expect.

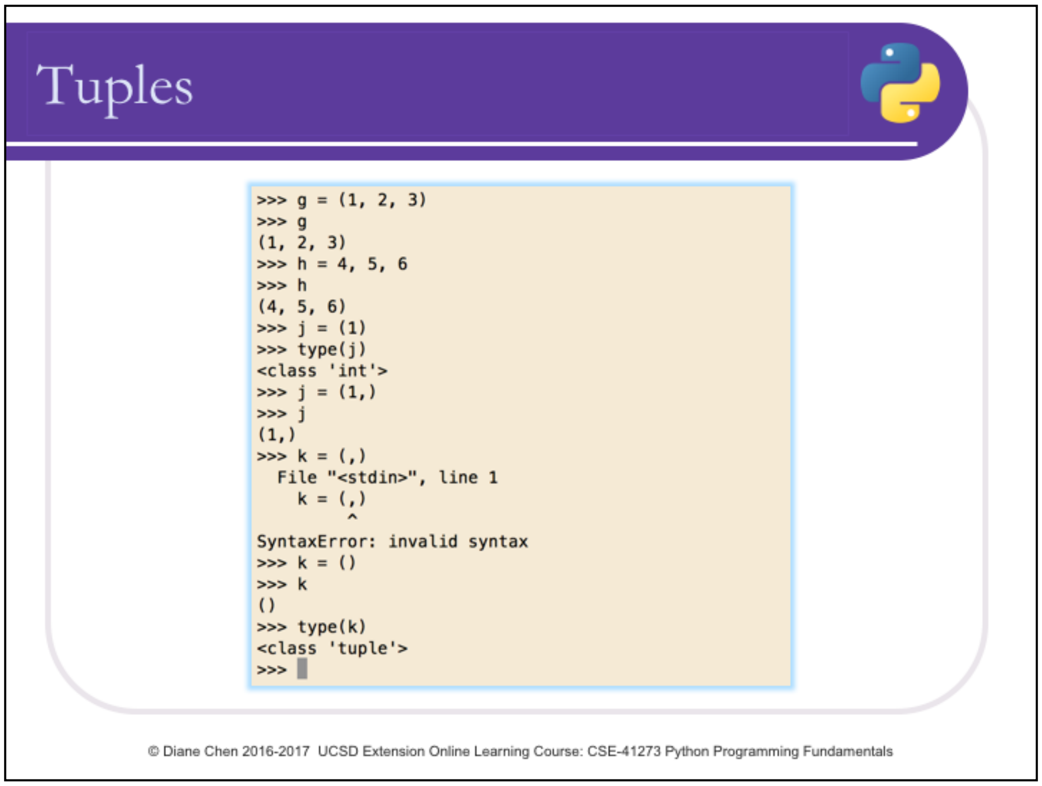


String Formatting

```
>>> name = "Diane"
>>> "My name is %s, which is %s letters long." % (name, len(name))
'My name is Diane, which is 5 letters long.'
>>>
>>> "My name is {}, which is {} letters long.".format(name, len(name))
'My name is Diane, which is 5 letters long.'
>>>
>>> f"My name is {name}, which is {len(name)} letters long."
'My name is Diane, which is 5 letters long.'
>>>
>>> num = 12.757575757575
>>> 'My number is {}'.format(num)
'My number is 12.757575757575'
>>> 'My number is {:+8.4f}'.format(num)
'My number is +12.7576'
>>> f'My number is {num}'
'My number is 12.757575757575'
>>> f'My number is {num:+8.4f}'
'My number is +12.7576'
>>>
```

There are lots of ways to format a string, when you want to put variables into a string. Most of the time, this is for printing purposes, but it could be for other reasons too. This is where Python can get very confusing. For a long time there were 2 ways to format strings, the “old” way using the percent symbol syntax, and the “new” way using the string format() method. The format() method was added to address some of the drawbacks of the old % syntax, and it is very powerful and flexible. However, it can be quite verbose and has some limitations for certain applications. In the notes, I have a link to pyformat.info, a great website to go to for help in figuring out any special formatting syntax. Those of you familiar with C will recognize that it is derived from the formats for printf() and sprintf().

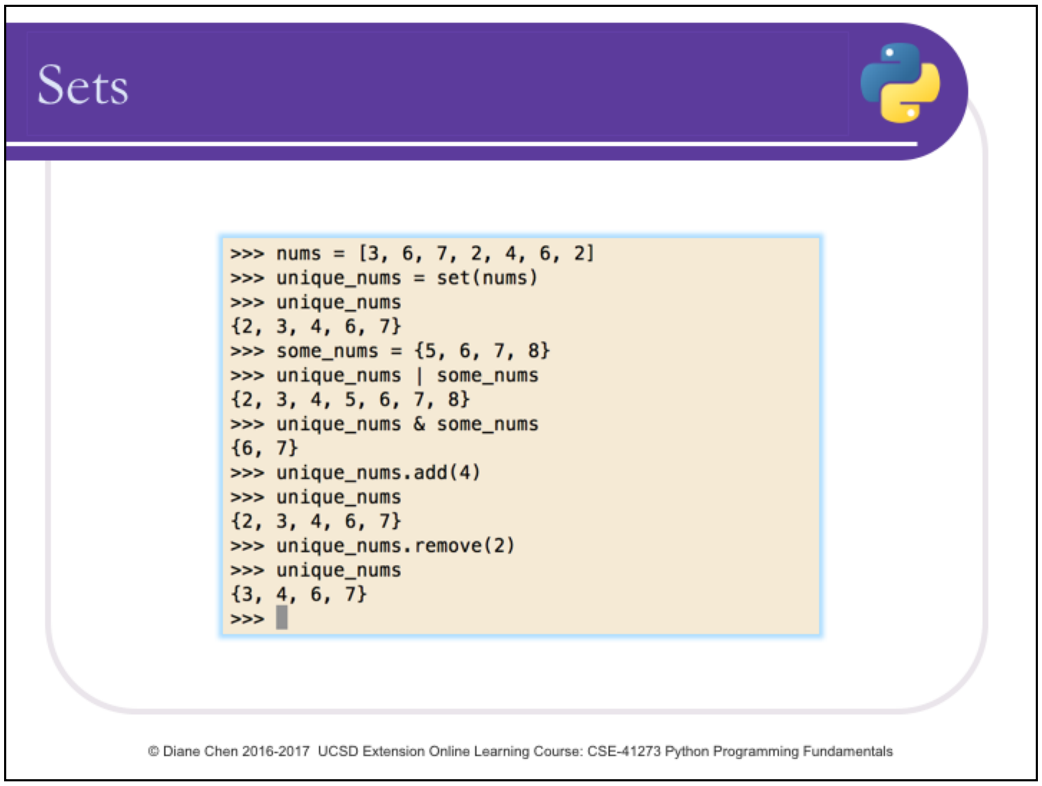
Python 3.6 added yet another “new new” way of formatting strings, called f-strings. The string is prefixed by the letter f, either lowercase or uppercase F. Expressions can be in-lined into the string itself. I have linked to the PEP-498 with details and explanation of the f-string format. Here I show some examples of formatting a floating point number in the “new” way with the format() method, and also with the “new new” way of using f-strings.



Tuples

```
>>> g = (1, 2, 3)
>>> g
(1, 2, 3)
>>> h = 4, 5, 6
>>> h
(4, 5, 6)
>>> j = (1)
>>> type(j)
<class 'int'>
>>> j = (1,)
>>> j
(1,)
>>> k = (,)
  File "<stdin>", line 1
    k = (,)
         ^
SyntaxError: invalid syntax
>>> k = ()
>>> k
()
>>> type(k)
<class 'tuple'>
>>> 
```

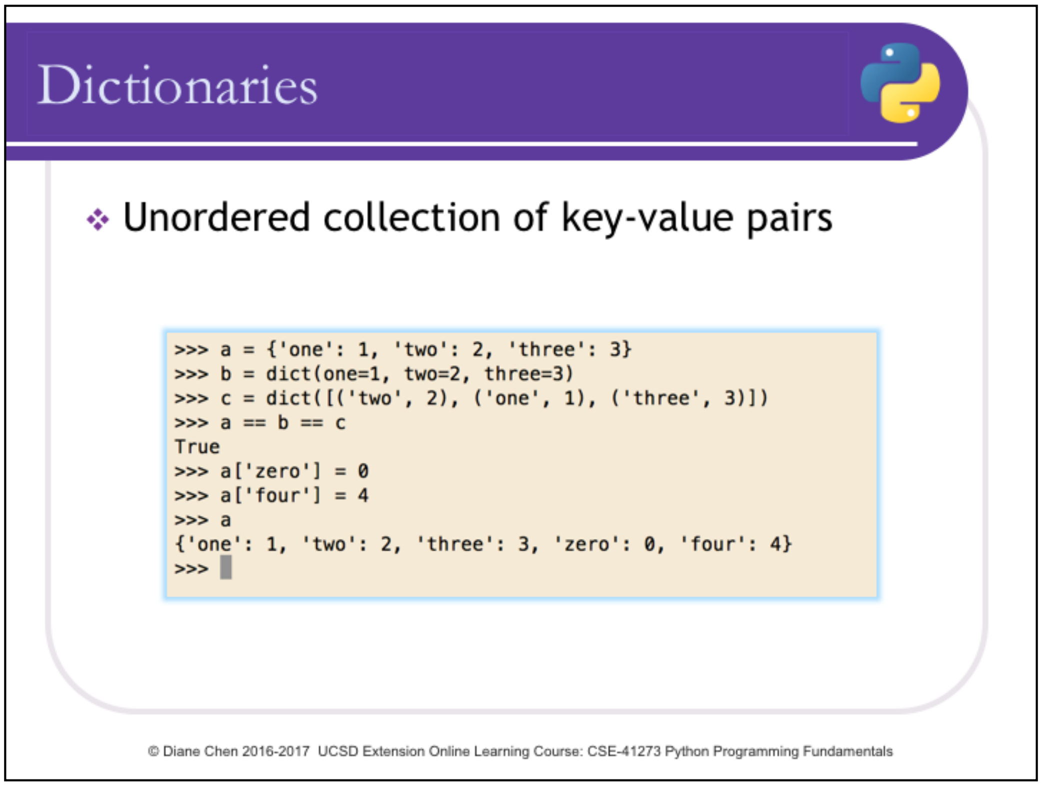
Tuples are immutable sequences – just like lists, except they are immutable. This may seem like it isn't very useful, but we will see as the class goes on that there is definitely a place for Tuples. You create them with commas and sometimes optional parentheses. Here both g and h are tuples. When the REPL prints the tuples, they are always shown with the parentheses. The format is a little inconsistent, because if you want a tuple with one element, you still need the comma, otherwise Python interprets the parentheses as precedence or grouping indicator. You can create an empty tuple with just the parentheses – no comma. There is also a tuple constructor function. Tuple elements can be accessed using square brackets and the element index, just like lists; although they cannot be modified.



Sets

```
>>> nums = [3, 6, 7, 2, 4, 6, 2]
>>> unique_nums = set(nums)
>>> unique_nums
{2, 3, 4, 6, 7}
>>> some_nums = {5, 6, 7, 8}
>>> unique_nums | some_nums
{2, 3, 4, 5, 6, 7, 8}
>>> unique_nums & some_nums
{6, 7}
>>> unique_nums.add(4)
>>> unique_nums
{2, 3, 4, 6, 7}
>>> unique_nums.remove(2)
>>> unique_nums
{3, 4, 6, 7}
>>> 
```

Sets are unordered collections of unique items. By unordered I mean that the individual elements cannot be accessed individually by index, as in lists or tuples. Here I first create a list of numbers, some of which are duplicates. We can use the set constructor to convert the list into a set of unique elements. To create a set, we can use the curly brackets or braces. Sets have overloaded operators for union, intersection, and other set operations. The comparison operators are also overloaded for testing subsets, disjointedness and other set notations. We can add or remove items from a set. If we add an element that is already there, it is just ignored; no error is raised.



Dictionaries

- Unordered collection of key-value pairs

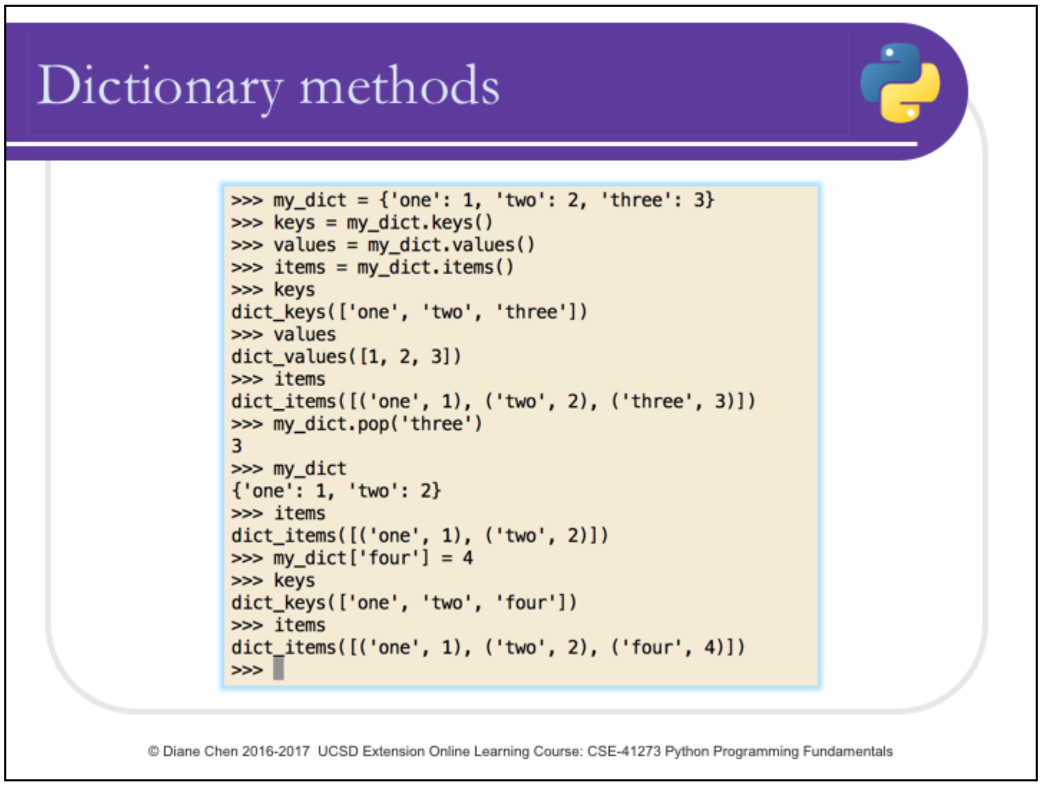
```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> b = dict(one=1, two=2, three=3)
>>> c = dict([('two', 2), ('one', 1), ('three', 3)])
>>> a == b == c
True
>>> a['zero'] = 0
>>> a['four'] = 4
>>> a
{'one': 1, 'two': 2, 'three': 3, 'zero': 0, 'four': 4}
>>> 
```

Dictionaries are arguably the most useful and interesting data structures. In some other languages they are referred to as hashmaps. They consist of key-value pairs, where the key acts as an ”index” of sorts. They are a mutable data structure. The key objects must be an immutable type, like numbers, strings or tuples. They are unordered, in that the ordering is not guaranteed. In Python 3.6, a lot of work was done in the standard CPython implementation to make dictionaries more efficient in both memory and time. As a result, dictionaries ended up being ordered by the order that the key-value pairs are added to the dictionary. However the ordering is not guaranteed, and may change when dictionary items are added or removed. Prior to version 3.6, the ordering of dictionaries was arbitrary.

What is CPython? It is the standard implementation of Python, that you get when you download from Python.org. There are other implementations of the Python programming language, like IronPython for .NET, Jython running on a Java Virtual Machine, PyPy that is fast using a JIT compiler, and MicroPython for microcontrollers.

Here we see several different methods of creating dictionaries. It also uses the curly braces like sets, but Python can tell from the context (namely the colon separating the key and value) that this is going to be a dictionary rather than a set. We can add more pairs to the dictionary just by referencing the index and assigning it a value.

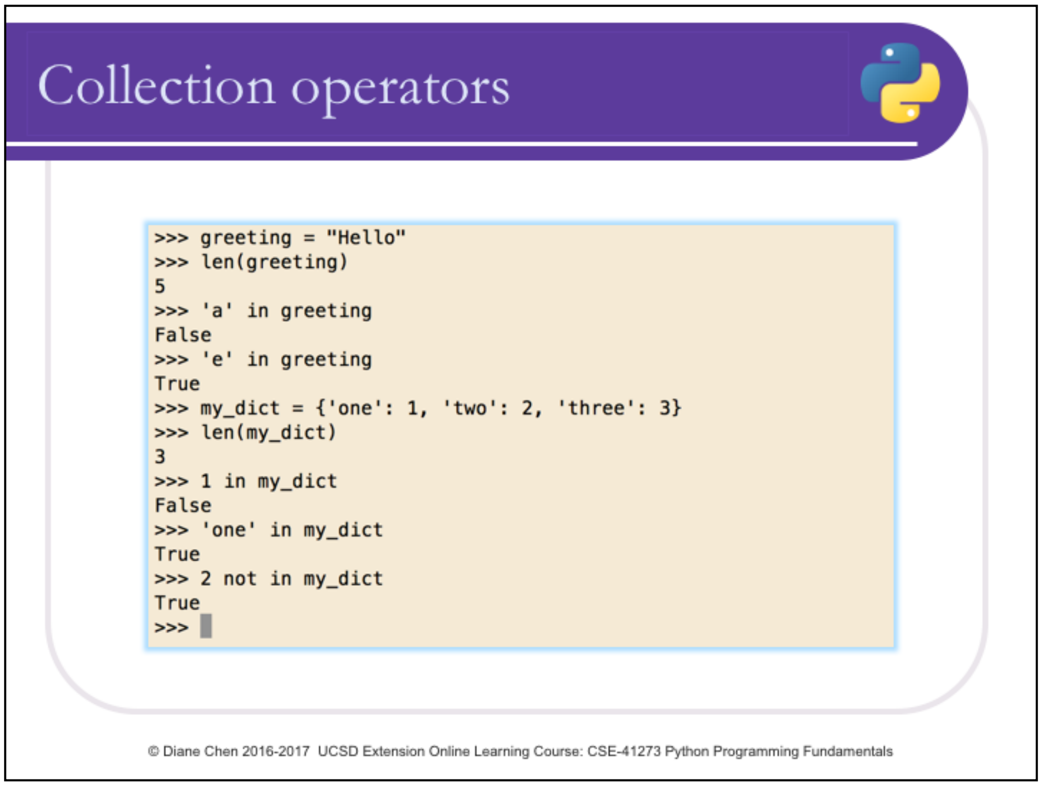
Dictionaries are mutable, which means that after you create them, you can modify them as you go along.



Dictionary methods

```
>>> my_dict = {'one': 1, 'two': 2, 'three': 3}
>>> keys = my_dict.keys()
>>> values = my_dict.values()
>>> items = my_dict.items()
>>> keys
dict_keys(['one', 'two', 'three'])
>>> values
dict_values([1, 2, 3])
>>> items
dict_items([('one', 1), ('two', 2), ('three', 3)])
>>> my_dict.pop('three')
3
>>> my_dict
{'one': 1, 'two': 2}
>>> items
dict_items([('one', 1), ('two', 2)])
>>> my_dict['four'] = 4
>>> keys
dict_keys(['one', 'two', 'four'])
>>> items
dict_items([('one', 1), ('two', 2), ('four', 4)])
>>> 
```

Dictionaries have a number of methods associated with them. Some of the most important ones are keys(), values(), items(), and pop(). The methods keys(), values() and items() return what are called “dictionary view objects”. That is, they contain a “view” into the dictionary that is dynamic and changes as the dictionary changes. The keys() and values() produce lists of keys and values, respectively. The items() produces a list of tuples where the first value is the key and the second value is the corresponding value. If we want to remove an item from the dictionary, we can use the pop() method, which requires a key as input and returns the value associated with the deleted key. You can see after each modification of the dictionary, the dictionary views have the correct changed data.



Collection operators

```
>>> greeting = "Hello"
>>> len(greeting)
5
>>> 'a' in greeting
False
>>> 'e' in greeting
True
>>> my_dict = {'one': 1, 'two': 2, 'three': 3}
>>> len(my_dict)
3
>>> 1 in my_dict
False
>>> 'one' in my_dict
True
>>> 2 not in my_dict
True
>>> 
```

Let’s talk about collection operators and functions. The function len() works on all collection types, whether they are ordered or unordered. Python also has the membership operator “in” and the two-word operator “not in”. These operators test for membership in the collection. Here we see the variable greeting that contains the string “hello”. There is no letter ‘a’ in greeting, so the ‘in’ operator returns False, whereas it returns True for the letter ‘e’. It also works on dictionaries, but only on the keys of the dictionary, not the values.