CHAPTER 2.1
Installing Python 3.7 on CentOS 7

Learn how to install Python 3 from source on a CentOS 7 machine.

Note: This course uses Python 3.7 and you will definitely run into issues if you are using Python < 3.7.

Download and Install Python 3 from Source

Here are the commands that we'll run to build and install Python 3.7:

```
$ sudo -i
$ yum groupinstall -y "development tools"
$ yum install -y \
  libffi-devel \
  zlib-devel \
  bzip2-devel \
  openssl-devel \
  ncurses-devel \
  sqlite-devel \
  readline-devel \
  tk-devel \
  gdbm-devel \
  db4-devel \
  libpcap-devel \
  xz-devel \
  expat-devel \
  postgresql-devel

$ cd /usr/src
$ wget http://python.org/ftp/python/3.7.2/Python-3.7.2.tar.xz
$ tar xf Python-3.7.2.tar.xz
$ cd Python-3.7.2
$ ./configure --enable-optimizations
$ make altinstall
$ exit
```

Important: make altinstall causes it to not replace the built in python executable.

Using sudo nano /etc/sudoers (or your preferred text editor), ensure that secure_path in /etc/sudoers file includes /usr/local/bin. The line should look something like this:

```
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin
```

Upgrade Pip (might not be necesary)

The version of pip that we have might be up-to-date, but it's a good practice to try to update it after the installation. We need to use the pip3.7 executable because we're working with Python 3, and we use sudo so that we can write files under the /usr/local directory.

```
$ sudo pip3.7 install --upgrade pip
```

CHAPTER 2.2
(Optional) Installing Python 3.7 on Debian/Ubuntu

Learn how to install Python 3 from source on a Debian or Ubuntu machine. This video uses an Ubuntu 18.04 Linux Academy Cloud Server.

Note: This course uses Python 3.7 and you will definitely run into issues if you are using Python < 3.7.

Download and Install Python 3 from Source
https://www.python.org/downloads/

Here are the commands that we'll run to build and install Python 3:

```
$ sudo -i
$ apt update -y
$ apt install -y \
  wget \
  build-essential \
  libffi-dev \
  libgdbm-dev \
  libc6-dev \
  libssl-dev \
  zlib1g-dev \
  libbz2-dev \
  libreadline-dev \
  libsqlite3-dev \
  libncurses5-dev \
  libncursesw5-dev \
  xz-utils \
  tk-dev

$ cd /usr/src
$ wget http://python.org/ftp/python/3.7.2/Python-3.7.2.tar.xz
$ tar xf Python-3.7.2.tar.xz
$ cd Python-3.7.2
$ ./configure --enable-optimizations
$ make altinstall
```

Note: make altinstall causes it to not replace the built in python executable.

Ensure that secure_path in /etc/sudoers file includes /usr/local/bin. The line should look something like this:

```
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/
```

sbin:/usr/bin:/sbin:/bin:/snap/bin
Upgrade Pip (might not be necesary)
The version of pip that we have might be up-to-date, but it's a good practice to try to update it after the installation. We need to use the pip3.7 executable because we're working with Python 3, and we use sudo so that we can write files under the /usr/local directory.

```
$ pip3.7 install --upgrade pip
```

Introduction to Python Development

CHAPTER 2.3
Picking a Text Editor or IDE

Before we start writing code, we should think about the tools that we're using to do the development. Having a well configured text editor can make the programming experience much more enjoyable. Much like a carpenter, having sharp tools leads to a more productive creative experience.

Documentation For This Video
Vim (https://www.vim.org/)
Emacs (https://www.gnu.org/software/emacs/)
Nano (https://www.nano-editor.org/)
Atom (https://atom.io/)
VS Code (https://code.visualstudio.com/)
SublimeText (https://www.sublimetext.com/)
Notepad++ (https://notepad-plus-plus.org/)
PyCharm (https://www.jetbrains.com/pycharm/)

Terminal Based Editors
There are a few different terminal editors that you can work with. The main reason to use a terminal based editor is that you can run them on servers that you're connected to and you can stay in a terminal to carry out any programming task, whether that be developing the code, debugging, or deploying. There are two terminal based editors I would consider to be extremely popular:

Vim - Modal editor, extremely customizable. (https://www.vim.org/)
Emacs - Unbelievably customizable, not modal (at least not by default). (https://www.gnu.org/software/emacs/)
Both of these tools are either pre-installed or readily available on all major Linux distros.

The third option is Nano/Pico (https://www.nano-editor.org/) and it's more of a tool that I would suggest using if nothing else is available.

For this course, I'll be using a lightly customized version of Vim.

GUI Based Editors
GUI based editors can be extremely powerful and more aestetically pleasing than terminal based editors. This list is comprised of classic "text editors", but most of them can be enhanced using plugins that add additional functionality. I'm going to divide them into two camps: native applications and Electron applications (built using JavaScript). This seems like a weird distinction, but plenty of people

don't like the resource overhead that running Electron based applications requires.

Native:

SublimeText - Multi-platform. Very performant and extended using Python 3. (https://www.sublimetext.com/)
Notepad++ - Windows only. Not as powerful as the others, but a good starter text editor that won't get in your way. (https://notepad-plus-plus.org/)
Electron Based:

Atom - The original Electron based editor. Aestetically pleasing and very extendable through plugins. (https://atom.io/)
VS Code - Arguably the most popular GUI based editor. Vast ecosystem of plugins and built-in debugger. (https://code.visualstudio.com/)

IDEs
The primary IDE that is used by people in the Python community would be PyCharm (https://www.jetbrains.com/pycharm/). There is a free community edition and there is also a paid edition. To connect to a remote server to do your editing, you'll need to have the paid version.

Introduction to Python Development

CHAPTER 2.4
(Optional) Setting Up a Vim Development Environment

To get started with the course you're going to need a few things installed:

Python 3
Git
A text editor of your choice
My preferred text editor is Vim and that's what I'll be using throughout the course.

For a great vim cheatsheet, see:
https://linuxacademy.com/site-content/uploads/2019/05/vim-1.png

This course is focused on using Python on a Linux system. You can definitely follow
along on a Mac or Windows machine, but we won't be covering potential differences with those systems
or how to install Python on Windows.

I will be using CentOS 7 and if you'd like to follow along exactly we'll go through what you need to
be able to use one of your Linux Academy Cloud Playground servers.

Installing Development tools and Vim
We've already installed most of what we need, but attempting to install development tools again won't hurt. Vim has yet to be installed, so we'll add that now:

```
$ sudo yum update -y
$ sudo yum groupinstall -y "development tools"
$ sudo yum install -y vim-enhanced
```

Configure git:
This is just a best practice, but we'll want to configure Git to have our name and email address so that we can sign any commits that we make as we're developing.

```
$ git config --global user.name "Keith Thompson"
$ git config --global user.email "keith@linuxacademy.com"
```

Pull down sample bashrc
Having a comfortable development environment makes life easier. For this course, I'm going to use a slightly customized bash configuration to improve my experience in the terminal. You can chose to skip this step if you'd like. We can download this file from this course's Github repository:

```
$ curl https://raw.githubusercontent.com/linuxacademy/content-intro-to-python-development/master/helpers/bashrc -o ~/.bashrc
$ exec $SHELL
```

The exec $SHELL reloaded our shell session and we can now see the customizations to the prompt. We'll come back to this in a second to see some enhancements that only show up when working within git repositories.

Pull down sample vimrc

Vim is a great text editor, but it doesn't have the best default settings. I've created a simple, yet usable vim configuration to use with this course that will improve our experience while writing Python code. Let's fetch this file now:

```
$ curl https://raw.githubusercontent.com/linuxacademy/content-intro-to-python-development/master/helpers/vimrc -o ~/.vimrc
```

Demonstrat PS1 changes

One of the big changes in the bash configuration is that it adds support for showing information about the git repository that we're currently working in. Let's create a sample project so we can see what it shows:

```
$ mkdir sample
$ cd sample
$ touch file.txt
$ git init
$ git add --all .
$ git commit -m 'Initial commit'
```

Introduction to Python Development

CHAPTER 3.1
Using the REPL (Read, Evaluate, Print, Loop)

Python is an interpreted language, and 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.

Documentation
Python Interpreter (https://docs.python.org/3/tutorial/interpreter.html)

What is a REPL?
REPL stands for: Read, Evaluate, Print, Loop

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

Python ships with a REPL, accessible by running python3.7 from a terminal.

```
$ python3.7
Python 3.7.2 (default, Jan 15 2019, 19:31:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> indicates a line to type in, like a command prompt for Python. Later on we'll see a ..., which means that we are currently sitting in a scoped area. The only way out is to enter a blank line (no spaces) before the interpreter will evaluate 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 we'll just land back at a >>> prompt. We'll cover this later, but an example would be None:

```
>>> None
>>>
```

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

Introduction to Python Development

CHAPTER 3.2
Creating and Running Python Files

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 pays that we can run this file. The first is by passing it to the python3.7 CLI.

```
$ python3.7 hello.py
Hello, World!
```

Setting a Shebang

You'll most likely want your scripts to be:

Executable from anywhere (in our $PATH)
Executable without explicitly using the python3.7 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 python3.7
print("Hello, World")
```

We're not quite done, because 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. Renaming it to get rid of the extension (mv hello.py hello) and running ./hello will still result in Python 3.7 executing the script.

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.py ~/bin/hello
```

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

```
$ export PATH=$PATH:$HOME/bin
```

Finally, let's run the hello script from our $PATH:

```
$ hello
Hello, World!
```

CHAPTER 3.3
Using Comments

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.

Single Line Comment
We can comment out a whole line:

```
# This is a full line 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.
"""
```

Introduction to Python Development

CHAPTER 4.1
Strings

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

Note:

\ - backslash

/ - forward slash

Documentation
[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:

$ python3.7
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 do 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
```

Introduction to Python Development

CHAPTER 4.2
Numbers

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 to main types of numbers we'll use in Python: int and float. For the most part, we won't be calling methods on number types — instead, we will use 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
It's not uncommon for us to need to convert from one type to another when writing a script. 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 "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1.1 things'
```

Introduction to Python Development

CHAPTER 4.3
Booleans and None

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.

CHAPTER 4.4
Working with Variables

Almost any script we write will need to have a way for us to hold on to 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 us 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 varible points to later.

```
>>> my_str = 1
>>> my_int = my_str
>>> my_str = "testing"
>>> print(my_int)
1
>>> print(my_str)
testing
```

Introduction to Python Development

CHAPTER 4.5
Lists

In Python, there are a few different "sequence" types that we're going to work with, the most common of which being 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[2]
3
```

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 "<stdin>", line 1, in <module>
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]
```
Items in lists can be set using slices also:

```
>>> my_list[0:2] = 'a'
>>> my_list
['a', 3, 4, 5]
>>> my_list[0:2] = [1, 2, 3]
>>> my_list
[1, 2, 3, 4, 5]
>>> my_list[0:2] = []
>>> my_list
[3, 4, 5]
```
Attempting to remove an item that isn't in the list will result in an error:

```
>>> my_list.remove(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```
If the item does exist in the list then it will be removed:

```
>>> my_list.remove(4)
>>> my_list
[3, 5]
```
Items can also be removed from the end of a list using the pop method:

```
>>> my_list.pop()
5
>>> my_list
[3]
>>> my_list.pop()
3
>>> my_list
[]
>>> my_list.pop()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```
Lists can be used to create a stack and pop can be used to take the first item like this:

```
>>> my_list = [1, 2, 3, 4]
>>> my_list.pop(0)
1
>>> my_list
[2, 3, 4]
```
Adding to the list can be done in a few ways. The first of which is by using the append method:

```
>>> my_list.append(5)
>>> my_list
[2, 3, 4, 5]
>>> my_list.insert(1, 3)
>>> my_list
[2, 3, 3, 4, 5]
>>> my_list.insert(0, 1)
>>> my_list
[1, 2, 3, 3, 4, 5]
```

Introduction to Python Development

CHAPTER 4.6
Tuples & Ranges

The most common immutable sequence type that we're going to work with is going to be the tuple. We'll also look at the range type as an alternative to a sequential 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)
Ranges (https://docs.python.org/3/library/stdtypes.html#range)

Tuples
Tuples are a fixed-width, immutable sequence type. We create tuples using parenthesis ((,)) 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 characteristic 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 we'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"))
```

Ranges
Ranges are an immutable sequence type that defines a start, a stop, and a step value, and then the values within are calculated as it is iteracted with. This allows for ranges to be used in place of sequential lists and while taking less memory and including more items.

```
>>> my_range = range(10)
>>> my_range
range(0, 10)
>>> list(my_range)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 14, 2))
[1, 3, 5, 7, 9, 11, 13]
```

Notice that the "stop" value is not included in the range, the values are all up-until the stop.

Introduction to Python Development

CHAPTER 4.7
Dictionaries (dicts)

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:

Unlike Python 2 dictionaries, as of Python 3.6 keys are ordered in dictionaries. Will need OrderedDict if you want this to work on another version of Python.
You can set the key to any IMMUTABLE type (no lists)
Avoid using things other than simple objects as keys.
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 "<stdin>", line 1, in <module>
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 "<stdin>", line 1, in <module>
NameError: name 'ages' is not defined
>>> ages = { 'kevin': 59, 'alex': 29, 'bob': 40 }
>>> ages.pop('alex')
29
>>> ages
{'kevin': 59, 'bob': 40}
>>> {}.pop('billy')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'billy'
```

Since this is our second time running into a KeyError it's worth looking at a way to avoid these when we're attempting to read data from a dictionary. For that, we can use the get method:

```
>>> ages.get('kevin')
59
>>> ages.get('billy')
>>>
```

Now we're receiving None instead of raising an error when we try to get the value for a key that doesn't exist.

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 using 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'}
```

Introduction to Python Development

CHAPTER 5.1
Conditionals and Comparisons

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 "<stdin>", line 1, in <module>
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.

Introduction to Python Development

CHAPTER 5.2
Logic Operations

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/library/stdtypes.html#boolean-operations-and-or-not)

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
```

Introduction to Python Development

CHAPTER 5.3
The `while` loop

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)

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
```

Introduction to Python Development

CHAPTER 5.4
The `for` Loop

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)

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 wanted not 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 could 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
```

Introduction to Python Development

CHAPTER 6.1
Writing Functions

Being able to write code that we can call multiple times without repeating ourselves is one of the most powerful things that we can do when programming. Let's learn how to define functions in Python.

Python Documentation For This Video
Defining Functions (https://docs.python.org/3/tutorial/controlflow.html#defining-functions)

Function Basics
We can create functions in Python using the following:

The def keyword
The function name - lowercase starting with a letter or underscore (_)
Left parenthesis (()
0 or more argument names
Right parenthesis ())
A colon :
An indented function body
Here's an example without an argument:

```
>>> def hello_world():
...     print("Hello, World!")
...
>>> hello_world()
Hello, World!
>>>
```

If we want to define an argument, we will put the variable name we want it to have within the parentheses:

```
>>> def print_name(name):
...     print(f"Name is {name}")
...
>>> print_name("Keith")
Name is Keith
```

Let's try to assign the value from print_name to a variable:

```
>>> output = print_name("Keith")
Name is Keith
>>> output
>>>
```

Neither of these examples has a return value, but we will usually want to have a return value unless the function is our "main" function, or carries out a "side-effect" like printing. If we don't explicitly declare a return value, then the result will be None.

We can declare what we're returning from a function using the return keyword:

```
>>> def add_two(num):
...     return num + 2
...
>>> result = add_two(2)
>>> result
4
```

Working with Multiple Arguments

When we have a function that takes multiple arguments, we need to separate them using commas and give them unique names:

```
>>> def add(num1, num2):
...     return num1 + num2
...
>>> result = add(1, 5)
>>> result
6
```

Using Keyword Arguments

Every function call we've made up to this point has used what are known as positional arguments, but if we know the name of the arguments and not necessarily the positions we can call them all using keyword arguments like so:

```
>>> def contact_card(name, age, car_model):
...     return f"{name} is {age} and drives a {car_model}"
...
>>> contact_card("Keith", 29, "Honda Civic")
'Keith is 29 and drives a Honda Civic'
>>> contact_card(age=29, car_model="Civic", name="Keith")
'Keith is 29 and drives a Civic'
>>> contact_card("Keith", car_model="Civic", age="29")
'Keith is 29 and drives a Civic'
>>> contact_card(age="29", "Keith", car_model="Civic")
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

When we're using position and keyword arguments, every argument after the first keyword argument must also be a keyword argument. It's sometimes useful to mix them, but often times we'll use either all positional or all keyword.

Defining Default Arguments

Along with being able to use keyword arguments when we're calling a function, we're able to define default values for arguments to make them optional when the information is commonly known and the same. To do this, we use the assignment operator (=) when we're defining the argument:

```
>>> def can_drive(age, driving_age=16):
...     return age >= driving_age
...
>>> can_drive(16)
True
>>> can_drive(16, driving_age=18)
False
```
Default arguments need to go at the end of the arguments list when defining the function, so that positional arguments can still be used to call the function.

Introduction to Python Development

CHAPTER 7.1
Creating Classes

The next step in our programming journey requires us to think about how we can model concepts from our problem's domain. To do that, we'll often use classes to create completely new data types. In this lesson, we'll create our very first class and learn how to work with its data and functionality.

Python Documentation For This Video
Classes (https://docs.python.org/3/tutorial/classes.html#classes)

Defining New Types
Up to this point, we've been working with the built-in types that Python provides (str, int, float, etc.), but when we're modeling problems in our programs we often want to have more complex objects that fit our problem's domain. For instance, if we were writing a program to model information about cars (for an automotive shop) then it would make sense for us to have an object type that represents a car. This is where we start working will classes.

From this point on, most of the code that we'll be writing will be in files. Let's create a learning_python directory to hold these files that are really only there to facilitate learning.

```
$ mkdir ~/learning_python
$ cd ~/learning_python
```

Creating Our First Class
For this lesson, we'll use a file called creating_classes.py. Our goal is to model a car that has tires and an engine. To create a class we use the class keyword, followed by a name for the class, starting with a capital letter. Let's create our first class, the Car class:

~/learning_python/creating_classes.py:

```
class Car:
    """
    Docstring describing the class
    """

    def __init__(self):
        """
        Docstring describing the method
        """
        pass
```

This is an incredibly simple class. A few things to note here are that by adding a triple-quoted string right under the definition of the class and also right under the definition of a method/function we can add documentation. This documentation is nice because we can even add examples in this string that can be run as tests to help ensure that our documentation stays up to date with the implementation.

A method is a function that is defined within the context of an object, and Python classes can define special functions that start with double underscores __, such as the __init__ method. This method overrides the initializer of the class. The initializer is what is used when we create a new version of our class by running code like this:

```
>>> my_car = Car()
```

We would like our Car class to hold onto a few pieces of data, the tires, and an engine. For the time being, we're just going to have those be a list of strings for the tires and a string for the engine. Let's modify our __init__ method to receive engine and tires as arguments:

~/learning_python/creating_classes.py

```
class Car:
    """
    Car models a car w/ tires and an engine
    """

    def __init__(self, engine, tires):
        self.engine = engine
        self.tires = tires
```

What is self?

A big change from writing functions to writing methods is the presence of self. This variable references the individual instance of the class that we're working with. The Car class holds on to the information about cars in general in our program, where an instance of the Car class (self) could represent my Honda Civic specifically. Let's load our class into the REPL using python3.7 -i creating_classes.py, and then we'll be able to create a Honda Civic:

```
$ python3.7 -i creating_classes.py
>>> civic = Car('4-cylinder', ['front-driver', 'front-passenger',
'rear-driver', 'rear-passenger'])
>>> civic.tires
['front-driver', 'front-passenger', 'rear-driver', 'rear-passenger']
>>> civic.engine
'4-cylinder'
```

Once we have our instance, we're able to access our internal attributes by using a period (.).

Defining a Custom Method
The last thing that we'll do, to round out the first rendition of our first class, is to define a method that prints a description of the car to the screen:

~/learning_python/creating_classes.py

```
class Car:
    """
    Car models a car w/ tires and an engine
    """

    def __init__(self, engine, tires):
        self.engine = engine
        self.tires = tires

    def description(self):
        print(f"A car with an {self.engine} engine, and {self.tires}
tires")
```

Our description method doesn't have any actual arguments, but we pass the instance in as self. From there, we can access the instance's attributes by calling self.ATTRIBUTE_NAME.

Let's use this new method:

```
$ python3.7 -i creating_classes.py
>>> honda = Car('4-cylinder', ['front-driver', 'front-passenger',
'rear-driver', 'rear-passenger'])
>>> honda.engine
'4-cylinder'
>>> honda.tires
['front-driver', 'front-passenger', 'rear-driver', 'rear-passenger']
>>> honda.description
<bound method Car.description of <__main__.Car object at
0x7fb5f3fbbda0>>
>>> honda.description()
A car with a 4-cylinder engine, and ['front-driver', 'front-
passenger', 'rear-driver', 'rear-passenger'] tires
```

Just like a normal function, if we don't use parenthesis the method won't execute.

Introduction to Python Development

CHAPTER 7.2
Composition

With one custom class under our belt, we're ready to think about how we can use classes together to create full-featured domain models. In this lesson, we'll create another class and utilize it with our Car class.

Python Documentation For This Video
Classes (https://docs.python.org/3/tutorial/classes.html#classes)
Doctest (https://docs.python.org/3.7/library/doctest.html)

Modeling the Tire

Currently, our Car class has tires and an engine, but they're all strings and don't really hold the information that we'd expect. For a tire, it should probably have these attributes:

brand - The brand of the tire.
tire_type - The type of the tire (valid options: None, 'P', 'LT'). We're not using type as the name because it's a name of the built-in function.
width - The tire width in millimeters
ratio - The ratio of the tire height to its width. A percentage represented as an integer.
construction - How the tire is constructed. The default (and only) option is 'R'.
diameter - The diameter of the wheel in inches.
Let's model our tire by creating a Tire class. We'll create this class in its own file (next to creating_classes.py) called tire.py:

~/learning_python/tire.py

```
class Tire:
    """
    Tire represents a tire that would be used with an automobile.
    """

    def __init__(self, tire_type, width, ratio, diameter, brand='',
construction='R'):
        self.tire_type = tire_type
        self.width = width
        self.ratio = ratio
        self.diameter = diameter
        self.brand = brand
        self.construction = construction
```

Now we have a way to represent an individual tire. Let's go into the

REPL and pass a list of Tire instances as tires when we create a Car:

```
$ python3.7 -i creating_classes.py
>>> from tire import Tire
>>> tire = Tire('P', 205, 55, 15)
>>> tires = [tire, tire, tire, tire]
>>> honda = Car(tires=tires, engine='4-cylinder')
>>> honda.description()
A car with a 4-cylinder engine, and [<tire.Tire object at
0x7ff1b0a7fe48>, <tire.Tire object at 0x7ff1b0a7fe48>, <tire.Tire
object at 0x7ff1b0a7fe48>, <tire.Tire object at 0x7ff1b0a7fe48>] tires
```

A few things to note here:

To load an additional file into the REPL, we were able to reference it by name using from [FILE_NAME_MINUS_EXTENSION] import [CLASS/FUNCTION/VARIABLE]. We'll learn more about loading code from other modules and packages when we look into the standard library, but this is handy for now.

We created a list of tires by using the same tire variable 4 times. The printing of each tire isn't very readable, and we can see that each item points to the same tire in memory (based on the at 0x7ff1b0a7fe48).

Before we discuss composition, let's improve this print out by adding a new double underscore method to the Tire class: the __repr__ method. The __repr__ method specifies what should be returned when an instance of the class is passed to the repr function, but also when it printed as part of another object being printed.

~/learning_python/tire.py

```
class Tire:
    """
    Tire represents a tire that would be used with an automobile.
    """

    def __init__(self, tire_type, width, ratio, diameter,
                 brand='', construction='R'):
        self.tire_type = tire_type
        self.width = width
        self.ratio = ratio
        self.diameter = diameter
        self.brand = brand
        self.construction = construction

    def __repr__(self):
        """
        Represent the tire's information in the standard notation
present
        on the side of the tire. Example: 'P215/65R15'
        """
```

```
        return (f"{self.tire_type}{self.width}/{self.ratio}"
                + f"{self.construction}{self.diameter}")
```

Now if we repeat the process of creating a car with some tires:

```
$ python3.7 -i creating_classes.py
>>> from tire import Tire
>>> tire = Tire('P', 205, 55, 15)
>>> tires = [tire, tire, tire, tire]
>>> honda = Car(tires=tires, engine='4-cylinder')
>>> honda.description()
A car with a 4-cylinder engine, and [P205/55R15, P205/55R15,
P205/55R15, P205/55R15] tires
```

Note: If we were just printing the tire by itself then it would use the __str__ method, and since we didn't implement that, it internally uses the __repr__ method.

What is Composition?

What we just did is use "composition" to build up our Car class by passing in Tire objects. One of the big ideas behind composition is that we can keep our classes focused on the behaviors and state that pertain to itself, and if it needs functionality from a different object we can inject those. The beautiful thing about composition is that it allows us to have a clean separation of concerns between our objects, and lets us reuse them. To show the power of composition, let's add a circumference method to our Tire class:

~/learning_python/tire.py

```
import math

class Tire:
    """
    Tire represents a tire that would be used with an automobile.
    """

    def __init__(self, tire_type, width, ratio, diameter, brand='',
construction='R'):
        self.tire_type = tire_type
        self.width = width
        self.ratio = ratio
        self.diameter = diameter
        self.brand = brand
        self.construction = construction

    def circumference(self):
        """
        The circumference of the tire in inches.

        >>> tire = Tire('P', 205, 65, 15)
        >>> tire.circumference()
```

```
        80.1
        """
        side_wall_inches = (self.width * (self.ratio / 100)) / 25.4
        total_diameter = side_wall_inches * 2 + self.diameter
        return round(total_diameter * math.pi, 1)

    def __repr__(self):
        """
        Represent the tire's information in the standard notation
present
        on the side of the tire. Example: 'P215/65R15'
        """
        return (f"{self.tire_type}{self.width}/{self.ratio}"
                + f"{self.construction}{self.diameter}")
```

Now we can use this method within our Car class by adding a wheel_circumference method:

~/learning_python/creating_classes.py

```
class Car:
    """
    Car models a car w/ tires and an engine
    """


    def __init__(self, engine, tires):
        self.engine = engine
        self.tires = tires

    def description(self):
        print(f"A car with a {self.engine} engine, and {self.tires}
tires")

    def wheel_circumference(self):
        if len(self.tires) > 0:
            return self.tires[0].circumference()
        else:
            return 0
```

This is the power of composition. Our Car class doesn't need to know how to calculate the circumference of its wheels (which makes sense, since you can swap out wheels on a car).

```
$ python3.7 -i creating_classes.py
>>> from tire import Tire
>>> tire = Tire('P', 205, 65, 15)
>>> tires = [tire, tire, tire, tire]
>>> honda = Car(tires=tires, engine='4-cylinder')
>>> honda.wheel_circumference()
80.1
>>> honda.tires = []
>>> honda.wheel_circumference()
```

A Quick Look at Doctests

You may have noticed the extra content that was added to the docstring of our circumference method. This is actually so that we can ensure our implementation works. We've simulated how we would use this code in the REPL, and we can use the doctest module to evaluate this, ensuring that the output would match the 80.1 we're expecting. Here's how we would run this:

```
$ python3.7 -m doctest -v tire.py
Trying:
    tire = Tire('P', 205, 65, 15)
Expecting nothing
ok
Trying:
    tire.circumference()
Expecting:
    80.1
ok
4 items had no tests:
    tire
    tire.Tire
    tire.Tire.__init__
    tire.Tire.__repr__
1 items passed all tests:
   2 tests in tire.Tire.circumference
2 tests in 5 items.
2 passed and 0 failed.
Test passed.
```

CHAPTER 7.3
Inheritance

Composition is a very powerful tool for code reuse, but one of the other tools that we have at our disposal is inheritance. Inheritance allows us to create new classes that add or modify the behavior of existing classes. In this lesson, we'll create a different type of Tire.

Documentation For This Video
Classes (https://docs.python.org/3/tutorial/classes.html#classes)
Inheritance (https://docs.python.org/3.7/tutorial/classes.html#inheritance)

Using Inheritance to Customize an Existing Class

Our existing Tire implementation does exactly what we need it to do for a general car tire, but there are other, more specific types of tires, such as racing slicks or snow tires. If we wanted to model these other types of tires, we could use our existing Tire class as a start by "inheriting" its existing implementation. Let's add a new SnowTire class to our tire.py file:

~/learning_python/tire.py

```
import math

class Tire:
    """
    Tire represents a tire that would be used with an automobile.
    """

    def __init__(self, tire_type, width, ratio, diameter, brand='',
construction='R'):
        self.tire_type = tire_type
        self.width = width
        self.ratio = ratio
        self.diameter = diameter
        self.brand = brand
        self.construction = construction

    def circumference(self):
        """
        The circumference of a tire in inches.

        >>> tire = Tire('P', 205, 65, 15)
        >>> tire.circumference()
```

```
        80.1
        """
        side_wall_inches = self._side_wall_inches()
        total_diameter = side_wall_inches * 2 + self.diameter
        return round(total_diameter * math.pi, 1)

    def __repr__(self):
        """
        Represent the tire's information in the standard notation
present
        on the side of the tire. Example: 'P215/65R15'
        """
        return (f"{self.tire_type}{self.width}/{self.ratio}"
                + f"{self.construction}{self.diameter}")

    def _side_wall_inches(self):
        return (self.width * (self.ratio / 100)) / 25.4

class SnowTire(Tire):
    def __init__(self, tire_type, width, ratio, diameter,
chain_thickness, brand='', construction='R'):
        Tire.__init__(self, tire_type, width, ratio, diameter, brand,
construction)
        self.chain_thickness = chain_thickness

    def circumference(self):
        """
        The circumference of a tire w/ show chains in inches.

        >>> tire = SnowTire('P', 205, 65, 15, 2)
        >>> tire.circumference()
        92.7
        """
        side_wall_inches = self._side_wall_inches()
        total_diameter = (side_wall_inches + self.chain_thickness) * 2
+ self.diameter
        return round(total_diameter * math.pi, 1)
```

We used another doctest here to show the usage of our SnowTire.circumference method. If we print a SnowTire instance it will automatically use the __repr__ implementation from the Tire class because we inherited all of the behavior of the Tire class. We customized both the __init__ and circumference methods to handle the changes that the chain_thickness value adds. Because the calculation of the tire sidewall thickness is a little complicated, we extracted that into a separate "private" method so that we could use it in both implementations (the method name starts with a single underscore).

Using super()

The circumference method is a situation where we needed to make a modification midway through the calculation, so it made more sense to

extract a helper method and write a whole new implementation. But most of the time when we're working with inheritance it's because we do want most of the initial implementation. In these situations, we have access to the super function that allows us to utilize the method implementations from our parent class. As it stands right now, our SnowTire class will display itself in the same way as the Tire class, but we'd like to distinguish them when they're printed out. To do this, we'll override the __repr__ method, but we want to simply add a (Snow) to the end of the original information. Let's utilize super to accomplish this:

~/learning_python/tire.py

```
# Implementation of Tire omitted

class SnowTire(Tire):
    def __init__(self, tire_type, width, ratio, diameter,
chain_thickness, brand='', construction='R'):
        Tire.__init__(self, tire_type, width, ratio, diameter, brand,
construction)
        self.chain_thickness = chain_thickness

    def circumference(self):
        """
        The circumference of a tire w/ show chains in inches.

        >>> tire = SnowTire('P', 205, 65, 15, 2)
        >>> tire.circumference()
        92.7
        """
        side_wall_inches = self._side_wall_inches()
        total_diameter = (side_wall_inches + self.chain_thickness) * 2
+ self.diameter
        return round(total_diameter * math.pi, 1)

    def __repr__(self):
        return super().__repr__() + " (Snow)"
```

This implementation is clean, and allows us to avoid repeating ourselves just to add a small modification to the __repr__ output. Additionally, we can (and should) use super as part of the __init__ customizations that we made earlier. The existing implementation was how it would be done in Python 2, and you might see it from time to time. But in Python 3, we can leverage super in the exact way that we did with __repr__. Let's clean up our __init__ method:

~/learning_python/tire.py

```
# Implementation of Tire omitted

class SnowTire(Tire):
```

```
    def __init__(self, tire_type, width, ratio, diameter,
chain_thickness, brand='', construction='R'):
        super().__init__(tire_type, width, ratio, diameter, brand,
construction)
        self.chain_thickness = chain_thickness

    def circumference(self):
        """
        The circumference of a tire w/ show chains in inches.

        >>> tire = SnowTire('P', 205, 65, 15, 2)
        >>> tire.circumference()
        92.7
        """
        side_wall_inches = self._side_wall_inches()
        total_diameter = (side_wall_inches + self.chain_thickness) * 2
+ self.diameter
        return round(total_diameter * math.pi, 1)

    def __repr__(self):
        return super().__repr__() + " (Snow)"
```

The only real differences are that instead of using the Tire constant, we call super() and we also don't need to pass self into the call to __init__. Using super allows us to contain the details about our superclass to the initial declaration, and if we end up changing our superclass later on we won't need to modify other spots where we hardcoded the superclass's name.

CHAPTER 7.4
Polymorphism

Composition works really well for allowing us to reuse code, and one of the other things that it allows us to do is swap out the dependencies that we pass in. This process works because of the idea of polymorphism. In this lesson, we'll learn what polymorphism is and how it's used.

Documentation For This Video
Classes (https://docs.python.org/3/tutorial/classes.html#classes)

What is Polymorphism?

Polymorphism is a pretty strange word that gets used fairly often when talking about object-oriented programming. Thankfully, the concept of polymorphism isn't as complicated as the name would imply. Our Car class is currently taking in a list of Tire objects, but do they need to be Tire instances? Let's take a look at every interaction with the tire instances that happens within the Car class's implementation:

~/learning_python/creating_classes.py

```
class Car:
    """
    Car models a car w/ tires and an engine
    """

    def __init__(self, engine, tires):
        self.engine = engine
        self.tires = tires

    def description(self):
        print(f"A car with a {self.engine} engine, and {self.tires}
tires")

    def wheel_circumference(self):
        if len(self.tires) > 0:
            return self.tires[0].circumference()
        else:
            return 0
```

We interact with the tires in two spots:

When printing in the description method
By calling the circumference method within wheel_circumference
If instead of Tire instances we used strings for the tires attribute,

then we would run into issues because the str type doesn't have a circumference method. Since variables aren't statically typed in Python (they aren't bound to one specific type) the only thing that we need to do to have our Car class work is to pass in tires that meet these requirements:

They can be printed
They implement the circumference method
This is polymorphism. It's the idea that we can make different data structures work together so long as the method requirements between them are met. It means that we can pass SnowTire instances into a Car class where we were currently using Tire instances, and there would be no errors or issues.

```
$ python3.7 -i creating_classes.py
>>> from tire import SnowTire
>>> tire = SnowTire('P', 205, 65, 15, 2)
>>> tires = [tire, tire, tire, tire]
>>> honda = Car(tires=tires, engine='4-cylinder')
>>> honda.wheel_circumference()
92.7
```

Technically, we could create a class called Circle that also implements a circumference method, and that would also work as a "tire" because of polymorphism.

Introduction to Python Development

CHAPTER 8.1
Using Standard Library Packages

One of Python's great strengths is that it comes with a standard library containing many useful modules. In this lesson, we'll learn the various ways that we can use modules, and we'll also take a look at some of the commonly used modules.

Documentation For This Video
Python Modules Tutorial (https://docs.python.org/3/tutorial/modules.html)
Python Module Index (https://docs.python.org/3/py-modindex.html)
The math module (https://docs.python.org/3/library/math.html#module-math)

Using Standard Library Modules
We've already utilized a standard library package when we used the math module to calculate the circumference of a tire. We used one of the variables from the math module in the form of pi, but we loaded the entire module using this line:

import math
Using import we're able to access the internals of the module, by chaining off of the module's name as we did with pi using math.pi, but there are other ways we could have accessed pi. Let's take a look at some of our options:

from math import pi - We can access pi by itself, and we can't reference math because we used a selective import.
from math import pi as p - This would allow us to have access to a p variable that contains the value of pi.
from math import pi, floor, ceil - This would selectively import the pi variable, the floor function, and the ceil function.
from math import * - This would import EVERYTHING (except names starting with an underscore) from the math module into the current namespace. Avoid doing this if possible.
Useful Standard Library Modules
Here are some of the most useful standard library modules that we'll use throughout the remainder of the course.

argparse - for creating CLIs
json - for working with JSON
math - for doing math operations
os - for interacting with operating system resources
pdb - the Python debugger
sys - for interacting with system specific parameters and functions

Introduction to Python Development

CHAPTER 8.2
Working with Third-Party Packages

The standard library is great, but the vast quantity of third-party packages in the Python ecosystem is also at our disposal. In this lesson, we'll learn how to install Python packages and separate our dependencies using a virtualenv.

Documentation For This Video
Installing Python Modules (https://docs.python.org/3/installing/index.html)
Pip (https://pip.pypa.io/en/stable/)
Pipenv (https://pipenv.readthedocs.io/en/latest/)
boto3 (https://boto3.readthedocs.io/en/latest/)

Using pip to Install Packages
As a language with strong open-source roots, Python has a very large repository of open-source packages that can be installed for our use. Thankfully, this repository is easy for us to use, and when we installed Python we were even given the tool to install packages. The simplest tool that we have is pip. Since we have more than one Python installation, we need to make sure that we're using the version of pip that corresponds to the version of Python that we would like to install the package for. With Python 3.7, we'll use pip3.7. Let's install our first package, the popular AWS client library boto3:

```
$ pip3.7 install boto3
...
Installing collected packages: docutils, jmespath, urllib3, six, python-dateutil, botocore, s3transfer, boto3
Could not install packages due to an EnvironmentError: [Errno 13] Permission denied: '/usr/local/lib/python3.7/site-packages/docutils'
Consider using the `--user` option or check the permissions.
$
```

There's an error because we don't have permissions to install a package globally unless we use sudo. If we do use sudo, then any other user on the system that could use our Python 3.7 install would also have access to boto3. An alternative approach is to install the package into a directory for packages only for our user using the --user flag when installing. Let's install the package locally to our user:

```
$ pip3.7 install --user boto3
...
Installing collected packages: urllib3, jmespath, six, python-dateutil, docutils, botocore, s3transfer, boto3
```

```
Successfully installed boto3-1.9.93 botocore-1.12.93 docutils-0.14
jmespath-0.9.3 python-dateutil-2.8.0 s3transfer-0.2.0 six-1.12.0
urllib3-1.24.1
$
```
The boto3 package has some dependencies, so pip also installed those as part of the installation process.

Viewing Installed Packages

If we want to view the packages that are already installed we'll also use the pip for that using the pip freeze command:

```
$ pip3.7 freeze
boto3==1.9.93
botocore==1.12.93
certifi==2018.11.29
Click==7.0
docutils==0.14
Flask==1.0.2
itsdangerous==1.1.0
Jinja2==2.10
jmespath==0.9.3
MarkupSafe==1.1.0
pipenv==2018.11.26
python-dateutil==2.8.0
s3transfer==0.2.0
six==1.12.0
urllib3==1.24.1
virtualenv==16.2.0
virtualenv-clone==0.4.0
Werkzeug==0.14.1
```
Since we installed boto3 with the --user flag, we'll still see it in this list. But a different user would not. The freeze subcommand gives us the information in a format that puts into a file, and then that file could be used to install packages with the specific version. Here's what that would look like:

```
$ pip3.7 freeze > requirements.txt
$ pip3.7 install --user -r requirements.txt
Requirement already satisfied: boto3==1.9.93 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 1)) (1.9.93)
Requirement already satisfied: botocore==1.12.93 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 2)) (1.12.93)
Requirement already satisfied: certifi==2018.11.29 in /usr/local/lib/
python3.7/site-packages (from -r requirements.txt (line 3))
(2018.11.29)
Requirement already satisfied: Click==7.0 in ./.local/lib/python3.7/
site-packages (from -r requirements.txt (line 4)) (7.0)
Requirement already satisfied: docutils==0.14 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 5)) (0.14)
Requirement already satisfied: Flask==1.0.2 in ./.local/lib/python3.7/
```

```
site-packages (from -r requirements.txt (line 6)) (1.0.2)
Requirement already satisfied: itsdangerous==1.1.0 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 7)) (1.1.0)
Requirement already satisfied: Jinja2==2.10 in ./.local/lib/python3.7/
site-packages (from -r requirements.txt (line 8)) (2.10)
Requirement already satisfied: jmespath==0.9.3 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 9)) (0.9.3)
Requirement already satisfied: MarkupSafe==1.1.0 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 10)) (1.1.0)
Requirement already satisfied: pipenv==2018.11.26 in /usr/local/lib/
python3.7/site-packages (from -r requirements.txt (line 11))
(2018.11.26)
Requirement already satisfied: python-dateutil==2.8.0 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 12)) (2.8.0)
Requirement already satisfied: s3transfer==0.2.0 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 13)) (0.2.0)
Requirement already satisfied: six==1.12.0 in ./.local/lib/python3.7/
site-packages (from -r requirements.txt (line 14)) (1.12.0)
Requirement already satisfied: urllib3==1.24.1 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 15)) (1.24.1)
Requirement already satisfied: virtualenv==16.2.0 in /usr/local/lib/
python3.7/site-packages (from -r requirements.txt (line 16)) (16.2.0)
Requirement already satisfied: virtualenv-clone==0.4.0 in /usr/local/
lib/python3.7/site-packages (from -r requirements.txt (line 17))
(0.4.0)
Requirement already satisfied: Werkzeug==0.14.1 in ./.local/lib/
python3.7/site-packages (from -r requirements.txt (line 18)) (0.14.1)
Requirement already satisfied: pip>=9.0.1 in /usr/local/lib/python3.7/
site-packages (from pipenv==2018.11.26->-r requirements.txt (line 11))
(18.1)
Requirement already satisfied: setuptools>=36.2.1 in /usr/local/lib/
python3.7/site-packages (from pipenv==2018.11.26->-r requirements.txt
(line 11))
(40.6.2)
```

Creating a virtualenv

If you're working on multiple packages that have varying dependency requirements, you can run into issues if you're installing packages either globally or localized to a user. Python's solution to this is what's known as a "virtualenv" (for "virtual environment"). A virtualenv is a localized Python install with its own packages, and it can be activated/deactivated. The Python module for creating a virtualenv is called venv, and we can use it by loading the module and providing a path to where we would like to place the virtualenv. Let's create a virtualenv where we can install the package psycopg2:

```
$ mkdir ~/venvs
$ python3.7 -m venv ~/venvs/pg
```

Now we have a virtualenv, but we need to "activate" it by running a script that was created within the virtualenv's bin directory.

```
$ source ~/venvs/pg/bin/activate
(pg) $ python --version
Python 3.7.2
```
The (pg) at the front of our prompt is to indicate to us which virtualenv we currently have active. While this virtualenv is active, the only python in our path is the Python 3.7 that we used to generate it, and pip will install packages for that Python (so we don't need to use pip3.7). Let's install the psycopg2 package:

```
(pg) $ pip install psycopg2
Collecting psycopg2
  Downloading https://files.pythonhosted.org/packages/0c/ba/
e521b9dfae78dc88d3e88be99c8d6f8737a69b65114c5e4979ca1209c99f/
psycopg2-2.7.7-cp37-cp37m-manylinux1_x86_64.whl (2.7MB)
    100% |????????????????????????????????| 2.7MB 14.4MB/s
Installing collected packages: psycopg2
Successfully installed psycopg2-2.7.7
```
To deactivate our virtualenv, we can use the deactivate executable that was put into our $PATH:

```
(pg) $ deactivate
$
```
Using pipenv

The last tool that we're going to look at is a newer tool, pipenv, built to handle both creating and working with a virtualenv, and also managing dependencies. Let's install pipenv and see what the workflow looks like:

```
$ pip3.7 install --user pipenv
...
```
Rather than putting all of the virtualenvs in a directory that we have to navigate to, Pipenv would have us create a project and then initialize a virtualenv for it. Let's create a database project and then initialize a virtualenv for it using Pipenv:

```
$ mkdir ~/database
$ cd ~/database
$ pipenv --python python3.7
Creating a virtualenv for this project…
Pipfile: /home/cloud_user/database/Pipfile
Using /usr/local/bin/python3.7 (3.7.2) to create virtualenv…
? Creating virtual environment...Already using interpreter /usr/local/
bin/python3.7
Using base prefix '/usr/local'
New python executable in /home/cloud_user/.local/share/virtualenvs/
database-OGMn9Yao/bin/python3.7
Also creating executable in /home/cloud_user/.local/share/virtualenvs/
database-OGMn9Yao/bin/python
Installing setuptools, pip, wheel...
done.
```

? Successfully created virtual environment!
Virtualenv location: /home/cloud_user/.local/share/virtualenvs/database-OGMn9Yao
A few things to note here:

Pipenv manages a virtualenv for each project that we set up within ~/.local/share/virtualenvs
We now have a Pipfile which is a more customizable file for us to manage our dependencies in than the requirements.txt file that we generated earlier.
To activate our virtualenv, we can use a new command from within our project directory:

```
$ pipenv shell
Launching subshell in virtual environment…
$  . /home/cloud_user/.local/share/virtualenvs/database-OGMn9Yao/bin/activate
(database) $
```

When we want to add a dependency to our project we also use pipenv:

```
(database) $ pipenv install psycopg2
Installing psycopg2…
Adding psycopg2 to Pipfile's [packages]…
? Installation Succeeded
Pipfile.lock not found, creating…
Locking [dev-packages] dependencies…
Locking [packages] dependencies…
? Success!
Updated Pipfile.lock (59b6f6)!
Installing dependencies from Pipfile.lock (59b6f6)…
  ???????????????????????????????? 1/1 — 00:00:00
```

The big difference between this and just installing the package using pip is that Pipenv will work through a dependency graph to make sure that if two of our dependencies have a common dependency, then we install a version that is compatible with both of our explicit dependencies.

The last thing to note is that to exit our virtualenv, we can simply use exit and it will drop us back to our previous session without the virtualenv loaded.

Introduction to Python Development

CHAPTER 9.1
Interacting with Files

There are some core actions that we need to understand how to do in any programming language, in order to be very productive. One of these actions is interacting with files. In this lesson, we'll learn how to read from and write to files, and we'll take a look at how bytes can be represented in code.

Documentation For This Video
The open function (https://docs.python.org/3/library/functions.html#open)
The file object (https://docs.python.org/3/glossary.html#term-file-object)
The io module (https://docs.python.org/3/library/io.html#io-overview)
Bytes Objects (https://docs.python.org/3.7/library/stdtypes.html#bytes-objects)

Files as Objects
One of the beautiful aspects of working in an object-oriented programming language is that we can represent concepts as objects with functionality. Files are a great use case for this. Python gives us the file object (or concept, really). These objects provide us a few things:

A read method to access the underlying data in the file
A write method to place data into the underlying file
To test this out, we're going to create a simple text file with some names it in, and then read and modify it to see what we can learn.

Opening a File
The first step to interacting with a file is to "open" it, and in Python we'll use the open function. This function takes two main arguments:

file - The path to the file on disk (or where you'd like it to create it)
mode - How you would like to interact with the file
The file argument is pretty simple, but the mode argument has a variety of options that all work a little differently:

'r' - Opens the file for reading, which is the default mode
'w' - Opens the file for writing, while removing the existing content (truncating the file)
'x' - Opens the file to create it, failing if the file already exists
'a' - Opens the file for writing without truncating, appending any new

writes to the end of the file
'b' - Opens the file in binary mode, in which the file expects to write and return bytes objects
't' - Opens the file in text mode, the default mode, where the object expects to write and return strings
'+' - Opens the file for reading and writing
These modes can be used in combination, so w+b is a valid mode saying that we want to read and write with bytes, and with the existing file being truncated (from the w).

Let's create a new script called using_files.py, and we'll start interacting with a file containing some names. The file doesn't exist yet, but if it did, we'd like to truncate it and prepare to write to it.

~/learning_python/using_files.py

```
my_file = open('xmen.txt', 'w+')
```
Now we have a new file object that we can write to.

Writing to the File
Before we can read from our file we need it to have some content. There are a few primary methods that we'll interact for with this depending on whether or not we want to work with lines or individual characters. The write method only writes the characters that we specify, where the writelines method takes a list of strings that should all be on their own line. Let's add some names to our file, each on its own line, using both methods:

~/learning_python/using_files.py

```
my_file = open('xmen.txt', 'w+')
my_file.write('Beast\n')
my_file.write('Phoenix\n')
my_file.writelines([
    'Cyclops',
    'Bishop',
    'Nightcrawler',
])
```
Let's save the file, run it, and then check the contents of xmen.txt:

```
$ python3.7 using_files.py
$ cat xmen.txt
Beast
Phoenix
CyclopsBishopNightcrawler
$
```
This isn't quite what we expected. You would probably think that writelines would add the line ending, but the truth is that we still need to add the '\n' to the end of each item. The writelines method is

more of a shorthand for multiple calls to write unless we used newline='\n' when we opened the file.

Another thing that we didn't do is close the file. When we're finished working with a file, we should call the close method. It's not necessary when running this script because the file handle will be closed when the program terminates. But when we're interacting with files, from within a server for instance, the program won't terminate for a long time.

Reading from a File

Now that we have some content in the file, let's close it within the script, and then re-open it for reading.

~/learning_python/using_files.py

```
my_file = open('xmen.txt', 'w+')
my_file.write('Beast\n')
my_file.write('Phoenix\n')
my_file.writelines([
    'Cyclops\n',
    'Bishop\n',
    'Nightcrawler\n',
])
my_file.close()

my_file = open('xmen.txt', 'r')
print(my_file.read())
my_file.close()
```

Now we can run the script again to see what happens:

```
$ python3.7 using_files.py
Beast
Phoenix
Cyclops
Bishop
Nightcrawler

$
```

Since we're reading the file in 'text' mode, we'll receive a single string from the read method that contains the newline characters and when printed it will print the newlines accordingly. If we didn't want this parsing to occur we could work with the file in bytes mode.

If we were to call the read method again we would receive an empty string in response. The reason for this is that the file holds onto a cursor for the location that it's currently at in the file and when we read it returns everything after that cursor position and moves the cursor to the end. To reread the existing content, we'll need to use seek to move earlier in the file.

The with Statement
Remembering to close a file that we open can be a little tedious and to get around this Python gives us the with statement. A with statement takes an object that has a close method and will call that method after the block has run.

Let's rewrite our existing code to utilize the with statement:

~/learning_python/using_files.py

```
with open('xmen.txt', 'w+') as my_file:
    my_file.write('Beast\n')
    my_file.write('Phoenix\n')
    my_file.writelines([
        'Cyclops\n',
        'Bishop\n',
        'Nightcrawler\n',
    ])

my_file = open('xmen.txt', 'r')
with my_file:
    print(my_file.read())
```

When we open the file to write, we're using the shorthand as expression to open the file within the with statement, and assigning it to the variable my_file within the block. This is a really handy tool if we don't need to use the file in any other way. An alternative would be to create the my_file variable manually, and then pass the variable into the with statement like we did when we were reading from the file.

CHAPTER 9.2
Environment Variables

A common way to configure our programs is to use environment variables. Let's learn how we can access environment variables from inside of our Python code.

Documentation For This Video
The os package (https://docs.python.org/3/library/os.html)
The os.environ attribute (https://docs.python.org/3/library/os.html#os.environ)
The os.getenv function (https://docs.python.org/3/library/os.html#os.getenv)

Working with Environment Variables
By importing the os package, we're able to access a lot of miscellaneous operating system level attributes and functions, not the least of which is the environ object. This object behaves like a dictionary, so we can use the subscript operation to read from it.

Let's create a simple script that will read a 'STAGE' environment variable and print out what stage we're currently running in:

~/learning_python/env_vars.py

```
import os

stage = os.environ["STAGE"].upper()

output = f"We're running in {stage}"

if stage.startswith("PROD"):
    output = "DANGER!!! - " + output

print(output)
```

We can set the environment variable when we run the script to test the differences:

```
$ STAGE=staging python3.7 env_vars.py
We're running in STAGING
$ STAGE=production python3.7 env_vars.py
DANGER!!! - We're running in PRODUCTION
```

What happens if the 'STAGE' environment variable isn't set though?

```
$ python3.7 env_vars.py
Traceback (most recent call last):
```

```
  File "/home/cloud_user/learning_python/env_vars.py", line 5, in
<module>
    stage = os.environ["STAGE"].upper()
  File "/usr/local/lib/python3.7/os.py", line 669, in __getitem__
    raise KeyError(key) from None
KeyError: 'STAGE'
```

This potential KeyError is the biggest downfall of using os.environ, and the reason that we will usually use os.getenv

Handling A Missing Environment Variable

If the 'STAGE' environment variable isn't set then, we want to default to 'DEV', and we can do that by using the os.getenv function:

~/learning_python/env_vars.py

```
import os

stage = os.getenv("STAGE", "dev").upper()

output = f"We're running in {stage}"

if stage.startswith("PROD"):
    output = "DANGER!!! - " + output

print(output)
```

Now if we run our script without a 'STAGE' we won't have an error:

```
$ python3.7 env_vars.py
We're running in DEV
```

CHAPTER 9.3
Error Handling

Not everything can go according to plan in our programs, but we should know when these scenarios arise and handle them appropriately. In this lesson, we'll take a look at how to deal with error handling in Python.

Documentation For This Video
The try statement & workflow (https://docs.python.org/3/reference/compound_stmts.html#the-try-statement)

Handling Errors with try/except/else/finally

When we know that there is a possibility that some our code might raise an error, we don't need to just accept it and let our program crash. We can actually handle these errors using the try statement. This is a compound statement kind of like the if statement where we will also need to use except, and have access to else and finally. Let's break down what these do by writing a small program that will potentially raise an error. We'll call this program handle_errors.py:

~/learning_python/handle_errors.py

```
my_file = open('recipes.txt', 'x')
my_file.write('Meatballs\n')
my_file.close()
```

If we run this script once, then it will run successfully, but if we run it twice we'll see the following error:

```
$ python3.7 handle_errors.py
$ python3.7 handle_errors.py
Traceback (most recent call last):
  File "handle_errors.py", line 1, in <module>
    my_file = open('recipes.txt', 'x+')
FileExistsError: [Errno 17] File exists: 'recipes.txt'
```

The error is a FileExistsError and it's being raised because we're opening the file for creation (using the x mode), but it already exists.

To handle this, we need to place our code that could raise an error within a try statement and then except an error if it happens and do something else.

~/learning_python/handle_errors.py

```
import sys
```

```
file_name = 'recipes.txt'

try:
    my_file = open(file_name, 'x+')
    my_file.write('Meatballs\n')
    my_file.close()
except:
    print(f"The {file_name} file already exists")
    sys.exit(1)
```

This is the simplest kind of try/except and this will catch any error that might be raised by open(file_name, 'x+'). If we run this file again, we should see our print out.

```
$ python3.7 handle_errors.py
The recipes.txt file already exists
```

We could make this more specific and only except a very specific error, and even have multiple separate except blocks catching different kinds of errors. Let's introduce another potential error by passing in a bytes object to a file open in text mode and catch the errors separately:

~/learning_python/handle_errors.py

```
import sys

file_name = 'recipes.txt'

try:
    my_file = open(file_name, 'x+')
    my_file.write(b'Meatballs\n')
    my_file.close()
except FileExistsError as err:
    print(f"The {file_name} file already exists")
    sys.exit(1)
except:
    print(f"Unable to write to the file")
    sys.exit(1)
```

Let's run this with an existing file and without:

```
$ python3.7 handle_errors.py
The recipes.txt file already exists
$ rm recipes.txt
$ python3.7 handle_errors.py
Unable to write to the file
$
```

The else and finally Statements

Now we're able to handle errors, but the error handling workflow also facilitates a way for us to run code if there is no error that gets caught using else, and there's also a way to run some code after any

error handling, or the else block, by using finally. Since we're using sys.exit we wouldn't be able to use finally as is, but let's make some modifications to see how both of these work.

~/learning_python/handle_errors.py

```
import sys

file_name = 'recipes.txt'

try:
    my_file = open(file_name, 'x+')
    my_file.write('Meatballs\n')
    my_file.close()
except FileExistsError as err:
    print(f"The {file_name} file already exists")
except:
    print(f"Unable to write to the {file_name} file")
else:
    print(f"Wrote to {file_name}")
finally:
    print("Execution complete")
```

Lastly, let's give this a run to see how it goes:

```
$ python3.7 handle_errors.py
The recipes.txt file already exists
Execution complete
$ rm recipes.txt
$ python3.7 handle_errors.py
Wrote to recipes.txt
Execution complete
```

Introduction to Python Development

CHAPTER 9.4
Decorators

In addition to being an object-oriented programming language, Python lends itself to applying some ideas from functional programming, because functions are also objects. In this lesson, we're going to take a look at decorators and how they allow us to extend our functions.

Documentation For This Video
The decorator documentation (https://docs.python.org/3/glossary.html#term-decorator)
The classmethod decorator (https://docs.python.org/3/library/functions.html#classmethod)
The staticmethod decorator (https://docs.python.org/3/library/functions.html#staticmethod)
The property decorator (https://docs.python.org/3/library/functions.html#property)

Higher Order Functions
The distinction between referencing a function and calling a function allows us to pass functions into other functions, and return functions from functions. A function that receives a function argument and/or returns a function is what's known as a "higher-order function." And in Python, there's a special syntax that allows us to apply these functions to the functions that we're defining, in order to get additional functionality.

This probably sounds a little complicated, and it can be, but it can be a very powerful tool once we understand what's going on behind the scenes.

Let's create a new file called decorators.py, and create our first higher-order function that receives a function as an argument:

~/learning_python/decorators.py

```
def inspect(func, *args):
    print(f"Running {func.__name__}")
    val = func(*args)
    print(val)
    return val

def combine(a, b):
    return a + b
```

This shows off something that we haven't seen yet with the *args argument. This is a way to capture all remaining positional arguments into a list. It allows inspect to take 1 or more arguments. Functions that can take a variable number of arguments are known as variadic functions. In this case, if we want inspect to be general-purpose, we need it to take in arguments to pass to the func argument. To unpack the arguments and pass them into another function, we need to prefix the variable name with an asterisk *. The documentation for this can provide more information and explains how we could use **kwargs to do a similar thing for keyword arguments.

Let's see how we would use this right now in the REPL:

```
$ python3.7 -i decorators.py
>>> inspect(combine, 1, 2)
Running combine
3
3
```

This is an example of a higher order function that takes a function argument and uses it, but it's not that useful. A more common way to use higher-order functions in Python is by defining decorators which take in a function as an argument and then return a modified version of the function.

Decorators

To make inspect more useful, we're going to modify it so that we can decorate combine when we're defining it. To do this, we'll stop receiving the *args argument, receiving just the function instead, and then we'll define a new function within the context of inspect before we return it. We'll use the decorator syntax to then wrap combine, with inspect above the line defining combine. Let's take a look at this in action:

~/learning_python/decorators.py

```
def inspect(func):
    def wrapped_func(*args, **kwargs):
        print(f"Running {func.__name__}")
        val = func(*args, **kwargs)
        print(f"Result: {val}")
        return val

    return wrapped_func

@inspect
def combine(a, b):
    return a + b
```

Notice that we're now using *args and **kwargs in wrapped_func and this will allow our returning function to handle any arguments before passing them to func.

Now when we call combine it will have the added functionality of inspect because what we're really calling is the wrapped_func returned from inspect:

```
$ python3.7 -i decorators.py
>>> combine(1, b=2)
Running combine
Result: 3
3
```

By using a decorator, we were able to add additional functionality to combine (or any function) without needing to modify the original, pure implementation.

Commonly Used Decorators

Now that we know how decorators work, it would be handy to know when to use them. Adding additional printing to functions is something that we can do, but that doesn't mean it's something we should do.

Some of the most common decorators are classmethod, staticmethod, and property. All of these allow us to modify how method inside of our classes work.

Let's create a new class within decorators.py that uses these decorators to understand what they do:

~/learning_python/decorators.py

```
# inspect and combine omitted
class User:
    base_url = 'https://example.com/api'

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @classmethod
    def query(cls, query_string):
        return cls.base_url + '?' + query_string

    @staticmethod
    def name():
        return "Kevin Bacon"

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"
```

This class does quite a lot, so let's go through it. First, we're creating a class level variable in base_url. This exists on the class itself, and each instance also has a copy of it. Next, we're defining

our initializer as we've done before. The query method is a function that we'd like to exist on the User class itself, not an instance of the class. This is what's known as a "class method." To achieve this, we use the @classmethod decorator, and the User class itself is passed in as the first argument.

The name static method is similar, but a @staticmethod doesn't need an implicit argument. It's really just a function that is attached to the User class, but won't use any of the class's state. Finally, we define the full_name method as a property by using the @property decorator. By doing this, when we reference user_instance.full_name it won't return the function to us, but will instead return the result of the function.

Let's see our class in action:

```
$ python3.7 -i decorators.py
>>> User.name
<function User.name at 0x7fcd82686f28>
>>> User.name()
'Kevin Bacon'
>>> User.query('name=test')
'https://example.com/api?name=test'
>>> user = User('Keith', 'Thompson')
>>> user.base_url
'https://example.com/api'
>>> user.full_name
'Keith Thompson'
```

It's worth noting that the property decorator is actually a class and not a function, and can be used in more complicated ways. I encourage you to read the documentation for it.

Introduction to Python Development

CHAPTER 9.5
Breakpoint Debugging with PDB

When our programs aren't behaving the way that we expect them to, then we need to start
debugging. One of the common ways to do this is by simply adding print lines throughout our code,
to see when something is run and/or what a variable's value is at a given point. But this isn't the only way
to debug. In this lesson, we'll take a look at pdb, the debugger that ships with Python.

Documentation For This Video
The pdb module (https://docs.python.org/3/library/pdb.html)
The pdb.set_trace function (https://docs.python.org/3/library/pdb.html#pdb.set_trace)
The breakpoint function (Python > 3.7) (https://docs.python.org/3/library/functions.html#breakpoint)

Using a Debugger
A debugger is a program that allows us to tie into our program while it's running. By doing
this we can go through the code step by step to see where things are going wrong. This is commonly
known as "breakpoint debugging." Python has its own debugger, the pdb module. This module will allow us to
set breakpoints in our code using the set_trace function, and when this function is run we will be dropped into a Python prompt.

Let's create a few new functions in debugging.py to test this out:

~/learning_python/debugging.py

```
import pdb

def map(func, values):
    output_values = []
    index = 0
    while index < len(values):
        pdb.set_trace()
        output_values = func(values[index])
        index += 1
    return output_values

def add_one(val):
    return val + 1
```

Our map function takes in a function and a list of values and returns list with the result of applying the function to each value in the original list. There are plenty of better ways to do this in Python, but this gives us a useful function to work with. We've imported the pdb module. Within our loop, in map, we've called pdb.set_trace. Let's load up the REPL with this file and use it:

```
$ python3.7 -i debugging.py
>>> map(add_one, list(range(5)))
> /home/cloud_user/learning_python/debugging.py(8)map()
-> output_values = func(values[index])
(Pdb)
```

The (Pdb) signifies that we've hit the breakpoint. From here we can run Python code, and we can see the line directly after our breakpoint (so, it hasn't run yet). Let's interact with the state a little to see what's going on:

```
(Pdb) values
[0, 1, 2, 3, 4]
(Pdb) index
0
(Pdb) output_values
[]
(Pdb)
```

Now that we have a debugger running we're ready to see how to use it through debugger commands.

Debugger Commands

The debugger has its own commands that allow us to do various things while it is running. To see these commands we can enter h or help from within the prompt:

```
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv
undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u
whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb
```

```
(Pdb) help cont
c(ont(inue))
        Continue execution, only stop when a breakpoint is
encountered.
(Pdb) help next
n(ext)
        Continue execution until the next line in the current function
        is reached or it returns.
```

Let's use the n command to execute the next line in our current context and then check the value of output_values:

```
(Pdb) n
> /home/cloud_user/learning_python/debugging.py(9)map()
-> index += 1
(Pdb) output_values
1
```

Here we can see that we have an issue on line 8 of our program, because we're reassigning output_values instead of appending to it.

If we use the c, cont, or continue commands (all the same thing), then we'll execute our program until we get to the next breakpoint. Unfortunately, our breakpoint is in the loop so we'd hit it right away. Let's instead quit out of the debugger using q:

```
(Pdb) q
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "debugging.py", line 9, in map
    index += 1
  File "debugging.py", line 9, in map
    index += 1
  File "/usr/local/lib/python3.7/bdb.py", line 88, in trace_dispatch
    return self.dispatch_line(frame)
  File "/usr/local/lib/python3.7/bdb.py", line 113, in dispatch_line
    if self.quitting: raise BdbQuit
bdb.BdbQuit
>>>
```

The code should be pretty easy to fix, so let's do that now:

~/learning_python/debugging.py

```
def map(func, values):
    output_values = []
    index = 0
    while index < len(values):
        output_values.append(func(values[index]))
        index += 1
    return output_values

def add_one(val):
```

```
    return val + 1

map(add_one, list(range(10)))
```
We've added a call to map at the bottom of the file, so that we can take a look at how we would debug this without putting the pdb module into our code.

Setting Breakpoints from Within Pdb

Adding breakpoints to our code is one way to achieve what we want to do, but we could also run the code with the pdb module loaded into the Python interpreter when we run a script:

```
$ python3.7 -m pdb debugging.py
> /home/cloud_user/learning_python/debugging.py(1)<module>()
-> def map(func, values):
(Pdb)
```
This placed us at the very beginning of the script. We can take a look at the source code by using the longlist or ll command:

```
(Pdb) ll
  1  -> def map(func, values):
  2         output_values = []
  3         index = 0
  4         while index < len(values):
  5             output_values.append(func(values[index]))
  6             index += 1
  7         return output_values
  8
  9     def add_one(val):
 10         return val + 1
 11
 12     map(add_one, list(range(10)))
(Pdb)
```
Now that we know all of the line numbers, we can add breakpoints using the break command, and make it so that our debugger will only stop if it hits one. Let's add a breakpoint to line 5, but only have it stop if index is exactly 5. After we've set the breakpoint, then we'll use c to run until we hit a breakpoint:

```
(Pdb) break 5, index == 5
Breakpoint 1 at /home/cloud_user/learning_python/debugging.py:5
(Pdb) c
> /home/cloud_user/learning_python/debugging.py(5)map()
-> output_values.append(func(values[index]))
(Pdb) index
5
(Pdb)
```
Checking the value of index we can see that our conditional breakpoint worked. From here, we're able to step through the code line by line or simply continue on with the program. If the program continues and

finished then the debugger will restart the program and we'll go back to the beginning.

There's more that you can do with Pdb, but we can use the built-in help information and the documentation (https://docs.python.org/3/library/pdb.html) to learn more.

Introduction to Python Development

CHAPTER 10.1
Project Setup

Now that we know about all of the standard building blocks that we have at our disposal,
we're going to take the time in the remaining sections of the course to look at
different types of projects that we could build using Python. For this section, we'll build a command-line tool to create database backups.
In this lesson, we'll be setting up our project with some
documentation and learn about the most important file for any Python package: setup.py.

Documentation for This Video
Distributing Python Modules (https://docs.python.org/3/distributing/index.html)
Packaging Python Projects (https://packaging.python.org/tutorials/packaging-projects/)
Python Gitignore File (https://github.com/github/gitignore/blob/master/Python.gitignore)

Our Project
For this project, our goal is to create a simple CLI that will allow us to do the following:

Back up a PostgreSQL database using pgdump
Write the backup locally or upload the backup to AWS S3
By the time we're finished, our tool will work like this.

Backing up to S3:

```
$ pgbackup postgres://bob@example.com:5432/db_one --driver s3 backups
```

Back up locally:

```
$ pgbackup postgres://bob@example.com:5432/db_one --driver local /var/local/db_one/backups
```

Setting Up the Project Directory
For Python projects that we intend on distributing as a package, we only really need to have one thing: a setup.py file. This file is used by pip (specifically setuptools behind the scenes) to know what to do to install the package.

Let's create a project directory called pgbackup with some subdirectories, a README.md file for documentation, and a setup.py file.

```
$ mkdir -p ~/projects/pgbackup/src/
$ cd ~/projects/pgbackup
$ touch README.md setup.py src/.gitkeep
```
The .gitkeep file that we added is so that we can commit an "empty" directory with Git before we write any of the projects source code.

To go through setting up our setup.py file, we'll only need to make a few adjustments from what is done in the official packaging tutorial (https://docs.python.org/3/distributing/index.html). Here's what our's looks like:

~/projects/pgbackup/setup.py

```
from setuptools import find_packages, setup

with open('README.md', 'r') as f:
    long_description = f.read()

setup(
    name='pgpackup',
    version='0.1.0',
    author='Keith Thompson',
    author_email='keith@linuxacademy.com',
    description='A utility for backing up PostgreSQL databases',
    long_description=long_description,
    long_description_content_type='text/markdown',
    url='https://github.com/your_username/pgbackup',
    packages=find_packages('src')
)
```
We want the bulk of our documentation to exist within our README.md file. This file will give us some documentation if we use a remote repository host like GitHub or GitLab. It also means that we won't have to modify this file often for the information about the package to change.

The two biggest things to note are that we needed to use the setup function from setuptools and that we used find_packages to ensure that any sub packages that we create will be picked up as part of our project without us needing to modify this file.

While we're still working with Python code, we should also initialize our project with a virtualenv using Pipenv:

```
$ pipenv --python python3.7
Creating a virtualenv for this project…
Pipfile: /home/cloud_user/projects/pgbackup/Pipfile
Using /usr/local/bin/python3.7 (3.7.2) to create virtualenv…
? Creating virtual environment...Already using interpreter /usr/local/
bin/python3.7
Using base prefix '/usr/local'
```

```
New python executable in /home/cloud_user/.local/share/virtualenvs/
pgbackup-uJWPrEHw/bin/python3.7
Also creating executable in /home/cloud_user/.local/share/virtualenvs/
pgbackup-uJWPrEHw/bin/python
Installing setuptools, pip, wheel...
done.

? Successfully created virtual environment!
Virtualenv location: /home/cloud_user/.local/share/virtualenvs/
pgbackup-uJWPrEHw
Creating a Pipfile for this project…
$ pipenv shell
Writing Our README
Our README.md file will document a few different things:

How to use the tool.
How to get up and running when developing the project.
How to install the project for use from source.
~/projects/pgbackup/README.md



pgbackup
CLI for backing up remote PostgreSQL databases locally or to AWS S3.

Usage
Pass in a full database URL, the storage driver, and destination.

S3 Example w/ bucket name:

$ pgbackup postgres://bob@example.com:5432/db_one --driver s3 backups
Local Example w/ local path:

$ pgbackup postgres://bob@example.com:5432/db_one --driver local /var/
local/db_one/backups
Installation From Source
To install the package after you've cloned the repository, you'll want
to run the following command from within the project directory:

$ pip install --user -e .
Preparing for Development
Follow these steps to start developing with this project:

Ensure pip and pipenv are installed
Clone repository: git clone git@github.com:example/pgbackup
cd into the repository
Activate virtualenv: pipenv shell
Install dependencies: pipenv install

Adding a Gitignore File
```

Before we commit anything, we're going to pull in a default Python .gitignore (https://github.com/github/gitignore/blob/master/Python.gitignore) file from Github so that we don't track files in our repository that we don't need:

```
(pgbackup) $ curl -o .gitignore https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore
...
```

Our Initial Commit

Now that we've created our setup.py, README.md, and .gitignore files, we're in a good position to stage our changes and make our first commit:

```
$ git init
$ git add --all .
$ git commit -m 'Initial commit'
```

Introduction to Python Development

CHAPTER 10.2
Setting Up External Dependencies

Before we dig too far into our pgbackup project we'll need to make sure that we have access to AWS S3 buckets and also have a PostgreSQL database to interact with. In this lesson, we'll use the Linux Academy Cloud Playground to deploy the database and set up an AWS user with access to S3.

Note: You'll need to have an AWS account and be able to create a new user in IAM.

Documentation For This Video
db_setup.sh (https://raw.githubusercontent.com/linuxacademy/content-intro-to-python-development/master/helpers/db_setup.sh)
PostgreSQL RPM (https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-7-x86_64/pgdg-centos96-9.6-3.noarch.rpm)
The AWS CLI (https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html)

Setting up PostgreSQL Cloud Playground
Before we begin, we're going to need to need a PostgreSQL database to work with. The code repository for this course contains a db_setup.sh script that we'll use on a CentOS 7 Cloud Playground to create and run our database. Create a "CentOS 7" Cloud Playground and run the following on it:

```
$ curl -o db_setup.sh https://raw.githubusercontent.com/linuxacademy/content-python-for-sys-admins/master/helpers/db_setup.sh
$ chmod +x db_setup.sh
$ ./db_setup.sh
```

You will be prompted for your sudo password and for the username and password you'd like to use to access the database.

Installing The Postgres 9.6 Client
On our development machines, we'll need to make sure that we have the Postgres client installed. The version needs to be 9.6.

On Red-hat systems we'll use the following:

```
$ wget https://download.postgresql.org/pub/repos/yum/9.6/redhat/rhel-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm
$ sudo yum install -y pgdg-redhat-repo-latest.noarch.rpm
$ sudo yum update -y
$ sudo yum autoremove -y postgresql
```

```
$ sudo yum install -y postgresql96
```
On debian systems, the equivalent would be:

```
$ sudo apt-get install postgres-client-9.6
```
Test connection from Workstation
Let's make sure that we can connect to the PostgreSQL server from our development machine by running the following command:

*Note: You'll need to substitute in your database user's values for [USERNAME], [PASSWORD], and [SERVER_IP].

```
$ psql postgres://[USERNAME]:[PASSWORD]@[SERVER_IP]:80/sample -c
"SELECT count(id) FROM employees;"
```
Setting Up Our AWS User
The last external dependency that we have is Amazon S3, so we're going to need an AWS account for this step.

NOTE: Linux Academy Cloud Sandboxes do not permit IAM user creation, so you must use your own account. AWS Free Tier accounts are available [here](https://aws.amazon.com/free/).

Once we're logged in we need to do the following:

Open IAM console
Select "Users" from the sidebar
Click "Add User"
Set the user name, and give programmatic access
Select "Attach existing policies directly"
Add the following permissions to the new user:
AmazonS3FullAccess
Continue through tags and review.
Now we have a user, copy the "Access Key ID" and the "Secret access key" so that we can use them in a moment.

Installing and Configuring the AWS CLI
The AWS CLI is written in Python and we can install it globally by exiting our virtualenv for a moment and running this command:

```
(pgbackup) $ exit
$ pip3.7 install --user awscli
```
Next, we'll take the access key ID and secret access key that we copied before to configure out AWS client:

```
$ aws configure
AWS Access Key ID [None]: XXXXXXXXXXXXXXXXXXX
AWS Secret Access Key [None]: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Default region name [None]:
Default output format [None]:
```
That will create a ~/.aws directory that includes our default configuration and credentials.

Creating Our S3 Bucket

From within the AWS Console, we need to create a new S3 bucket that we can use with our project. To do this, we need to search for "S3" in the services. Next, click "Create Bucket", give it a name, and use the default settings.

Once the bucket is created, we'll select it by name and go to the "Permissions" tab so that we can change a few things. From the "Permissions" tab, select the "Access Control List" sub-tab and then for "Public Access" allow "List objects".

Now we should be able to add items to our S3 bucket and make them publically readable to make things a little easier for us later.

Introduction to Python Development

CHAPTER 10.3
Building the CLI: Handling Arguments and Flags

Our project has a few different discrete parts and we're going to build them in isolation before eventually connecting them together. In this lesson, we'll go about building the CLI that handles the user interaction and gets the variables that we need.

Documentation For This Video
The argparse package (https://docs.python.org/3/library/argparse.html)
The argparse.ArgumentParser class (https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser)
The argparse.Action class (https://docs.python.org/3/library/argparse.html#argparse.Action)

Creating Our pgbackup Package
Up to this point pgbackup exists as a project, but not a package. We'll be creating quite a few different modules and want them to exist within the pgbackup package. To create a package we need to have a directory with our package name that includes a __init__.py file. Let's create that now within src:

```
(pgbackup) $ mkdir src/pgbackup
(pgbackup) $ touch src/pgbackup/__init__.py
```

Now we can create the rest of our modules within the src/pgbackup directory.

The argparse Package
When it comes to creating CLIs in Python, the standard library comes with a great package: the argparse package. Let's create a new Python file that will hold onto our CLI related logic at src/pgbackup/cli.py. In this file, we're going to write a function that will configure and return a parse that we can use when we finally run our tool.

Here's our initial implementation:

src/pgbackup/cli.py

```
from argparse import ArgumentParser

def create_parser():
    parser = ArgumentParser(description="""
    Back up PostgreSQL databases locally or to AWS S3.
    """)
    return parser
```

The ArgumentParser class is going to be the backbone of our user interface. It makes it easy for us to define the arguments and options available from our CLI.

Handling Arguments and Flags
If we remember back to how we stated our tool would work in the documentation (README.md), we can see that we need a few things:

```
$ pgbackup postgres://bob@example.com:5432/db_one --driver s3 backups
```

or

```
$ pgbackup postgres://bob@example.com:5432/db_one --driver local /var/local/db_one/backups
```

We need:

A positional argument for the url of the database we're connecting to.
A 2 part flag that gives us the driver name, and a context specific value of either the S3 bucket name or the local path to back up the database.
The positional argument will be easy to define, but the optional argument is actually more complicated than we can handle with an ArgumentParser out of the box, but the argparse.Action class will allow us to create a custom flag handler class. Here's what our implementation will look like to handle the arguments and flags:

~/projects/pgbackup/src/pgbackup/cli.py

```
from argparse import Action, ArgumentParser

class DriverAction(Action):
    def __call__(self, parser, namespace, values, option_string=None):
        driver, destination = values
        namespace.driver = driver.lower()
        namespace.destination = destination

def create_parser():
    parser = ArgumentParser(description="""
    Back up PostgreSQL databases locally or to AWS S3.
    """)
    parser.add_argument("url", help="URL of database to backup")
    parser.add_argument("--driver",
            help="how & where to store backup",
            nargs=2,
            action=DriverAction,
            required=True)
    return parser
```

By defining a custom "action" we're able to specify that the --driver flag will actually populate 2 attributes on our arguments namespace when we're finished parsing the arguments.

Manually Testing the Parser
As we've done many times before, we're going to pull our code into the REPL so that we can test it out to see if it works. Ideally, we would write some automated tests that could verify that the function works as expected, but that's a little beyond what we're trying to learn right now.

Let's see our parser in action:

```
(pgbackup) $ $ python -i src/pgbackup/cli.py
>>> parser = create_parser()
>>> args = parser.parse_args(['https://some_url', '--driver', 's3',
'bucket_name'])
>>> args.url
'https://some_url'
>>> args.driver
's3'
>>> args.destination
'bucket_name'
>>> parser.parse_args()
usage: cli.py [-h] --driver DRIVER DRIVER url
cli.py: error: the following arguments are required: url, --driver
```

The create_parser function will return our pre-configured parser and then to use it we'll call the parse_args method. If we don't pass anything to the parse_args method then it will automatically parse the arguments in sys.argv. To the parser out with expected arguments though, we can pass in a list of the the tokens that would come from stdin.

By using add_argument with a --driver we were specifying that this is a flag. The url argument is positional because we didn't add dashes. With this implemented we're ready to move onto another section of the tool.

Before we conclude, let's not forget to commit:

```
(pgbackup) $ git add --all .
(pgbackup) $ git commit -m 'Create pgbackup package and cli module'
```

Introduction to Python Development

CHAPTER 10.4
Interacting with External Processes: Utilizing `pg_dump`

When you install the standard PostgreSQL client on a Unix machine, you also get access to the pg_dump utility. Interfacing with this tool will allow us to easily get backups from a PostgreSQL database. In this lesson, we'll learn how to interact with processes external to Python so that we can trigger a pg_dump.

Documentation for This Video

The subprocess package (https://docs.python.org/3/library/subprocess.html)
The subprocess.Popen class (https://docs.python.org/3/library/subprocess.html#subprocess.Popen)
The sys.exit function (https://docs.python.org/3/library/sys.html#sys.exit)

Interacting with Subprocesses

The pg_dump utility is great for interacting with PostgreSQL databases so we're going to interact with the tool directly from within our code using the subprocess package. Since we're building a thin wrapper for the pg_dump utility, we're going to put this code into a pgdump.py file:

src/pgbackup/pgdump.py

```
import subprocess

def dump(url):
    return subprocess.Popen(['pg_dump', url], stdout=subprocess.PIPE)
```

Remembering back to the desired UX for our tool, we're expecting to receive a URL for the database, so we're writing a function that will receive this URL. Thankfully, pg_dump can also receive a URL, so we build a list of tokens to build up the external command to run. We're utilizing subprocess.PIPE to capture Stdout into a file-like object and prevent it from being written to the terminal when we run this code.

This first draft is pretty simple, but there's no guarantee that it will always succeed, so we need to add some error handling.

Implementing Error Handling

The subprocess.Popen can raise an OSError so we're going to wrap that call in a try block, print the error message, and use sys.exit to set the error code.

src/pgbackup/pgdump.py

```
import sys
import subprocess

def dump(url):
    try:
        return subprocess.Popen(['pg_dump', url],
stdout=subprocess.PIPE)
    except OSError as err:
        print(f"Error: {err}")
        sys.exit(1)
```

By using try and except we can ensure that the Python stack trace won't be printed out in the terminal of the person using this project if pg_dump happens to not be installed.

Manual Testing

Now that we've implemented our subprocess interaction, we'll take a moment to ensure that it actually works. We can load our entire project into the Python REPL by setting the PYTHONPATH environment variable to be our ./src directory:

```
(pgbackup) $ PYTHONPATH=./src python
>>> from pgbackup import pgdump
>>> dump = pgdump.dump('postgres://demo:password@54.245.63.9:80/
sample')
>>> f = open('dump.sql', 'w+b')
>>> f.write(dump.stdout.read())
>>> f.close()
```

Note: We needed to open our dump.sql file using the w+b flag because we know that the .stdout value from a subprocess will be a bytes object and not a str.

If we exit and take a look at the contents of the file using cat, we should see the SQL output. With the pgdump module implemented, it's now a great time to commit our code.

Introduction to Python Development

CHAPTER 10.5
Storing Data Locally

Now that we're able to get the database information from our remote server, we're ready to implement the two backup methods that we need. To start, we'll add the local storage functionality.

Documentation for This Video
BufferedWriter.read method (https://docs.python.org/3/library/io.html#io.BufferedWriter.read)
BufferedWriter.write method (https://docs.python.org/3/library/io.html#io.BufferedWriter.write)
IOBase.close method (https://docs.python.org/3/library/io.html#io.IOBase.close)

Implement Local Storage

Our pgbackup.dump method is returning subprocess.Popen and that object has a stdout attribute that acts like a file. We also want to write the contents of the backup to a file, so we can write a function that assumes it will receive two open files and transfers the contents from one file to another. Let's add all of the code related to storing the backup into a new file called storage.py.

By limiting our scope in that way, we can write a fairly simple function that creates the local backup. Here's what we plan to do:

Receive 2 open files: an infile and outfile.
Read the contents from the infile and write them to the outfile.
Close both of the files.
We want to call close on the "writeable" file to ensure that all of the content gets written (the database backup could be quite large).

src/pgbackup/storage.py

```
def local(infile, outfile):
    outfile.write(infile.read())
    outfile.close()
    infile.close()
```

Introduction to Python Development

CHAPTER 10.6
Interacting with AWS S3

The last individual unit of functionality that we need to implement is storing data in AWS S3. In this lesson, we'll expand our storage module by adding a way to write a file to S3.

Documentation for This Video
The boto3 package (https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj)

Installing boto3
To interface with AWS (S3 specifically), we're going to use the wonderful boto3 package. We can install this to our virtualenv using pipenv now:

```
(pgbackup) $ pipenv install boto3
```

With boto3 installed, we'll be able to connect to S3 since it will read the same configuration files that we created in an earlier lesson using the AWS CLI.

Implementing S3 Storage
The storage module that we've already created is the perfect place to put our code that interacts with S3 since it's just another form of storage. We'll create another function for s3 that will take a few things:

client: An AWS client object that has an upload_fileobj method.
infile: A file object with the data from our PostgreSQL backup.
bucket: The name of the bucket that we'll be storing the backup in.
name: The name of the file we'd like to create in our S3 bucket.
The reason that we are injecting the client object is that we don't want our storage module to be in charge of configuring the client, we'll do that when we tie all of the pieces together.

The upload_fileobj method works in the exact way that we need it to, so our implementation should be pretty simple:

src/pbgackup/storage.py

```
# ... previous function ommited
def s3(client, infile, bucket, name):
    client.upload_fileobj(infile, bucket, name)
```

Manually Testing S3 Integration
Like we did with our PostgreSQL interaction, let's manually test uploading a file to S3 using our storage.s3 function. First, we'll

create an example.txt file, and then we'll load into a Python REPL with our code loaded:

```
(pgbackup) $ echo "UPLOADED" > example.txt
(pgbackup) $ PYTHONPATH=./src python
>>> import boto3
>>> from pgbackup import storage
>>> client = boto3.client('s3')
>>> infile = open('example.txt', 'rb')
>>> storage.s3(client, infile, 'python-backups', infile.name)
```

When we check our S3 console, we should see the file there. Lastly, remove the example.txt file and then commit these changes:

```
(pgbackup) $ rm example.txt
(pgbackup) $ git add .
(pgbackup) $ git commit -m 'Implement S3 interactions'
```

Introduction to Python Development

CHAPTER 10.7
Wiring the Pieces Together

With all of the functionality built into small, separate units, we're now ready to combine them to build our pgbackup tool. In this lesson, we'll create our main function that we can run when the tool is used.

Documentation for This Video
The boto3 package (https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj)
The setuptools script creation (https://setuptools.readthedocs.io/en/latest/setuptools.html#automatic-script-creation)
The time.strftime function (https://docs.python.org/3/library/time.html#time.strftime)

An Overview of the Parts
We've successfully written the following:

CLI parsing
Postgres Interaction
Local storage driver
AWS S3 storage driver
Now we want to create a function that we can use as a "console_script". Console scripts allow us to have setuptools create an executable script that we can install when someone installs our package.

Add "console_script" to project
We can make our project create a console script for us when a user runs pip install. This is similar to the way that we make executables before, except we don't need to manually do the work. To do this, we need to add an entry point in our setup.py:

setup.py (partial)

```
    package_dir={'': 'src'},
    install_requires=['boto3'],
    entry_points={
        'console_scripts': [
            'pgbackup=pgbackup.cli:main',
        ],
    }
```

Notice that we're referencing our cli module with a : and a main. That main is the function that we need to create now.

Wiring the Units Together

Our main function is going to go into the cli module, and it needs to do the following:

Import the boto3 package.
Import our pgdump and storage modules.
Create a parser and parse the arguments.
Fetch the database dump.
Depending on the driver type, do either:
Create a boto3 S3 client and use storage.s3
or
Open a local file and use storage.local
src/pgbackup/cli.py

```
def main():
    import boto3
    from pgbackup import pgdump, storage

    args = create_parser().parse_args()
    dump = pgdump.dump(args.url)
    if args.driver == 's3':
        client = boto3.client('s3')
        # TODO: create a better name based on the database name and
the date
        storage.s3(client, dump.stdout, args.destination,
'example.sql')
    else:
        outfile = open(args.destination, 'wb')
        storage.local(dump.stdout, outfile)
```

Let's test it out:

```
$ pipenv shell
(pgbackup) $ pip install -e .
(pgbackup) $ pgbackup --driver local ./local-dump.sql postgres://
demo:password@54.245.63.9:80/sample
(pgbackup) $ pgbackup --driver s3 pyscripting-db-backups postgres://
demo:password@54.245.63.9:80/sample
```

Reviewing the Experience

It worked! That doesn't mean there aren't things to improve though. Here are some things we should fix:

Generate a good file name for S3
Create some output while the writing is happening
Create a shorthand switch for --driver (-d)

Generating a Dump File Name

For generating our file name, let's put all database URL interactions in the pgdump module with a function name of dump_file_name. We want the file name returned to be based on the database name, and it should also accept an optional timestamp. Let's work on the implementation now:

src/pgbackup/pgdump.py (partial)

```
def dump_file_name(url, timestamp=None):
    db_name = url.split("/")[-1]
    db_name = db_name.split("?")[0]
    if timestamp:
        return f"{db_name}-{timestamp}.sql"
    else:
        return f"{db_name}.sql"
```

Improving the CLI and Main Function

We want to add a shorthand -d flag to the driver argument, let's add that to the create_parser function:

src/pgbackup/cli.py (partial)

```
def create_parser():
    parser = argparse.ArgumentParser(description="""
    Back up PostgreSQL databases locally or to AWS S3.
    """)
    parser.add_argument("url", help="URL of database to backup")
    parser.add_argument("--driver", "-d",
            help="how & where to store backup",
            nargs=2,
            metavar=("DRIVER", "DESTINATION"),
            action=DriverAction,
            required=True)
    return parser
```

Lastly, let's generate a timestamp with time.strftime, generate a database file name, and print what we're doing as we upload/write files.

src/pgbackup/cli.py (partial)

```
def main():
    import time
    import boto3
    from pgbackup import pgdump, storage

    args = create_parser().parse_args()
    dump = pgdump.dump(args.url)

    if args.driver == 's3':
        client = boto3.client('s3')
        timestamp = time.strftime("%Y-%m-%dT%H:%M", time.localtime())
        file_name = pgdump.dump_file_name(args.url, timestamp)
        print(f"Backing database up to {args.destination} in S3 as
{file_name}")
        storage.s3(client,
                dump.stdout,
                args.destination,
```

```
                file_name)
    else:
        outfile = open(args.destination, 'wb')
        print(f"Backing database up locally to {outfile.name}")
        storage.local(dump.stdout, outfile)
```

Feel free to test the CLI's modifications and commit these changes.

Introduction to Python Development

CHAPTER 10.8
Distributing the Package

Our pgbackup utility is most likely going to be something that we distribute internally in our organization. To make that easy, we'll want to build an installable "wheel" for it. In this lesson, we'll learn how to build installable distributions for our projects.

Note: Building a wheel works for both proprietary and open source projects.

Documentation For This Video
The python_requires documentation (https://packaging.python.org/guides/distributing-packages-using-setuptools/#python-requires)

Configuring a Required Python Version
Before we can generate our wheel, we're going to want to configure setuptools to know that it requires Python 3.6 or newer because we used the "f-string" syntax. We'll use the python_requires argument and the setuptools.setup method within our setup.py to specify this:

setup.py (partial)

```
    package_dir={'': 'src'},
    install_requires=['boto3'],
    python_requires='>=3.6',
    entry_points={
        'console_scripts': [
            'pgbackup=pgbackup.cli:main',
        ],
    }
```

Now we can run the following command to build our wheel:

```
(pgbackup) $ python setup.py bdist_wheel
```

Next, let's uninstall and re-install our package using the wheel file:

```
(pgbackup) $ pip uninstall pgbackup
(pgbackup) $ pip install dist/pgbackup-0.1.0-py3-none-any.whl
```

Install a Wheel From Remote Source (S3)
We can use pip to install wheels from a local path, but it can also install from a remote source over HTTP. Let's upload our wheel to S3 and then install the tool outside of our virtualenv from S3:

```
(pgbackup) $ python
>>> import boto3
>>> f = open('dist/pgbackup-0.1.0-py3-none-any.whl', 'rb')
```

```
>>> client = boto3.client('s3')
>>> client.upload_fileobj(f, 'python-backups', 'pgbackup-0.1.0-py3-
none-any.whl')
>>> exit()
```
We'll need to go into the S3 console and make this file public so that we can download it to install.

Let's exit our virtualenv and install pgbackup as a user package:

```
(pgbackup) $ exit
$ pip3.7 install --user https://s3.amazonaws.com/python-backups/
pgbackup-0.1.0-py3-none-any.whl
$ pgbackup --help
```

Introduction to Python Development

CHAPTER 11.1
Visualizing Web Development

Another popular use case for Python is web development. In this lesson, we'll learn a little bit about the basics of how web applications work and also look at the application that we'll be building throughout this section. We'll be using Flask as our web framework for this section of the course.

Documentation For This Video
PDF of video's slideshow (https://linuxacademy.com/cp/guides/download/refsheets/guides/refsheets/s5-video1-visualizing-our-web-application_1552078244.pdf)
Flask (http://flask.pocoo.org/)

The Life Span of a Web Request
Web development becomes a lot easier once you know how web requests are handled by applications in general. This is the life cycle of a request through a web application:

Make an HTTP request - Whether this is from curl, code, or a web browser, an HTTP request needs to be made. This request will include headers, a body, and the "request method" which will usually be one of the following (there are some other methods infrequently used):
GET - This requests a representation of a specific resource. These requests should only receive data, not cause changes.
HEAD - It's like a GET request, but only receives the headers (no response body).
POST - These submit data to the application, and usually cause a change in the application's state.
PUT - This replaces the current representation of the target resource.
DELETE - Just like it sounds, it deletes the specified resource.
PATCH - These partially modify a resource.
The application receives the request and routes the request to the proper handler in code.
The handler executes the expected code and returns the corresponding response. The response includes headers and a body.
Aspects of Our Web Application
For our notes application, we're going to be structuring our code using the MVC pattern: Model, View, Controller. At its core, MVC architecture revolves around 3 concepts:

Model - Holds onto the data and business logic:
Ideally, the model holds onto "the brains" of our application.
View - The representation of our business logic:
For our application, the view portion will be the HTML templates that

we render and return.
Controller - The input/output interface for the application:
The controller receives requests and utilizes the model and view to display the proper response. In Flask, there are a few different ways to create a "controller," but in its simplest form the controller will actually be the Flask application itself.
A common problem with MVC architecture is that people put too much logic into the controller and view portions, but we'll do our best to keep the important contents in our Model.

Flask is a micro-framework that doesn't force us to use an MVC architecture, and it really doesn't provide us with any structure for doing so, but we can achieve this distinction on our own by building our own distinct modules.

Introduction to Python Development

CHAPTER 11.2
Project Setup

Our first step in building a web application (or any application really) is to set up the basic project structure.
In this lesson, we'll set up our Flask application and the database that we'll use while developing.

Documentation For This Video
Flask (http://flask.pocoo.org/)
Pipenv (https://pipenv.readthedocs.io/en/latest/)
Markdown (https://daringfireball.net/projects/markdown/syntax)

Our Application: Markdown Notes
For this portion of the course, we're going to be building a note-taking application that allows us to write notes using Markdown. Ideally, this application would be great for storing code snippets with some notes about what we were doing and how the code works.

The Basic Directory Structure
Being a micro-framework, Flask doesn't have a lot of opinions about how we structure our project. Because of this, we're going to keep it as simple as we can and to start, we won't have many directories at all. Here are the things that we want to create:

Pipfile (via pipenv install) - To specify our Python version and dependencies
/templates - To eventually hold Jinja templates that we'll use to render HTML
/static - To hold static assets like HTML, CSS, and JavaScript files
Let's create these things:

```
$ cd ~/projects
$ mkdir notes
$ cd notes
$ pipenv --python python3.7 install flask
...
$ pipenv shell
(notes) $ mkdir templates static
(notes) $ touch {templates,static}/.gitkeep
```

For right now that's all we need to set up. Let's pull down a reasonable gitignore from Github and write our initial commit.

```
(notes) $ curl -o .gitignore https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore
(notes) $ git init
```

```
Initialized empty Git repository in /home/cloud_user/projects/
notes/.git/
(notes) $ git add --all .
(notes) $ git commit -m 'Initial commit'
[master (root-commit) ef81eca] Initial commit
 5 files changed, 218 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Pipfile
 create mode 100644 Pipfile.lock
 create mode 100644 static/.gitkeep
 create mode 100644 templates/.gitkeep
```

Now we have the basic directory structure that we want for our application to start with and we're ready to start developing.

CHAPTER 11.3
Creating the Flask Application and Database

The first step in our web development process is to actually have a running application, but this application doesn't need to do anything at the start. In this lesson, we'll create our application, ensure that it can run, and configure it to connect to our PostgreSQL database.

Documentation For This Video

The Official Flask Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/factory/)
SQLAlchemy (https://www.sqlalchemy.org/)
python-dotenv (https://github.com/theskumar/python-dotenv)

Our Application Factory

There are a few different ways that we can go about creating Flask applications, but we're going to use the "application factory" approach. This approach is shown off in the official Flask tutorial and I really think it's a great way to get started. To begin, let's create an __init__.py in our notes project that will hold our application factory method:

notes/__init__.py:

```
import os

from flask import Flask

def create_app(test_config=None):
    app = Flask(__name__)
    app.config.from_mapping(
        SECRET_KEY=os.environ.get('SECRET_KEY', default='dev'),
    )

    if test_config is None:
        app.config.from_pyfile('config.py', silent=True)
    else:
        app.config.from_mapping(test_config)

    return app
```

There are a few things to note here:

We're naming our Flask application based on the __name__ value which will correspond to the FLASK_APP environment variable.
Our function can receive an optional test_config object so that it can be configured differently if we ever decide to write automated tests.

The app.config.from_mapping call allows us to set configuration constants. We need to have a SECRET_KEY and we're setting that now, but should load a specific value from the environment when this application is deployed.
The final thing that we need to do is return the application from the function so that the Flask CLI can run it.
To run our application, we'll use the flask CLI after setting a few environment variables:

```
(notes) $ export FLASK_ENV=development
(notes) $ export FLASK_APP='.'
(notes) $ flask run --host=0.0.0.0 --port=3000
 * Serving Flask app "." (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://0.0.0.0:3000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 143-698-723
```

The flask CLI will automatically reload code for us as we're developing, and serve our application on port 3000. We'll want to leave this running, so let's connect to the workstation again in a separate terminal tab. We can currently see our empty Flask app by going to our server's public IP address on port 3000.

We should receive a "Not Found" but that's fine. We haven't defined any routes yet, so there are no resources to find.

Creating a Database
For our note-taking application to be useful, we're going to need to be able to store data in a database. We'll be using a PostgreSQL database. If you've already created a PostgreSQL server to follow along with the CLI use case, then you can use that same server. If you haven't, then create a new CentOS 7 Cloud Playground and run the following commands after it's been spun up:

```
$ curl -o db_setup.sh https://raw.githubusercontent.com/linuxacademy/content-python-for-sys-admins/master/helpers/db_setup.sh
$ chmod +x db_setup.sh
$ ./db_setup.sh
```

During this process, the server will update itself, install Docker, create a PostgreSQL database, and expose it. You'll need to enter your sudo password at least once and you'll also need to pick a database username and password. There's a "sample" database populated with some information, but we won't be using that.

While still connected to the Cloud Playground running the database, run the following command to create the notes database:

NOTE: Set [POSTGRES_USER] to whatever you set it to when running

db_setup.sh. For me, that's demo.

```
$ sudo docker exec -i postgres psql postgres -U [POSTGRES_USER] -c
"CREATE DATABASE notes;"
```

Now we have a database that our new application can use.

Configuring Database Access

Before we conclude this lesson, we're going to add some configuration to our Flask application so that it can connect to our new notes database. Our application factory function already references a config.py file, and that's what we're going to create for holding on to our database configuration. We'll be using an "Object Relational Mapper" (ORM) to map our classes to tables in the database later in this project and our ORM)https://www.sqlalchemy.org/) has a very specific configuration value that it wants us to set: the SQLALCHEMY_DATABASE_URI value. This value will be the full URL that we need to connect to the database.

Now that we know what we need to set up, let's create the config.py at the root of our project directory:

notes/config.py

```
import os

db_host = os.environ.get('DB_HOST', default='localhost')
db_name = os.environ.get('DB_NAME', default='notes')
db_user = os.environ.get('DB_USERNAME', default='notes')
db_password = os.environ.get('DB_PASSWORD', default='')
db_port = os.environ.get('DB_PORT', default='5432')

SQLALCHEMY_TRACK_MODIFICATIONS = False
SQLALCHEMY_DATABASE_URI = f"postgres://{db_user}:{db_password}
@{db_host}:{db_port}/{db_name}"
```

We've provided a way for ourselves to use environment variables to configure how we're connecting to the database. In the process, we've also set some defaults that make sense for when you'd be developing this web application on a local workstation, but they don't work for us. We'll need to set the DB_USERNAME, DB_PASSWORD, DB_HOST, and DB_PORT. For the DB_HOST we'll use the public IP address and then also set the DB_PORT to 80 if we're developing on our local machine.

Lastly, in a new terminal session connected to our workstation, we'll export these environment variables so that they'll be used when we run the application. To make it easier on ourselves each time we open the project, we'll put these in a .env file:

notes/.env:

```
export DB_USERNAME='demo'
```

```
export DB_PASSWORD='secure_password'
export DB_HOST='<PUBLIC_IP>'
export DB_PORT='80'
export FLASK_ENV='development'
export FLASK_APP='.'
```
The .env file is already in our .gitignore file, so we don't have to worry about committing our username and password into our repository. Let's make sure that we can still start the application:

```
(notes) $ source .env
(notes) $ flask run --host=0.0.0.0 --port=3000
 * Tip: There are .env files present. Do "pip install python-dotenv"
to use them.
 * Serving Flask app "." (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://0.0.0.0:3000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Tip: There are .env files present. Do "pip install python-dotenv"
to use them.
 * Debugger is active!
 * Debugger PIN: 143-698-723
```
Notice that it's suggesting that we install python-dotenv to have it use the .env file by default. Let's do this by adding it as a development dependency to our project:

```
(notes) $ pipenv install --dev python-dotenv
...
```
Now when we perform a flask run, it will automatically read in the environment variable within the .env file.

CHAPTER 11.4
Modeling Data with an Object-Relational Mapper

Now that we have our base application structure, we're going to create the models to store the notes that we're going to be writing with our application.

Documentation For This Video
Flask Database Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/database/)
SQLAlchemy (https://www.sqlalchemy.org/0)
Flask-SQLAlchemy (http://flask-sqlalchemy.pocoo.org/2.3/)
psycopg2 (http://initd.org/psycopg/)
Flask-Migrate (https://flask-migrate.readthedocs.io/en/latest/)

Installing SQLAlchemy

Before we start our database modeling, we need to install a few dependencies. Let's add Flask-SQLAlchemy, Flask-Migrate, and psycopg2:

```
(notes) $ pipenv install psycopg2-binary Flask-SQLAlchemy Flask-Migrate
...
```

Now we're ready to start modeling some data.

Modeling a User

Since we're using an object-oriented language, we would really like to work with objects, but relational databases are all about rows. Thankfully, "object-relational mappers" (or ORMs) allow us to map rows from relational databases into instances of classes. For our application, we have two pieces of data that we need to model:

User: An individual that is logged into the application.
Note: An individual note that a User has created.

There are many ways that we could go about creating these, but we're going to define both of the classes in the same file, our models.py. Let's create a User class:

notes/models.py:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
```

```
    password = db.Column(db.String(200))
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(),
server_onupdate=db.func.now())
```

We're doing a few things here. First, we're importing SQLAlchemy from the flask_sqlalchemy package so that we can create an instance to use as our db. This db object will be important for us to connect to the database in our application. Next, we create the User class that inherits from the db.Model class. By inheriting from db.Model we're gaining some functionality that SQLAlchemy provides. Now, we need to define some fields on our class itself. These are columns, and when we read an item in from the database, these columns will be read and get assigned to the attribute on the instance. The created_at and updated_at fields are special in that we don't want to explicitly have to set them, so we have the database server running some functions to automatically set and update the rows.

We also need to model a Note, so let's add this in the same file below our User class:

notes/models.py:

```
# User omitted
class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200))
    body = db.Column(db.Text)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(),
server_onupdate=db.func.now())
```

Our Note model is fairly simple, only showing the additional db.Text field option for a field that can hold a large amount of text. To make our models as useful as possible, we also want to tie them together because a User will own many Notes.

Creating Model Associations

We're trying to create a one-to-many relationship because one User can have many Notes. To model this in our database, we will have the notes table contain a user_id column to point to the User that it belongs to. To tie database models together when using an ORM, we need to create a "relationship" on the User and a new column on the Note:

notes/models.py:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

```
    username = db.Column(db.String(80), unique=True, nullable=False)
    password = db.Column(db.String(200))
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(),
server_onupdate=db.func.now())
    notes = db.relationship('Note', backref='author', lazy=True)

class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200))
    body = db.Column(db.Text)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(),
server_onupdate=db.func.now())
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
```

Looking at the relationship set on the User, we specify that the items should use the Note class. The backref key says to add an attribute on the Note instance that points back to the User that owns it. The lazy attribute allows us to state that we don't want to fetch all of the notes every time we get fetch a User.

On the Note class, we're defining a column with a foreign key constraint on user.id. This means that the user_id value needs to map to a valid row in the user table.

Registering our Database with Our Application

With the data modeled, we need to use our db within our application factory. From within the function, we'll need to import db, initialize it with the configuration that the application has read in, and finally, utilize Flask-Migrate to run database migrations for us. Migrate will allow us to iterate on our database schema easily.

notes/__init__.py:

```
import os

from flask import Flask
from flask_migrate import Migrate

def create_app(test_config=None):
    app = Flask(__name__)
    app.config.from_mapping(
        SECRET_KEY=os.environ.get('SECRET_KEY', default='dev'),
    )

    if test_config is None:
        app.config.from_pyfile('config.py', silent=True)
    else:
        app.config.from_mapping(test_config)
```

```
    from .models import db

    db.init_app(app)
    migrate = Migrate(app, db)

    return app
```

With our application factory ready to work, let's create our migrations using some of the additional commands added to the Flask CLI by Flask-Migrate. We'll initialize a migrations directory with one command and have Flask-Migrate create our initial migration based on our current models' structures.

```
(notes) $ flask db init
  Creating directory /home/cloud_user/projects/notes/migrations ... done
  Creating directory /home/cloud_user/projects/notes/migrations/versions ... done
  Generating /home/cloud_user/projects/notes/migrations/README ... done
  Generating /home/cloud_user/projects/notes/migrations/alembic.ini ... done
  Generating /home/cloud_user/projects/notes/migrations/env.py ... done
  Generating /home/cloud_user/projects/notes/migrations/script.py.mako ... done
  Please edit configuration/connection/logging settings in '/home/cloud_user/projects/notes/migrations/alembic.ini' before proceeding.
(notes) $ flask db migrate
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added table 'note'
  Generating /home/cloud_user/projects/notes/migrations/versions/da4d0c35ae3f_.py ... done
(notes) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> da4d0c35ae3f, empty message
```

Now our database has note and user tables. We've successfully modeled our information and we're ready to move onto adding web functionality for people to register and log in and also to create their own notes.

Introduction to Python Development

CHAPTER 11.5
Building User Registration

Being able to connect to the database allows us to persist data made by our requests. But we first need to start adding some routes and view functions in order to add functionality to our application. In this lesson, we'll add the routes and views that will let a person to create an account, log in, and log out.

Documentation For This Video
Flask 'Blueprints and Views' Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/views/)
Flask 'Templates' Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/templates/)
Werkzeug (http://werkzeug.pocoo.org/)
Course's GitHub Repository (branch: use-case-web-app) (https://github.com/linuxacademy/content-intro-to-python-development.git)

Creating Our First View
Web applications work by routing requests to specific pieces of code on the server, based on the combination of HTTP method and the path. These combinations are often called "routes." In Flask, the code that is run for a given route is called its "view." For the time being, we're going to define all of our views within our __init__.py file. Let's start by defining a function inside of our create_app function to handle a GET request to the /sign_up path. We'll define our view functions after our migrate = Migrate(app, db) line, but before we run return app.

notes/__init__.py:

```
import os

from flask import Flask, render_template
from flask_migrate import Migrate

def create_app(test_config=None):
    # previous code omitted
    db.init_app(app)
    migrate = Migrate(app, db)

    @app.route('/sign_up')
    def sign_up():
        return render_template('sign_up.html')
    return app
```

To assign a view function to a route, we'll use the app.route method

to decorate the function. This allows the app that we return to run this function, even though we didn't add it to the class itself. The app will run the function when the app receives a request to the /sign_up path. From within the view, we want to render some HTML to the page with a form for the user to fill out in order to sign up for an account. To render this HTML, we'll import the render_template function from Flask and then pass it the name of our template (that doesn't exist yet). If we navigate to our server's public IP address on port 3000, with the path of /sign_up, we should now see that there's a jinja2.exceptions.TemplateNotFound being raised. This shows that our function is being run, and now we just need to create this template.

The Sign Up Template

By default, Flask will try to search for template files in the templates directory and serve static assets from the static directory (CSS, JavaScript, images, etc.). We'll access these files by cloning the content repository for this course into a temporary directory, changing the branch, and then moving the starter_templates and starter_static directories:

```
(notes) $ cd /tmp
(notes) $ git clone https://github.com/linuxacademy/content-intro-to-python-development.git
...
(notes) $ cd content-intro-to-python-development
(notes) $ git checkout use-case-web-app
...
(notes) $ cp -R starter_templates ~/projects/notes/templates
(notes) $ cp -R starter_static ~/projects/notes/static
```

It's beyond the scope of this course to teach HTML and CSS, but we'll cover how Jinja is used for adding dynamic content to HTML files. We've pre-created some templates to download, and we can go through them to understand what's going on. Let's take a look at templates/base.html, which holds onto the layout for our web application and code that needs to be on most pages:

notes/templates/base.html:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Markdown Notes</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='bulma.min.css') }}">
    <link rel="stylesheet" href="{{ url_for('static', filename='highlight.min.css') }}">
```

```
    <link rel="stylesheet" href="{{ url_for('static',
filename='styles.css') }}">
  </head>
  </head>
  <body>
    <nav class="navbar" role="navigation" aria-label="main
navigation">
      <div class="container">
        <div class="navbar-brand">
          <a class="navbar-item" href="#">Notes</a>

          <div class="navbar-burger" data-target="navMenu">
            <span></span>
            <span></span>
            <span></span>
          </div>
        </div>
        <div class="navbar-menu" id="navMenu">
          <div class="navbar-end">
            {% if g.user %}
              <a class="navbar-item" href="#" aria-label="New Note">
                <i class="far fa-plus-square"></i>
              </a>
              <a class="navbar-item"  href="#">
                Log Out
              </a>
            {% else %}
              <a class="navbar-item" href="#">
                Log In
              </a>
              <a class="navbar-item" href="{{ url_for('sign_up') }}">
                Sign Up
              </a>
            {% endif %}
          </div>
        </div>
      </div>
    </nav>
    {% if get_flashed_messages() %}
      <div class="container">
        {% for category, message in
get_flashed_messages(with_categories=True) %}
          <div id="message{{ loop.index }}">
            {% if category %}
              {% if category == 'error' %}
                <div class="notification is-danger">
              {% elif category == 'warning' %}
                <div class="notification is-warning">
              {% elif category == 'success' %}
                <div class="notification is-success">
```

```
              {% else %}
                <div class="notification">
              {% endif %}
            {% else %}
              <div class="notification">
            {% endif %}
              <button class="delete" data-
target="message{{ loop.index }}" aria-label="delete"></button>
              {{ message }}
            </div>
          </div>
        {% endfor %}
      </div>
    {% endif %}
    <section class="section">
      <div class="container">
        {% block content %}
        {% endblock %}
      </div>
    </section>
    <script defer src="https://use.fontawesome.com/releases/v5.0.6/js/
all.js"></script>
    <script src="{{ url_for('static',
filename='highlight.min.js') }}"></script>
    <script src="{{ url_for('static', filename='app.js') }}"></script>
  </body>
</html>
```

There's quite a bit going on here, but the big things to notice are:

We occasionally use blocks that start with {% and end with %}. These blocks allow us to define sections that other templates can fill, and to add control flow logic to our views such as looping and conditional logic.
When we need to put values from Python into the HTML, we use blocks that start with {{ and end with }}.
Flask templates automatically have access to some functions and objects such as get_flashed_messages, url_for, and the g object.
From each of our views, we can specify that we "extend" this base view and fill in any blocks of the template that we'd like. We do this in our templates/sign_up.html:

notes/templates/sign_up.html*:

```
{% extends "base.html" %}

{% block content %}

<div class="columns is-desktop">
  <div class="column"></div>
  <div class="column is-half-desktop">
```

```
    <h2 class="is-size-3">Sign Up</h2>

    <form method="post" action="{{ url_for('sign_up') }}">
      <div class="field">
        <label class="label" for="username">Username</label>
        <div class="control">
          <input name="username" type="input" class="input"
required></input>
        </div>
      </div>

      <div class="field">
        <label class="label" for="password">Password</label>
        <div class="control">
          <input name="password" type="password" class="input"
required></input>
        </div>
      </div>

      <div class="field is-grouped">
        <div class="control">
          <input type="submit" value="Sign Up" class="button is-
link" />
        </div>
        <div class="control">
          <a href="#" class="button is-text">
            Already have an account? Log In.
          </a>
        </div>
      </div>
    </form>
  </div>
  <div class="column"></div>
</div>

{% endblock %}
```

Now when we visit the /sign_up path we will actually see a form.

Handling Form Submission

When we fill out our sign up form and submit it right now we're met with a 405 error stating "Method Not Allowed". This is because the form submission is making a POST request to the /sign_up path, but our route by default only specifies that it handles GET requests. We're going to have the same function handle both. Depending on the request, type we'll either render the form, create a user and log them in, or hit an error when creating the user and re-render the form with those errors displayed. Here's what our finally sign_up view looks like:

notes/__init__.py:

```
import os

from flask import Flask, render_template, redirect, url_for, request,
flash
from flask_migrate import Migrate
from werkzeug.security import generate_password_hash

def create_app(test_config=None):
    # Initial setup omitted

    from .models import db, User

    db.init_app(app)
    migrate = Migrate(app, db)

    @app.route('/sign_up', methods=('GET', 'POST'))
    def sign_up():
        if request.method == 'POST':
            username = request.form['username']
            password = request.form['password']
            error = None

            if not username:
                error = 'Username is required.'
            elif not password:
                error = 'Password is required.'
            elif User.query.filter_by(username=username).first():
                error = 'Username is already taken.'

            if error is None:
                user = User(username=username, password=password)
                db.session.add(user)
                db.session.commit()
                flash("Successfully signed up! Please log in.",
'success')
                return redirect(url_for('log_in'))

            flash(error, 'error')

        return render_template('sign_up.html')

    @app.route('/log_in')
    def log_in():
        return "Login"

    return app
```

We did quite a bit here! First, we had to import a few more helper functions from Flask. Here's what each of them does:

redirect - Allows us to do an HTTP redirect so that the browser is

redirected to a different URL
url_for - The same helper that we're using in the template files, allowing us to get the route for a given view function
request - Gives us access to the request information that the server received to trigger the view:
In this case, we get access to the method to know if the form was submitted, and the form attribute to see what the user submitted.
flash - Allows us to populate a list of messages to display based on what happened in the view:
Our base template renders these with a variety of styles by using the get_flashed_messages template function.
werkzeug.security.generate_password_hash - Werkzeug is a WSGI helper library that Flask uses behind the scenes. It provides some useful utilities. In this case, we're loading the generate_password_hash function from the security module, so that we're storing an encrypted version of the password for the User instead of the plain text version.
Once we've imported all of these new functions, we need to also pull in our User class so that we can create a new instance and store it assuming that the form was filled out correctly. We add some conditional validation logic to check for potential errors. Take note of the third elif where we check to see if there is already a user with the given username. If there are no issues, then we finally get to create an instance of User (with a hashed password) and store it in the database by using db.session.add(user) and then db.session.commit(). No SQL statements are run until we run db.session.commit().

Lastly, we redirect the user to /log_in so that they can attempt to sign in. We did need to create the log_in view and route so that url_for won't raise an error for a missing route, but we just have it return a string for now. Go ahead and create a user, navigate back to /sign_up, and try to create the same user.

We'll continue with user authentication in the next lecture.

CHAPTER 11.6
Building User Authentication

Continuing where we left off in the previous lesson, we'll be adding user authentication to our application now that we have the ability to create a User.

Documentation For This Video
Flask 'Blueprints and Views' Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/views/)
Flask 'Templates' Tutorial (http://flask.pocoo.org/docs/1.0/tutorial/templates/)
Werkzeug (http://werkzeug.pocoo.org/)
Course's GitHub Repository (branch: use-case-web-app) (https://github.com/linuxacademy/content-intro-to-python-development.git)
Logging In
With our user in the database, we're ready to log in. Let's start by creating the view. We're going to duplicate the sign_up.html as log_in.html and then modify a few things:

templates/log_in.html:

```
{% extends "base.html" %}

{% block content %}

<div class="columns is-desktop">
  <div class="column"></div>
  <div class="column is-half-desktop">
    <h2 class="is-size-3">Log In</h2>

    <form method="post" action="{{ url_for('log_in') }}">
      <div class="field">
        <label class="label" for="username">Username</label>
        <div class="control">
          <input name="username" type="input" class="input"
required></input>
        </div>
      </div>

      <div class="field">
        <label class="label" for="password">Password</label>
        <div class="control">
          <input name="password" type="password" class="input"
required></input>
        </div>
      </div>
```

```
      <div class="field is-grouped">
        <div class="control">
          <input type="submit" value="Log In" class="button is-link" /
>
        </div>
        <div class="control">
          <a href="{{ url_for('sign_up') }}" class="button is-text">
            Don't have an account? Sign Up.
          </a>
        </div>
      </div>
    </form>
  </div>
  <div class="column"></div>
</div>

{% endblock %}
```

Besides changing some of the wording around and linking back to the sign-up page, we really only needed to change the action of our form. We could actually leave this off and the browser would automatically submit the form to the same URL, but I like to be explicit. Make sure to also go back into the sign_up.html form and add a url_for('log_in') there for the "Already have an account?" link.

Next, let's implement the log_in view:

```
import os

from flask import Flask, render_template, redirect, url_for, request,
session, flash
from flask_migrate import Migrate
from werkzeug.security import generate_password_hash,
check_password_hash

def create_app(test_config=None):
    # Previous code omitted

    @app.route('/log_in', methods=('GET', 'POST'))
    def log_in():
        if request.method == 'POST':
            username = request.form['username']
            password = request.form['password']
            error = None

            user = User.query.filter_by(username=username).first()

            if not user or not check_password_hash(user.password,
password):
                error = 'Username or password are incorrect'
```

```
            if error is None:
                session.clear()
                session['user_id'] = user.id
                return redirect(url_for('index'))

            flash(error, category='error')
        return render_template('log_in.html')

    @app.route('/')
    def index():
        return 'Index'

    return app
```

We needed to import check_password_hash, the sister function to generate_password_hash, so that we could verify that the password passed in by the user and the hashed password in the database match. Additionally, we need to import session so that we can store information about the user between requests. The session object is a dictionary that our Flask app stores as a cookie in the user's browser. To validate the login attempt, we do two things:

Find the user in the database based on the username. If we can't find the user then something is wrong.
Use check_password_hash to ensure that user.password is the hashed version of the password that the user submitted.
We write this as a compound conditional because we want to show the same message if something goes wrong. There's no need for us to be more specific because that just makes it easier for people to check to see if certain usernames exist in our system.

Finally, if there is not an error. We clear any existing session that might exist, and then add the user_id so that we can look up the user based on the session later.

Logging Out
Compared to signing up or logging in, logging out is pretty simple. We just need to remove our user information from the session. The /log_out route should work for a GET request, but also from a DELETE request because we're "deleting" the session. Let's add this route and view now:

```
# imports omitted

def create_app(test_config=None):
    # Previous code omitted

    @app.route('/log_out', methods=('GET', 'DELETE'))
    def log_out():
        session.clear()
```

```
        flash('Successfully logged out.', 'success')
        return redirect(url_for('log_in'))

    return app
```

Now if we log in and then navigate to /log_out we should be redirected back to the /log_in path with a success message showing.

Adding sign up and authentication to our application has forced us to learn a lot about how Flask views, templates, and sessions work. From here, we're ready to add the main feature to our application: note taking!

Introduction to Python Development

CHAPTER 11.7
Implementing Notes CRUD - Creating and Reading

One of the most common things to implement in any web application is the CRUD functionality around
a specific type of data: Create, Read, Update, and Delete. In this lesson, we'll implement the C and R portions of CRUD for our Note class.

Documentation For This Video
Flask-SQLAlchemy Select, Insert, Delete (http://flask-sqlalchemy.pocoo.org/2.3/queries/#select-insert-delete)
The functools Module (https://docs.python.org/3.7/library/functools.html#functools.wraps)
The mistune Package (https://github.com/lepture/mistune)
Jinja Automatic Escaping (http://jinja.pocoo.org/docs/dev/templates/#working-with-automatic-escaping)

Populating user Based on the Session
Before we create our first note, we need to add some code to our application to make working with a logged in user nicer:

We want to automatically populate g.user if there is user information in the session.
We want to create a decorator that can wrap views that require a logged in User to work.
For the first one, we'll use a different decorator on app to state that the specified function should run before the request is processed by a view, using app.before_request. To handle the second, we'll use functools.wraps to create a decorator that redirects to /log_in if g.user is not set. Let's implement these now:

notes/__init__.py:

```
import os
import functools

from flask import Flask, render_template, redirect, url_for, request, session, flash, g
from flask_migrate import Migrate
from werkzeug.security import generate_password_hash, check_password_hash

def create_app(test_config=None):
    # Ommit initial setup

    db.init_app(app)
```

```
    migrate = Migrate(app, db)

    def require_login(view):
        @functools.wraps(view)
        def wrapped_view(**kwargs):
            if not g.user:
                return redirect(url_for('log_in'))
            return view(**kwargs)
        return wrapped_view

    @app.before_request
    def load_user():
        user_id = session.get('user_id')
        if user_id:
            g.user = User.query.get(user_id)
        else:
            g.user = None

    # Remaining code omitted
```

We needed to import both functools and the g object from Flask itself. From there, we created the require_login function that does a few things:

Takes in a view to decorate
Defines the wrapped_view that decorates the view, redirecting if there is no user and calling the original view if there is
The used of functools.wraps allows the wrapped_view that we return to copy some of the contents that are implicitly designated for a function, such as __name__. So the wrapped_view.__name__ == view.__name__ would be true.

The load_user function doesn't do anything but assign g.user if there is a user_id in the session. Without adding any more code, we can actually see changes to our navigation now.

Creating a Note
We need to create a few different routes for our Notes CRUD:

/notes - The notes index that will list out the currently logged in User's notes
/notes/new - The note creation page that will have a form for creating a new note
/notes/<note_id>/delete - This route will be used to delete a note by handling using either a GET or DELETE method. If we receive that request, then we'll delete the note with the given note_id. The most accurate HTTP method to use is DELETE, but browsers mostly utilize GET and POST so we'll support GET to avoid writing JavaScript code to execute our request.
/notes/<note_id>/edit - The note edit page to allow us to make modifications to an existing note

To limit the amount of time that we need to spend writing HTML, we can copy over the contents of /tmp/contents-intro-to-python-development/note_templates into our project's templates directory:

```
(notes) $ cp /tmp/content-intro-to-python-development/note_templates/* ~/projects/notes/templates/
```

We can't really do anything else in our system before we have notes to render, so we'll start by adding some skeleton views and then moving from there:

notes/__init__.py:

```
# Imports omitted

def create_app(test_config=None):
    # Initial setup omitted

    from .models import db, User, Note

    db.init_app(app)
    migrate = Migrate(app, db)

    # Earlier views omitted

    @app.route('/notes')
    @require_login
    def note_index():
        return 'Note Index'

    @app.route('/notes/new', methods=('GET', 'POST'))
    @require_login
    def note_create():
        if request.method == 'POST':
            title = request.form['title']
            body = request.form['body']
            error = None

            if not title:
                error = 'Title is required.'

            if not error:
                note = Note(author=g.user, title=title, body=body)
                db.session.add(note)
                db.session.commit()
                flash(f"Successfully created note: '{title}'", 'success')
                return redirect(url_for('note_index'))

            flash(error, 'error')
```

```
        return render_template('note_create.html')

    return app
```

When we're dealing with the 'C' in CRUD it usually goes the same way. We check if the request is a POST request, validate the information, create the object, and then redirect OR render the original page with errors to display.

Looking at our note_create.html template, the only new thing is that we're now setting the value for our inputs to match the corresponding item in request.form. If there is an error, we'll re-render the page with the content that the user created.

notes/templates/note_create.html:

```
{% extends 'base.html' %}

{% block content %}

  <h1 class="is-size-3">New Note</h1>

  <form action="{{ url_for('note_create') }}" method="post">
    <div class="field">
      <label class="label" for="title">Title</label>
      <div class="control">
        <input name="title" value="{{ request.form['title'] }}"
class="input"></input>
      </div>
    </div>

    <div class="field">
      <label class="label" for="body">Body (Supports Markdown)</label>
      <div class="control">
        <textarea name="body" class="textarea has-text-
monospaced">{{ request.form['body'] }}</textarea>
      </div>
    </div>

    <div class="field is-grouped">
      <div class="control">
        <input type="submit" value="Create Note" class="button is-
primary" />
      </div>
      <div class="control">
        <a href="{{ url_for('note_index') }}" class="button is-
text">Cancel</a>
      </div>
    </div>
  </form>
```

```
{% endblock %}
```
Now that we have a note, we can implement the 'R' portion of CRUD by implementing the note index page and the note show page.

Parsing Markdown

Parsing markdown and displaying it is something that's outside the scope of this course, but this is a great situation where we can rely on the open source community and pull in another dependency. We're going to use on the mistune markdown parsing library, so let's install that now:

```
(notes) $ pipenv install mistune
...
```
There are a few ways that we can use this, but in general, we need to use a mistune.Markdown object or the mistune.markdown function to take the text that we store in Note.body. We'll add a new calculated property to our Note class called body_html. Here's what it looks like:

notes/models.py:

```
from flask_sqlalchemy import SQLAlchemy
from mistune import markdown

# db and User omitted

class Note(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200))
    body = db.Column(db.Text)
    created_at = db.Column(db.DateTime, server_default=db.func.now())
    updated_at = db.Column(db.DateTime, server_default=db.func.now(),
server_onupdate=db.func.now())
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)

    @property
    def body_html(self):
        return markdown(self.body)
```
Now we're able to use note.body_html in our templates to render out the HTML.

Rendering Notes

Now that we can convert the Note.body into HTML, let's implement the note_index view so that we can see our notes from the UI.

notes/__init__.py:

```
# Imports omitted
```

```
def create_app(test_config=None):
    # Earlier code omitted

    @app.route('/notes')
    @require_login
    def note_index():
        return render_template('note_index.html', notes=g.user.notes)

    # Remaining views omitted

    return app
```

Because we're using require_login, we know that we'll never make it into the note_index view if there is not a g.user object. From there, we're able to simply pass the currently logged in user's notes to the template as the variable "notes", by using the corresponding keyword argument in the template we're rendering.

Let's take a look at the template that will render these notes:

notes/templates/note_index.html:

```
{% extends 'base.html' %}

{% block content %}
  {% if not notes %}
    <div class="content">
      <p>You haven't created any notes! <a
href="{{ url_for('note_create') }}">Create your first note.</a>
    </div>
  {% endif %}

  {% for note in notes %}
    <article class="message">
      <div class="message-header">
        <p>{{ note.title }}</p>
        <div>
          <a class="button is-primary is-small has-text-weight-bold"
href="#">
            Edit Note
          </a>
          <a class="button is-danger is-small has-text-weight-bold"
href="#">
            Delete Note
          </a>
        </div>
      </div>
      <div class="message-body content">
        {{ note.body_html|safe }}
      </div>
    </article>
```

```
  {% endfor %}
{% endblock %}
```

For the most part, we've seen everything in this template already. The exception being |safe when we're rendering out the note.body_html. This is a Jinja filter, and the safe filter will tell Jinja that it should not escape the value coming from Python. Instead, we've already declared it to be "safe" to render.

Now if we log in and head to /notes, we can see the notes that we've created. And if we used fenced code blocks, we'll even see them rendered with syntax highlighting, thanks to some JavaScript that we packaged with the static assets.

CHAPTER 11.8
Implementing Notes CRUD - Updating and Deleting

The last two CRUD features that we need to implement are "update" and "delete". In this lesson, we'll add views to allow a user to edit, update, and delete their own notes.

Documentation For This Video
Flask-SQLAlchemy Select, Insert, Delete (http://flask-sqlalchemy.pocoo.org/2.3/queries/#select-insert-delete)

Editing and Updating a Note
Editing and updating a note isn't much different from creating a note, but it does give us the ability to use dynamic routes that contain variables. We'll almost use the exact same template. Let's create our note_update view now:

notes/__init__.py:

```
# Imports omitted

def create_app(test_config=None):
    # Earlier code omitted

    db.init_app(app)
    migrate = Migrate(app, db)

    def require_login(view):
        @functools.wraps(view)
        def wrapped_view(**kwargs):
            if not g.user:
                return redirect(url_for('log_in'))
            return view(**kwargs)
        return wrapped_view

    @app.errorhandler(404)
    def page_not_found(e):
        return render_template('404.html'), 404

    # Earlier views omitted

    @app.route('/notes/<note_id>/edit')
    @require_login
    def note_update(note_id):
        note = Note.query.filter_by(user_id=g.user.id,
id=note_id).first_or_404()
        return render_template('note_update.html', note=note)
```

```
    return app
```

This is the first time that we've taken in user input in our routes, so we need to handle potential 404 errors if the object that the user is requesting doesn't exist, or they don't have access to it. In this case, we can search for a Note, with the proper user_id, and id using the first_or_404 feature of Flask-SQLAlchemy. By using this, if the object is not found, then we'll stop processing the current view and instead abort with a 404 error. To make that fit with our application, we created another view called page_not_found, and registered it as the errorhandler for 404 errors thrown by our application.

Let's take a look at the template that we're rendering now:

notes/templates/note_update.html:

```
{% extends 'base.html' %}

{% block content %}

  <h1 class="is-size-3">Edit Note: {{ note.title }}</h1>

  <form action="{{ url_for('note_update', note_id=note.id) }}"
method="PATCH">
    <div class="field">
      <label class="label" for="title">Title</label>
      <div class="control">
        <input name="title" value="{{ request.form['title'] or
note.title }}" class="input"></input>
      </div>
    </div>

    <div class="field">
      <label class="label" for="body">Body (Supports Markdown)</label>
      <div class="control">
        <textarea name="body" class="textarea has-text-
monospaced">{{ request.form['body'] or note.body }}</textarea>
      </div>
    </div>

    <div class="field is-grouped">
      <div class="control">
        <input type="submit" value="Update Note" class="button is-
primary" />
      </div>
      <div class="control">
        <a href="{{ url_for('note_index') }}" class="button is-
text">Cancel</a>
      </div>
    </div>
```

```
  </form>

{% endblock %}
```
This form is almost exactly like the creation form, but we needed to change the method to PATCH. We could potentially be doing a partial update to a note when we submit the form. The action also had to change, and this is a good opportunity to look at how url_for works when we have an argument in the URL. Lastly, we're setting the values of the title's input and the body's textarea to be either the value from the request or the existing value on the note variable. We're using this approach so that the user doesn't lose all of their edits if there is a validation error when we go to handle the update. Nobody wants to have to start over when there's a form error.

Let's go ahead and implement the update handling now:

notes/__init__.py:

```
# Imports omitted

def create_app(test_config=None):
    # Earlier code omitted

    @app.route('/notes/<note_id>/edit', methods=('GET', 'PATCH'))
    @require_login
    def note_update(note_id):
        note = Note.query.filter_by(user_id=g.user.id,
id=note_id).first_or_404()
        if request.method == 'PATCH':
            title = request.form['title']
            body = request.form['body']
            error = None

            if not title:
                error = 'Title is required.'

            if not error:
                note.title = title
                note.body = body
                db.session.add(note)
                db.session.commit()
                flash(f"Successfully updated note: '{title}'",
'success')
                return redirect(url_for('note_index'))

            flash(error, 'error')

        return render_template('note_update.html', note=note)

    return app
```

There's only one real big difference between between what we're doing in this view and in the note_create view. We're reassigning the title and body on an existing Note before adding it to the list of updates to make, instead of creating a whole new object. Now when we go to our edit page and submit the form with differences, we can see that it's just sending us back to the edit page without changing anything and our URL has all of the contents of our form. What happened?

While PATCH is the proper HTTP method to use when partially updating a model, it doesn't work in HTML form tags. In this case, we'll adjust the form to have a POST method instead and we'll change our view to handle GET, POST, and PATCH.

Note: Be sure to change the method in templates/note_update.html to POST first.

notes/__init__.py:

```
# Imports omitted

def create_app(test_config=None):
    # Earlier code omitted

    @app.route('/notes/<note_id>/edit', methods=('GET', 'POST',
'PATCH'))
    @require_login
    def note_update(note_id):
        note = Note.query.filter_by(user_id=g.user.id,
id=note_id).first_or_404()
        if request.method in ['POST', 'PATCH']:
            title = request.form['title']
            body = request.form['body']
            error = None

            if not title:
                error = 'Title is required.'

            if not error:
                note.title = title
                note.body = body
                db.session.add(note)
                db.session.commit()
                flash(f"Successfully updated note: '{title}'",
'success')
                return redirect(url_for('note_index'))

            flash(error, 'error')

        return render_template('note_update.html', note=note)
```

```
    return app
```

Now if we refresh the edit page so that the HTML changes and then we submit some changes we should see them happen as expected. We're supporting both POST and PATCH because it's the right thing to do.

Deleting a Note

The last thing that we're going to do is handle the deleting of a note. To do this, we're going to have a view for both GET and DELETE, without a template. We will utilize first_or_404 again, and then once we have the note we'll use db.session.delete(note) to get rid of it before redirecting back to the note index page. Let's add the code now:

notes/__init__.py:

```
# Imports omitted

def create_app(test_config=None):
    # Earlier code omitted

    @app.route('/notes/<note_id>/delete', methods=('GET', 'DELETE'))
    @require_login
    def note_delete(note_id):
        note = Note.query.filter_by(user_id=g.user.id,
id=note_id).first_or_404()
        db.session.delete(note)
        db.session.commit()
        flash(f"Successfully deleted note: '{note.title}'", 'success')
        return redirect(url_for('note_index'))

    return app
```

We should edit the delete links within note_index.html to use url_for('note_delete', note_id=note.id), and then we'll be able to delete a note from the note index page.

We've successfully implemented all of the CRUD functionality for our notes and now we have a working application. There's more we could do to improve security and add additional functionality, but this project has done a good job of showing us how web development in general works, and some of the things that we have to consider when writing web-based applications.

Additional Learning

Flask is a great micro-framework and was good for forcing us to explicitly implement the features that we needed. There are larger frameworks that will do more of the work for you such as Django and you can see this same application implemented with Django by heading here. One of the main other things that you might be interested in would be implementing web APIs using JSON. To serve up JSON from your application you'll want to use the jsonify function in Flask. It can

handle lists and dictionaries and will convert them into JSON before returning them. Head here to learn more about this feature of the framework.

Introduction to Python Development

What's Next?

Thank you for taking the time to go through this course! I hope that you learned a lot and I want to hear about it. If you could please take a moment to rate the course, it will help me know what is working and what is not. Now that you have a solid programming foundation with Python, there are a lot of doors open to you to continue your learning.

Here are some courses that I think you should consider taking that would leverage your knowledge of Python while teaching you something new:

Google App Engine Deep Dive (https://linuxacademy.com/cp/modules/view/id/167)
Google Kubernetes Engine Deep Dive (https://linuxacademy.com/cp/modules/view/id/249)
Cloud Functions Deep Dive (https://linuxacademy.com/cp/modules/view/id/186)
AWS Lambda Deep Dive (https://linuxacademy.com/cp/modules/view/id/204)

Remember that programming is a skill and it needs to be practiced so find problems to solve and push yourself. Let me know what you build in the community.