Python is an interpreted language, and the code is evaluated line-by-line. Since each line can be evaluated by itself, the time between evaluating each line doesn’t matter, and this allows us to have a REPL.

#### What is a REPL?

REPL stands for: **R**ead, **E**valuate, **P**rint, **L**oop

Each line is read, evaluated, the return value is then printed to the screen, and then the process repeats.

Python ships with a REPL, and you can access it by running python3.6 from your terminal.

$ python3.6

Python 3.6.4 (default, Jan 5 2018, 20:24:27)

[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>

The >>> indicates that you can type on that line. Later on, you’ll also see a ... which means that you are currently in a scoped area and will need to enter a blank line (no spaces) before it evaluates the entire code block.

The simplest use of this would be to do some math:

>>> 1 + 1

2

>>>

2 is the return value of the expression, and it is then printed to the screen. If something doesn’t have a return value, then nothing will be printed to the screen and you’ll see the next prompt immediately. We’ll cover this later, but an example would be None:

>>> None

>>>

Lastly, to exit the REPL, you can either type exit() (the parentheses are important), or you can hit Ctrl+d on your keyboard.

The most common immutable sequence type that we’re going to work with is going to be the tuple.

Since this is a course about Python scripting, we will be writing the majority of our code in scripts instead of using the REPL. To create a Python script we can create a file ending with the file extension of .py.

#### Creating Our First Python Script

Let’s create our first script to write our obligatory “Hello, World!” program:

$ vim hello.py

From inside this file, we can enter the lines of Python that we need. For the “Hello, World!” example we only need:

print("Hello, World!")

There are a few different ways that we can run this file. The first is by passing it to the python3.6 CLI:

$ python3.6 hello.py

Hello, World!

#### Setting a Shebang

You’ll most likely want your scripts to be:

1. Executable from anywhere (in our $PATH).
2. Executable without explicitly using the python3.6 CLI.

Thankfully, we can set the process to interpret our scripts by setting a shebang at the top of the file:

*hello.py*

#!/usr/bin/env python3.6

print("Hello, World")

We’re not quite done; now we need to make the file executable using chmod:

$ chmod u+x hello.py

Run the script now by using ./hello.py and we’ll see the same result. If we’d rather not have a file extension on our script, we can now remove that since we’ve put a shebang in the file mv hello.py hello, and running ./hello will still result in the same thing.

#### Adding Scripts to Our $PATH

Now we need to make sure that we can put this in our $PATH. For this course, we’ll be using a bin directory in our $HOME folder to store our custom scripts, but scripts can go into any directory that is in your $PATH.

Let’s create a bin directory and move our script:

$ mkdir ~/bin

$ mv hello ~/bin/

Here’s how we add this directory to the $PATH in our .bashrc (the .bashrc for this course already contains this):

$ export PATH=$HOME/bin:$PATH

Finally, run the hello script from our $PATH:

$ hello

Hello, World!

When writing scripts, we often want to leave ourselves notes or explanations. Python (along with most scripting languages) uses the # character to signify that the line should be ignored and not executed.

**Data Types**

Let’s learn about one of the core data types in Python: the str type.

#### Python Documentation For This Video

* [Strings (the str type)](https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str)

#### Strings

Open a REPL to start exploring Python strings:

$ python3.6

We’ve already worked with a string when we created our “Hello, World!” program. We create strings using either single quotes ('), double quotes ("), or triple single or double quotes for a multi-line string:

>>> 'single quoted string'

'single quoted string'

>>> "double quoted string"

'double quoted string'

>>> '''

... this is a triple

... quoted string

... '''

'\nthis is a triple\nquoted string\n'

Strings also work with some arithmetic operators.

We can combine strings using the + operator and multiply a string by a number using the \* operator:

>>> "pass" + "word"

'password'

>>> "Ha" \* 4

'HaHaHaHa'

A string is a sequence of characters grouped together. We need to cover the concept of an “Object” in object-oriented programming before moving on.

An “object” encapsulates two things:

1. State
2. Behavior

For the built-in types, the state makes sense because it’s the entire contents of the object. The behavior aspect means that there are functions that we can call on the instances of the objects that we have. A function bound to an object is called a “method”. Here are some example methods that we can call on strings:

find locates the first instance of a character (or string) in a string. This function returns the index of the character or string:

>>> "double".find('s')

-1

>>> "double".find('u')

2

>>> "double".find('bl')

3

lower converts all of the characters in a string to their lowercase versions (if they have one). This function returns a new string without changing the original, and this becomes important later:

>>> "TeStInG".lower() # "testing"

'testing'

>>> "another".lower()

'another'

>>> "PassWord123".lower()

'password123'

Lastly, if we need to use quotes or special characters in a string we can do that using the '’' character:

>>> print("Tab\tDelimited")

Tab Delimited

>>> print("New\nLine")

New

Line

>>> print("Slash\\Character")

Slash\Character

>>> print("'Single' in Double")

'Single' in Double

>>> print('"Double" in Single')

"Double" in Single

>>> print("\"Double\" in Double")

"Double" in Double

Let’s learn about some of the core data types in Python: the number types int and float.

#### Python Documentation For This Video

* [Numeric types (the int and float types)](https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex)

#### Numbers

There are two main types of numbers that we’ll use in Python, int and float. For the most part, we won’t be calling methods on number types, and we will instead be using a variety of operators.

>>> 2 + 2 # Addition

4

>>> 10 - 4 # Subtraction

6

>>> 3 \* 9 # Multiplication

27

>>> 5 / 3 # Division

1.66666666666667

>>> 5 // 3 # Floor division, always returns a number without a remainder

1

>>> 8 % 3 # Modulo division, returns the remainder

2

>>> 2 \*\* 3 # Exponent

8

If either of the numbers in a mathematical operation in Python is a float, then the other will be converted before carrying out the operation, and the result will always be a float.

#### Converting Strings and Numbers

Conversion is not uncommon since we need to convert from one type to another when writing a script and Python provides built-in functions for doing that with the built-in types. For strings and numbers, we can use the str, int, and float functions to convert from one type to another (within reason).

>>> str(1.1)

'1.1'

>>> int("10")

10

>>> int(5.99999)

5

>>> float("5.6")

5.6

>>> float(5)

5.0

You’ll run into issues trying to convert strings to other types if they aren’t present in the string

>>> float("1.1 things")

Traceback (most recent call last):

File "", line 1, in

ValueError: could not convert string to float: '1.1 things'

#### Single Line Comment

We can comment out a whole line:

# This is a full like comment

or we can comment at the end of a line:

2 + 2 # This will add the numbers

#### What About Block Comments?

Python does not have the concept of block commenting that you may have encountered in other languages. Many people mistake a triple-quoted string as being a comment, but it is not, it’s a multi-line string. That being said, multi-line strings can functionally work like comments, but they will still be allocated into memory.

"""

This is not a block comment,

but it will still work when you really need

for some lines of code to not execute.

"""

Learn about how Python represents truthiness and nothingness.

#### Python Documentation For This Video

* [Booleans & None](https://docs.python.org/3/library/stdtypes.html#truth-value-testing)

#### Booleans

Booleans represent “truthiness” and Python has two boolean constants: True and False.

Notice that these both start with capital letters. Later we will learn about comparisons operations, and those will often return either True or False.

#### Representing Nothingness with None

Most programming languages have a type that represents the lack of a value, and Python is no different. The constant used to represent nothingness in Python is None. None is a “falsy”, and we’ll often use it to represent when a variable has no value yet.

An interesting thing to note about None is that if you type None into your REPL, there will be nothing printed to the screen. That’s because None actually evaluates into nothing.

Almost any script that we write will need to have a way for us to hold onto information for use later on. That’s where variables come into play.

#### Working with Variables

We can assign a value to a variable by using a single = and we don’t need to (nor can we) specify the type of the variable.

>>> my\_str = "This is a simple string"

Now we can print the value of that string by using my\_var later on:

>>> print(my\_str)

This is a simple string

Before, we talked about how we can’t change a string because it’s immutable. This is easier to see now that we have variables.

>>> my\_str += " testing"

>>> my\_str

'This is a simple string testing'

That didn’t change the string; it reassigned the variable. The original string of "This is a simple string" was unchanged.

An important thing to realize is that the contents of a variable can be changed and we don’t need to maintain the same type:

>>> my\_str = 1

>>> print(my\_str)

1

Ideally, we wouldn’t change the contents of a variable called my\_str to be an int, but it is something that python would let use do.

One last thing to remember is that if we assign a variable with another variable, it will be assigned to the result of the variable and not whatever that variable points to later.

>>> my\_str = 1

>>> my\_int = my\_str

>>> my\_str = "testing"

>>> print(my\_int)

1

>>> print(my\_str)

testing

n Python, there are a few different “sequence” types that we’re going to work with, the most common of which is the list type.

#### Python Documentation For This Video

* [Sequence Types](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range)
* [Lists](https://docs.python.org/3/library/stdtypes.html#list)

#### Lists

A list is created in Python by using the square brackets ([, and ]) and separating the values by commas. Here’s an example list:

>>> my\_list = [1, 2, 3, 4, 5]

There’s really not a limit to how long our list can be (there is, but it’s very unlikely that we’ll hit it while scripting).

#### Reading from Lists

To access an individual element of a list, you can use the index and Python uses a zero-based index system:

>>> my\_list[0]

1

>>> my\_list[1]

2

If we try to access an index that is too high (or too low) then we’ll receive an error:

>>> my\_list[5]

Traceback (most recent call last):

File "", line 1, in

IndexError: list index out of range

To make sure that we’re not trying to get an index that is out of range, we can test the length using the len function (and then subtract 1):

>>> len(my\_list)

5

Additionally, we can access subsections of a list by “slicing” it. We provide the starting index and the ending index (the object at that index won’t be included).

>>> my\_list[0:2]

[1, 2]

>>> my\_list[1:]

[2, 3, 4, 5]

>>> my\_list[:3]

[1, 2, 3]

>>> my\_list[0::1]

[1, 2, 3, 4, 5]

>>> my\_list[0::2]

[1, 3, 5]

#### Modifying a List

Unlike strings which can’t be modified (you can’t change a character in a string), you can change a value in a list using the subscript equals operation:

>>> my\_list[0] = "a"

>>> my\_list

['a', 2, 3, 4, 5]

If we want to add to a list we can use the .append method. This is an example of a method that modifies the object that is calling the method:

>>> my\_list.append(6)

>>> my\_list.append(7)

>>> my\_list

['a', 2, 3, 4, 5, 6, 7]

Lists can be added together (concatenated):

>>> my\_list + [8, 9, 10]

['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> my\_list += [8, 9, 10]

>>> my\_list

['a', 2, 3, 4, 5, 6, 7, 8, 9, 10]

Items in lists can be set using slices also:

>>> my\_list[1:3] = ['b', 'c']

>>> my\_list

['a', 'b', 'c', 4, 5, 6, 7, 8, 9, 10]

# Replacing 2 sized slice with length 3 list inserts new element

my\_list[3:5] = ['d', 'e', 'f']

print(my\_list)

We can remove a section of a list by assigning an empty list to the slice:

>>> my\_list = ['a', 'b', 'c', 'd', 5, 6, 7]

>>> my\_list[4:] = []

>>> my\_list

['a', 'b', 'c', 'd']

Removing items from a list based on value can be done using the .remove method:

>>> my\_list.remove('b')

>>> my\_list

['a', 'c', 'd']

Attempting to remove and item that isn’t in the list will result in an error:

>>> my\_list.remove('f')

Traceback (most recent call last):

File "", line 1, in

ValueError: list.remove(x): x not in list

Items can also be removed from the end of a list using the pop method:

>>> my\_list = ['a', 'c', 'd']

>>> my\_list.pop()

'd'

>>> my\_list

['a', 'c']

We can also use the pop method to remove items at a specific index:

>>> my\_list.pop(0)

'a'

>>> my\_list

['c']

>>> my\_list.pop(1)

Traceback (most recent call last):

File "", line 1, in

IndexError: pop index out of range

>>> [].pop()

Traceback (most recent call last):

File "", line 1, in

IndexError: pop from empty list

#### Python Documentation For This Video

* [Sequence Types](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range)
* [Tuples](https://docs.python.org/3/library/stdtypes.html#tuple)

## Tuples

Tuples are a fixed width, immutable sequence type. We create tuples using parenthesis (( and )) and at least one comma (,):

>>> point = (2.0, 3.0)

Since tuples are immutable, we don’t have access to the same methods that we do on a list. We can use tuples in some operations like concatenation, but we can’t change the original tuple that we created.

>>> point\_3d = point + (4.0,)

>>> point\_3d

(2.0, 3.0, 4.0)

One interesting characterist of tuples is that we can unpack them into multiple variables at the same time:

>>> x, y, z = point\_3d

>>> x

2.0

>>> y

3.0

>>> z

4.0

The time you’re most likely to see tuples will be when looking at a format string that’s compatible with Python 2:

>>> print("My name is: %s %s" % ("Keith", "Thompson"))

Learn how to use dictionaries (the dict type) to hold onto key/value information in Python.

#### Python Documentation For This Video

* [Dictionaries](https://docs.python.org/3/library/stdtypes.html#mapping-types-dict)

#### Dictionaries

Dictionaries are the main mapping type that we’ll use in Python. This object is comparable to a Hash or “associative array” in other languages.

Things to note about dictionaries:

1. Unlike Python 2 dictionaries, as of Python 3.6, keys are ordered in dictionaries. You'll need OrderedDict if you want this to work on another version of Python.
2. You can set the key to any IMMUTABLE TYPE (no lists).
3. Avoid using things other than simple objects as keys.
4. Each key can only have one value (so don’t have duplicates when creating a dict).

We create dictionary literals by using curly braces ({ and }), separating keys from values using colons (:), and separating key/value pairs using commas (,). Here’s an example dictionary:

>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }

>>> ages

{'kevin': 59, 'alex': 29, 'bob': 40}

We can read a value from a dictionary by subscripting using the key:

>>> ages['kevin']

59

>>> ages['billy']

Traceback (most recent call last):

File "", line 1, in

KeyError: 'billy'

Keys can be added or changed using subscripting and assignment:

>>> ages['kayla'] = 21

>>> ages

{'kevin': 59, 'alex': 29, 'bob': 40, 'kayla': 21}

Items can be removed from a dictionary using the del statement or by using the pop method:

>>> del ages['kevin']

>>> ages

{'alex': 29, 'bob': 40, 'kayla': 21}

>>> del ages

>>> ages

Traceback (most recent call last):

File "", line 1, in

NameError: name 'ages' is not defined

>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }

>>> ages.pop('alex')

29

>>> ages

{'kevin': 59, 'bob': 40}

It’s not uncommon to want to know what keys or values we have without caring about the pairings. For that situation we have the values and keys methods:

>>> ages = {'kevin': 59, 'bob': 40}

>>> ages.keys()

dict\_keys(['kevin', 'bob'])

>>> list(ages.keys())

['kevin', 'bob']

>>> ages.values()

dict\_values([59, 40])

>>> list(ages.values())

[59, 40]

#### Alternative Ways to Create a dict Using Keyword Arguments

There are a few other ways to create dictionaries that we might see, those being those that use the dict constructor with key/value arguments and a list of tuples:

>>> weights = dict(kevin=160, bob=240, kayla=135)

>>> weights

{'kevin': 160, 'bob': 240, 'kayla': 135}

>>> colors = dict([('kevin', 'blue'), ('bob', 'green'), ('kayla', 'red')])

>>> colors

{'kevin': 'blue', 'bob': 'green', 'kayla': 'red'}

Scripts become most interesting when they do the right thing based on the inputs that we provide. To start building robust scripts, we need to understand how to make comparisons and use conditionals.

#### Python Documentation For This Video

* [Comparisons](https://docs.python.org/3/library/stdtypes.html#comparisons)
* [if/elif/else](https://docs.python.org/3/tutorial/controlflow.html#if-statements)

#### Comparisons

There are some standard comparison operators that we’ll use that match pretty closely to those used in mathematical equations. Let’s take a look at them:

>>> 1 < 2

True

>>> 0 > 2

False

>>> 2 == 1

False

>>> 2 != 1

True

>>> 3.0 >= 3.0

True

>>> 3.1 <= 3.0

False

If we try to make comparisons of types that don’t match up, we will run into errors:

>>> 3.1 <= "this"

Traceback (most recent call last):

File "", line 1, in

TypeError: '<=' not supported between instances of 'float' and 'str'

>>> 3 <= 3.1

True

>>> 1.1 == "1.1"

False

>>> 1.1 == float("1.1")

True

We can compare more than just numbers. Here’s what it looks like when we compare strings:

>>> "this" == "this"

True

>>> "this" == "This"

False

>>> "b" > "a"

True

>>> "abc" < "b"

True

Notice that the string 'b' is considered greater than the strings 'a' and 'abc'. The characters are compared one at a time alphabetically to determine which is greater. This concept is used to sort strings alphabetically.

#### The in Check

We often get lists of information that we need to ensure contains (or doesn’t contain) a specific item. To make this check in Python, we’ll use the in and not in operations.

>>> 2 in [1, 2, 3]

True

>>> 4 in [1, 2, 3]

False

>>> 2 not in [1, 2, 3]

False

>>> 4 not in [1, 2, 3]

True

#### if/elif/else

With a grasp on comparisons, we can now look at how we can run different pieces of logic based on the values that we’re working with using conditionals. The keywords for conditionals in Python are if, elif, and else. Conditionals are the first language feature that we’re using that requires us to utilize whitespace to separate our code blocks. We will always use indentation of 4 spaces. The basic shape of an if statement is this:

if CONDITION:

pass

The CONDITION portion can be anything that evaluates to True or False, and if the value isn’t explicitly a boolean, then it will be converted to determine how to carry out proceed past the conditional (basically using the bool constructor).

>>> if True:

... print("Was True")

...

Was True

>>> if False:

... print("Was True")

...

>>>

To add an alternative code path, we’ll use the else keyword, followed by a colon (:), and indenting the code underneath:

>>> if False:

... print("Was True")

... else:

... print("Was False")

...

Was False

In the even that we want to check multiple potential conditions we can use the elif CONDITION: statement. Here’s a more robust example:

>>> name = "Kevin"

>>> if len(name) >= 6:

... print("name is long")

... elif len(name) == 5:

... print("name is 5 characters")

... elif len(name) >= 4:

... print("name is 4 or more")

... else:

... print("name is short")

...

name is 5 characters

Notice that we fell into the first elif statement’s block and then the second elif block was never executed even though it was true. We can only exercise one branch in an if statement.

It’s incredibly common to need to repeat something a set number of times or to iterate over content. Here is where looping and iteration come into play.

#### Python Documentation For This Video

* [while statement](https://docs.python.org/3/tutorial/introduction.html#first-steps-towards-programming)
* [for statement](https://docs.python.org/3/tutorial/controlflow.html#for-statements%3CPaste%3E)

#### NOTE - encouraged to re-play these videos and revew explanations:

#### The while Loop

The most basic type of loop that we have at our disposal is the while loop. This type of loop repeats itself based on a condition that we pass to it. Here’s the general structure of a while loop:

while CONDITION:

pass

The CONDITION in this statement works the same way that it does for an if statement. When we demonstrated the if statement, we first tried it by simply passing in True as the condition. Let’s see when we try that same condition with a while loop:

>>> while True:

... print("looping")

...

looping

looping

looping

looping

That loop will continue forever, we’ve created an infinite loop. To stop the loop, press Ctrl-C. Infinite loops are one of the potential problems with while loops if we don’t use a condition that we can change from within the loop then it will continue forever if initially true. Here’s how we’ll normally approach using a while loop where we modify something about the condition on each iteration:

>>> count = 1

>>> while count <= 4:

... print("looping")

... count += 1

...

looping

looping

looping

looping

>>>

We can use other loops or conditions inside of our loops; we need only remember to indent four more spaces for each context. If in a nested context, we want to continue to the next iteration or stop the loop entirely. We also have access to the continue and break keywords:

>>> count = 0

>>> while count < 10:

... if count % 2 == 0:

... count += 1

... continue

... print(f"We're counting odd numbers: {count}")

... count += 1

...

We're counting odd numbers: 1

We're counting odd numbers: 3

We're counting odd numbers: 5

We're counting odd numbers: 7

We're counting odd numbers: 9

>>>

In that example, we also show off how to “string interpolation” in Python 3 by prefixing a string literal with an f and then using curly braces to substitute in variables or expressions (in this case the count value).

Here’s an example using the break statement:

>>> count = 1

>>> while count < 10:

... if count % 2 == 0:

... break

... print(f"We're counting odd numbers: {count}")

... count += 1

...

We're counting odd numbers: 1

It’s incredibly common to need to repeat something a set number of times or to iterate over content. Here is where looping and iteration come into play.

#### Python Documentation For This Video

* [for statement](https://docs.python.org/3/tutorial/controlflow.html#for-statements%3CPaste%3E)

#### The for Loop

The most common use we have for looping is when we want to execute some code for each item in a sequence. For this type of looping or iteration, we’ll use the for loop. The general structure for a for loop is:

for TEMP\_VAR in SEQUENCE:

pass

The TEMP\_VAR will be populated with each item as we iterate through the SEQUENCE and it will be available to us in the context of the loop. After the loop finishes one iteration, then the TEMP\_VAR will be populated with the next item in the SEQUENCE, and the loop’s body will execute again. This process continues until we either hit a break statement or we’ve iterated over every item in the SEQUENCE. Here’s an example looping over a list of colors:

>>> colors = ['blue', 'green', 'red', 'purple']

>>> for color in colors:

... print(color)

...

blue

green

red

purple

>>> color

'purple'

If we didn't want to print out certain colors we could utilize the continue or break statements again. Let’s say we want to skip the string 'blue' and terminate the loop if we see the string 'red':

>>> colors = ['blue', 'green', 'red', 'purple']

>>> for color in colors:

... if color == 'blue':

... continue

... elif color == 'red':

... break

... print(color)

...

green

>>>

#### Other Iterable Types

Lists will be the most common type that we iterate over using a for loop, but we can also iterate over other sequence types. Of the types we already know, we can iterate over strings, dictionaries, and tuples.

Here’s a tuple example:

>>> point = (2.1, 3.2, 7.6)

>>> for value in point:

... print(value)

...

2.1

3.2

7.6

>>>

A dictionary example:

>>> ages = {'kevin': 59, 'bob': 40, 'kayla': 21}

>>> for key in ages:

... print(key)

...

kevin

bob

kayla

A string example:

>>> for letter in "my\_string":

... print(letter)

...

m

y

\_

s

t

r

i

n

g

>>>

#### Unpacking Multiple Items in a for Loop

We discussed in the tuples video how you can separate a tuple into multiple variables by “unpacking” the values. Unpacking works in the context of a loop definition, and you’ll need to know this to most effectively iterate over dictionaries because you’ll usually want the key and the value. Let’s iterate of a list of “points” to test this out:

>>> list\_of\_points = [(1, 2), (2, 3), (3, 4)]

>>> for x, y in list\_of\_points:

... print(f"x: {x}, y: {y}")

...

x: 1, y: 2

x: 2, y: 3

x: 3, y: 4

Seeing how this unpacking works, let’s use the items method on our ages dictionary to list out the names and ages:

>>> for name, age in ages.items():

... print(f"Person Named: {name}")

... print(f"Age of: {age}")

...

Person Named: kevin

Age of: 59

Person Named: bob

Age of: 40

Person Named: kayla

Age of: 21

Up to this point, we’ve learned how to make simple comparisons, and now it’s time to make compound comparisons using logic/boolean operators.

#### Python Documentation For This Video

* [Boolean Operators](https://docs.python.org/3/tutorial/introduction.html#first-steps-towards-programming)

#### The not Operation

Sometimes we want to know the opposite boolean value for something. This might not sound intuitive, but sometimes we want to execute an if statement when a value is False, but that’s not how the if statement works. Here’s an example of how we can use not to make this work:

>>> name = ""

>>> not name

True

>>> if not name:

... print("No name given")

...

>>>

We know that an empty string is a “falsy” value, so not "" will always return True. not will return the opposite boolean value for whatever it’s operating on.

#### The or Operation

Occasionally, we want to carry out a branch in our logic if one condition OR the other condition is True. Here is where we’ll use the or operation. Let’s see or in action with an if statement:

>>> first = ""

>>> last = "Thompson"

>>> if first or last:

... print("The user has a first or last name")

...

The user has a first or last name

>>>

If both first and last were “falsy” then the print would never happen:

>>> first = ""

>>> last = ""

>>> if first or last:

... print("The user has a first or last name")

...

>>>

Another feature of or that we should know is that we can use it to set default values for variables:

>>> last = ""

>>> last\_name = last or "Doe"

>>> last\_name

'Doe'

>>>

The or operation will return the first value that is “truthy” or the last value in the chain:

>>> 0 or 1

1

>>> 1 or 2

1

#### The and Operation

The opposite of or is the and operation, which requires both conditions to be True. Continuing with our first and last name example, let’s conditionally print based on what we know:

>>> first = "Keith"

>>> last = ""

>>> if first and last:

... print(f"Full name: {first} {last}")

... elif first:

... print(f"First name: {first}")

... elif last:

... print(f"Last name: {last}")

...

First name: Keith

>>>

Now let’s try the same thing with both first and last:

>>> first = "Keith"

>>> last = "Thompson"

>>> if first and last:

... print(f"Full name: {first} {last}")

... elif first:

... print(f"First name: {first}")

... elif last:

... print(f"Last name: {last}")

...

Full name: Keith Thompson

>>>

The and operation will return the first value that is “falsy” or the last value in the chain:

>>> 0 and 1

0

>>> 1 and 2

2

>>> (1 == 1) and print("Something")

Something

>>> (1 == 2) and print("Something")

False