#### Linux Academy Tutorial : Python : Python Documentation For This Video

* [The argparse module](https://docs.python.org/3/library/argparse.html)
* [The argparse.ArgumentParser class](https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser)

**Building a CLI to Reverse Files**

The tool that we’re going to build in this video will need to do the following:

1. Require a filename argument, so it knows what file to read.
2. Print the contents of the file backward (bottom of the script first, each line printed backward)
3. Provide help text and documentation when it receives the --help flag.
4. Accept an optional --limit or -l flag to specify how many lines to read from the file.
5. Accept a --version flag to print out the current version of the tool.

This sounds like quite a bit, but thankfully the [argparse](https://docs.python.org/3/library/argparse.html) module will make doing most of this trivial. We’ll build this script up gradually as we learn what the [argparse.ArgumentParser](https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser) can do. Let’s start by building an ArgumentParser with our required argument:

*~/bin/reverse-file*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument('filename', help='the file to read')

args = parser.parse\_args()

print(args)

Here we've created an instance of ArgumentParser without any arguments. Next, we'll use the [add\_argument](https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument) method to specify a positional argument called filename and provide some help text using the help argument. Finally, we tell the parser to parse the arguments from stdin using the [parse\_args](https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.parse_args) method and stored off the parsed arguments as the variable args.

Let’s make our script executable and try this out without any arguments:

$ chmod u+x ~/bin/reverse-file

$ reverse-file

usage: reverse-file [-h] filename

reverse-file: error: the following arguments are required: filename

Since filename is required and wasn’t given the ArgumentParser object recognized the problem and returned a useful error message. That’s awesome! We can also see that it looks like it takes the -h flag already, let’s try that now:

$ reverse-file -h

usage: reverse-file [-h] filename

positional arguments:

filename the file to read

optional arguments:

-h, --help show this help message and exit

It looks like we’ve already handled our requirement to provide help text. The last thing we need to test out is what happens when we do provide a parameter for filename:

$ reverse-file testing.txt

Namespace(filename='testing.txt')

We can see here that args in our script is a [Namespace](https://docs.python.org/3/library/argparse.html#argparse.Namespace) object. This is a simple type of object that’s sole purpose is to hold onto named pieces of information from our ArgumentParser as attributes. The only attribute that we've asked it to hold onto is the filename attribute, and we can see that it set the value to 'testing.txt' since that’s what we passed in. To access these values in our code, we will chain off of our args object with a period:

>>> args.filename

'testing.txt'

**Adding Optional parameters**

We’ve already handled two of the five requirements we set for this script; let’s continue by adding the optional flags to our parser and then we’ll finish by implementing the real script logic. We need to add a --limit flag with a -l alias.

*~/bin/reverse-file*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Read a file in reverse')

parser.add\_argument('filename', help='the file to read')

parser.add\_argument('--limit', '-l', type=int, help='the number of lines to read')

args = parser.parse\_args()

print(args)

To specify that an argument is a flag, we need to place two hyphens at the beginning of the flag’s name. We’ve used the type option for add\_argument to state that we want the value converted to an integer, and we specified a shorter version of the flag as our second argument.

Here is what args now looks like:

$ reverse-file --limit 5 testing.txt

Namespace(filename='testing.txt', limit=5)

Next, we’ll add a --version flag. This one will be a little different because we’re going to use the action option to specify a string to print out when this flag is received:

*~/bin/reverse-file*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Read a file in reverse')

parser.add\_argument('filename', help='the file to read')

parser.add\_argument('--limit', '-l', type=int, help='the number of lines to read')

parser.add\_argument('--version', '-v', action='version', version='%(prog)s 1.0')

args = parser.parse\_args()

print(args)

This uses a built-in action type of [version](https://docs.python.org/3/library/argparse.html#action) which we’ve found in the documentation.

Here’s what we get when we test out the --version flag:

$ reverse-file --version

reverse-file 1.0

*Note:* Notice that it carried out the version action and didn’t continue going through the script.

#### Adding Our Business Logic

We finally get a chance to use our file IO knowledge in a script:

~/bin/reverse-file

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Read a file in reverse')

parser.add\_argument('filename', help='the file to read')

parser.add\_argument('--limit', '-l', type=int, help='the number of lines to read')

parser.add\_argument('--version', '-v', action='version', version='%(prog)s 1.0')

args = parser.parse\_args()

with open(args.filename) as f:

lines = f.readlines()

lines.reverse()

if args.limit:

lines = lines[:args.limit]

for line in lines:

print(line.strip()[::-1])

Here’s what we get when we test this out on the xmen\_base.txt file from our working with files video:

$ reverse-file xmen\_base.txt

gnihtemoS

reivaX rosseforP

relwarcthgiN

pohsiB

spolcyC

enirevloW

mrotS

~ $ reverse-file -l 2 xmen\_base.txt

gnihtemoS

reivaX rosseforP

#### Try /Catch: 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**

In our reverse-file script, what happens if the filename doesn’t exist? Let’s give it a shot:

$ reverse-file fake.txt

Traceback (most recent call last):

File "/home/user/bin/reverse-file", line 11, in

with open(args.filename) as f:

FileNotFoundError: [Errno 2] No such file or directory: 'fake.txt'

This FileNotFoundError is something that we can expect to happen quite often and our script should handle this situation. Our parser isn’t going to catch this because we’re technically using the CLI properly, so we need to handle this ourselves. To handle these errors we’re going to utilize the keywords [try](https://docs.python.org/3/reference/compound_stmts.html#the-try-statement), except, and else.

*~/bin/reverse-file*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Read a file in reverse')

parser.add\_argument('filename', help='the file to read')

parser.add\_argument('--limit', '-l', type=int, help='the number of lines to read')

parser.add\_argument('--version', '-v', action='version', version='%(prog)s verison 1.0')

args = parser.parse\_args()

try:

f = open(args.filename)

limit = args.limit

except FileNotFoundError as err:

print(f"Error: {err}")

else:

with f:

lines = f.readlines()

lines.reverse()

if limit:

lines = lines[:limit]

for line in lines:

print(line.strip()[::-1])

We utilize the try statement to denote that it’s quite possible for an error to happen within it. From there we can handle specific types of errors using the except keyword (we can have more than one). In the event that there isn’t an error, then we want to carry out the code that is in the else block. If we want to execute some code regardless of there being an error or not, we can put that in a finally block at the very end of our t, except for workflow.

Now when we try our script with a fake file, we get a much better response:

$ reverse-file fake.txt

Error: [Errno 2] No such file or directory: 'fake.txt'

#### Exit Status : Python Documentation For This Video

* [The sys module](https://docs.python.org/3/library/sys.html)
* [The sys.exit function](https://docs.python.org/3/library/sys.html#sys.exit)

**Adding Error Exit Status to reverse-file**

When our reverse-file script receives a file that doesn’t exist, we show an error message, but we don’t set the exit status to 1 to be indicative of an error.

$ reverse-file -l 2 fake.txt

Error: [Errno 2] No such file or directory: 'fake.txt'

~ $ echo $?

0

Let’s use the [sys.exit](https://docs.python.org/3/library/sys.html#sys.exit) function to accomplish this:

*~/bin/reverse-file*

#!/usr/bin/env python3.6

import argparse

import sys

parser = argparse.ArgumentParser(description='Read a file in reverse')

parser.add\_argument('filename', help='the file to read')

parser.add\_argument('--limit', '-l', type=int, help='the number of lines to read')

parser.add\_argument('--version', '-v', action='version', version='%(prog)s verison 1.0')

args = parser.parse\_args()

try:

f = open(args.filename)

limit = args.limit

except FileNotFoundError as err:

print(f"Error: {err}")

sys.exit(1)

else:

with f:

lines = f.readlines()

lines.reverse()

if limit:

lines = lines[:limit]

for line in lines:

print(line.strip()[::-1])

Now, if we try our script with a missing file, we will exit with the proper code:

$ reverse-file -l 2 fake.txt

Error: [Errno 2] No such file or directory: 'fake.txt'

$ echo $?

1

#### Executing Shell Command : Python Documentation For This Video

* [The subprocess module](https://docs.python.org/3/library/subprocess.html)
* [The subprocess.run function](https://docs.python.org/3/library/subprocess.html#subprocess.run)
* [The subprocess.CompletedProcess class](https://docs.python.org/3/library/subprocess.html#subprocess.CompletedProcess)
* [The subprocess.PIPE object](https://docs.python.org/3/library/subprocess.html#subprocess.PIPE)
* [The bytes type](https://docs.python.org/3/library/stdtypes.html#bytes)
* [The subprocess.CalledProcessError class](https://docs.python.org/3/library/subprocess.html#subprocess.CalledProcessError)

**Executing Shell Commands With subprocess.run**

For working with external processes, we’re going to experiment with the [subprocess](https://docs.python.org/3/library/subprocess.html) module from the REPL. The main function that we’re going to work with is the [subprocess.run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function, and it provides us with a lot of flexibility:

>>> import subprocess

>>> proc = subprocess.run(['ls', '-l'])

total 20

drwxrwxr-x. 2 user user 54 Jan 28 15:36 bin

drwxr-xr-x. 2 user user 6 Jan 7 2015 Desktop

-rw-rw-r--. 1 user user 44 Jan 26 22:16 new\_xmen.txt

-rw-rw-r--. 1 user user 98 Jan 26 21:39 read\_file.py

-rw-rw-r--. 1 user user 431 Aug 6 2015 VNCHOWTO

-rw-rw-r--. 1 user user 61 Jan 28 14:11 xmen\_base.txt

-rw-------. 1 user user 68 Mar 18 2016 xrdp-chansrv.log

>>> proc

CompletedProcess(args=['ls', '-l'], returncode=0)

Our proc variable is a [CompletedProcess](https://docs.python.org/3/library/subprocess.html#subprocess.CompletedProcess) object, and this provides us with a lot of flexibility. We have access to the returncode attribute on our proc variable to ensure that it succeeded and returned a 0 to us. Notice that the lscommand was executed and printed to the screen without us specifying to print anything. We can get around this by capturing STDOUT using a [subprocess.PIPE](https://docs.python.org/3/library/subprocess.html#subprocess.PIPE).

>>> proc = subprocess.run(

... ['ls', '-l'],

... stdout=subprocess.PIPE,

... stderr=subprocess.PIPE,

... )

>>> proc

CompletedProcess(args=['ls', '-l'], returncode=0, stdout=b'total 20\ndrwxrwxr-x. 2 user user 54 Jan 28 15:36 bin\ndrwxr-xr-x. 2 user user 6 Jan 7 2015 Desktop\n-rw-rw-r--. 1 user user 44 Jan 26 22:16 new\_xmen.txt\n-rw-rw-r--. 1 user user 98 Jan 26 21:39 read\_file.py\n-rw-rw-r--. 1 user user 431 Aug 6 2015 VNCHOWTO\n-rw-rw-r--. 1 user user 61 Jan 28 14:11 xmen\_base.txt\n-rw-------. 1 user user 68 Mar 18 2016 xrdp-chansrv.log\n', stderr=b'')

>>> proc.stdout

b'total 20\ndrwxrwxr-x. 2 user user 54 Jan 28 15:36 bin\ndrwxr-xr-x. 2 user user 6 Jan 7 2015 Desktop\n-rw-rw-r--. 1 user user 44 Jan 26 22:16 new\_xmen.txt\n-rw-rw-r--. 1 user user 98 Jan 26 21:39 read\_file.py\n-rw-rw-r--. 1 user user 431 Aug 6 2015 VNCHOWTO\n-rw-rw-r--. 1 user user 61 Jan 28 14:11 xmen\_base.txt\n-rw-------. 1 user user 68 Mar 18 2016 xrdp-chansrv.log\n'

Now that we’ve captured the output to attributes on our proc variable, we can work with it from within our script and determine whether or not it should ever be printed. Take a look at this string that is prefixed with a b character. It is because it is a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object and not a string. The bytes type can only contain ASCII characters and won’t do anything special with escape sequences when printed. If we want to utilize this value as a string, we need to explicitly convert it using the [bytes.decode](https://docs.python.org/3/library/stdtypes.html#bytes.decode) method.

>>> print(proc.stdout)

b'total 20\ndrwxrwxr-x. 2 user user 54 Jan 28 15:36 bin\ndrwxr-xr-x. 2 user user 6 Jan 7 2015 Desktop\n-rw-rw-r--. 1 user user 44 Jan 26 22:16 new\_xmen.txt\n-rw-rw-r--. 1 user user 98 Jan 26 21:39 read\_file.py\n-rw-rw-r--. 1 user user 431 Aug 6 2015 VNCHOWTO\n-rw-rw-r--. 1 user user 61 Jan 28 14:11 xmen\_base.txt\n-rw-------. 1 user user 68 Mar 18 2016 xrdp-chansrv.log\n'

>>> print(proc.stdout.decode())

total 20

drwxrwxr-x. 2 user user 54 Jan 28 15:36 bin

drwxr-xr-x. 2 user user 6 Jan 7 2015 Desktop

-rw-rw-r--. 1 user user 44 Jan 26 22:16 new\_xmen.txt

-rw-rw-r--. 1 user user 98 Jan 26 21:39 read\_file.py

-rw-rw-r--. 1 user user 431 Aug 6 2015 VNCHOWTO

-rw-rw-r--. 1 user user 61 Jan 28 14:11 xmen\_base.txt

-rw-------. 1 user user 68 Mar 18 2016 xrdp-chansrv.log

>>>

**Intentionally Raising Errors**

The [subprocess.run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function will not raise an error by default if you execute something that returns a non-zero exit status. Here’s an example of this:

>>> new\_proc = subprocess.run(['cat', 'fake.txt'])

cat: fake.txt: No such file or directory

>>> new\_proc

CompletedProcess(args=['cat', 'fake.txt'], returncode=1)

In this situation, we might want to raise an error, and if we pass the check argument to the function, it will raise a [subprocess.CalledProcessError](https://docs.python.org/3/library/subprocess.html#subprocess.CalledProcessError) if something goes wrong:

>>> error\_proc = subprocess.run(['cat', 'fake.txt'], check=True)

cat: fake.txt: No such file or directory

Traceback (most recent call last):

File "", line 1, in

File "/usr/local/lib/python3.6/subprocess.py", line 418, in run

output=stdout, stderr=stderr)

subprocess.CalledProcessError: Command '['cat', 'fake.txt']' returned non-zero exit status 1.

>>>

**Python 2 Compatible Functions**

If you’re interested in writing code with the [subprocess](https://docs.python.org/3/library/subprocess.html) module that will still work with Python 2, then you cannot use the [subprocess.run](https://docs.python.org/3/library/subprocess.html#subprocess.run) function because it’s only in Python 3. For this situation, you’ll want to look into using [subprocess.call](https://docs.python.org/3/library/subprocess.html#subprocess.call)and [subprocess.check\_output](https://docs.python.org/3/library/subprocess.html#subprocess.check_output).

1. Filter out items in that list
2. Modify every item in the list

For this, we can utilize [“list comprehensions”](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions).

**Python Documentation For This Video**

* [List Comprehensions](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions)

*Note:* we need the words file to exist at /usr/share/dict/words for this video. This can be installed via:

$ sudo yum install -y words

**Our contains Script**

To dig into [list comprehensions](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions), we’re going to write a script that takes a word that then returns all of the values in the “words” file on our machine that contain the word. Our first step will be writing the script using standard iteration, and then we’re going to refactor our script to utilize a [list comprehension](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions).

*~/bin/contains*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Search for words including partial word')

parser.add\_argument('snippet', help='partial (or complete) string to search for in words')

args = parser.parse\_args()

snippet = args.snippet.lower()

with open('/usr/share/dict/words') as f:

words = f.readlines()

matches = []

for word in words:

if snippet in word.lower():

matches.append(word)

print(matches)

Let’s test out our first draft of the script to make sure that it works:

$ chmod u+x bin/contains

$ contains Keith

['Keith\n', 'Keithley\n', 'Keithsburg\n', 'Keithville\n']

*Note:* Depending on your system’s words file your results may vary.

**Utilizing a List Comprehension**

This portion of our script is pretty standard:

*~/bin/contains* (partial)

words = open('/usr/share/dict/words').readlines()

matches = []

for word in words:

if snippet in word.lower():

matches.append(word)

print(matches)

We can rewrite that chunk of our script as one or two lines using a [list comprehension](https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions):

*~/bin/contains* (partial)

words = open('/usr/share/dict/words').readlines()

print([word for word in words if snippet in word.lower()])

We can take this even further by removing the '\n' from the end of each “word” we return:

*~/bin/contains* (partial)

words = open('/usr/share/dict/words').readlines()

print([word.strip() for word in words if snippet in word.lower()])

**Final Version**

Here’s the final version of our script that works (nearly) the same as our original version:

*~/bin/contains*

#!/usr/bin/env python3.6

import argparse

parser = argparse.ArgumentParser(description='Search for words including partial word')

parser.add\_argument('snippet', help='partial (or complete) string to search for in words')

args = parser.parse\_args()

snippet = args.snippet.lower()

words = open('/usr/share/dict/words').readlines()

print([word.strip() for word in words if snippet in word.lower()])

Here’s our output:

$ contains Keith

['Keith', 'Keithley', 'Keithsburg', 'Keithville']

#### Random and Json: Python Documentation For This Video

* [The random module](https://docs.python.org/3/library/random.html)
* [The json module](https://docs.python.org/3/library/json.html)
* [The range type](https://docs.python.org/3/library/stdtypes.html#range)

**Generating Random Test Data**

To write our receipt reconciliation tool, we need to have some receipts to work with as we’re testing out our implementation. We’re expecting receipts to be JSON files that contain some specific data and we’re going to write a script that will create some receipts for us.

We’re working on a system that requires some local paths, so let’s put what we’re doing in a receipts directory:

$ mkdir -p receipts/new

$ cd receipts

The receipts that haven’t been reconciled will go in the new directory, so we’ve already created that. Let’s create a gen\_receipts.py file to create some unreconciled receipts when we run it:

*~/receipts/gen\_receipts.py*

import random

import os

import json

count = int(os.getenv("FILE\_COUNT") or 100)

words = [word.strip() for word in open('/usr/share/dict/words').readlines()]

for identifier in range(count):

amount = random.uniform(1.0, 1000)

content = {

'topic': random.choice(words),

'value': "%.2f" % amount

}

with open(f'./new/receipt-{identifier}.json', 'w') as f:

json.dump(content, f)

We’re using the [json.dump](https://docs.python.org/3/library/json.html#json.dump) function to ensure that we’re writing out valid JSON (we’ll read it in later). [random.choice](https://docs.python.org/3/library/random.html#random.choice)allows us to select one item from an iterable (str, tuple, or list). The function [random.uniform](https://docs.python.org/3/library/random.html#random.uniform) gives us a float between the two bounds specified. This code does show us how to create a [range](https://docs.python.org/3/library/stdtypes.html#range), which takes a starting number and an ending number and can be iterated through the values between.

Now we can run our script using the python3.6 command:

$ FILE\_COUNT=10 python3.6 gen\_receipts.py

$ ls new/

receipt-0.json receipt-2.json receipt-4.json receipt-6.json receipt-8.json

receipt-1.json receipt-3.json receipt-5.json receipt-7.json receipt-9.json

$ cat new/receipt-0.json

{"topic": "microceratous", "value": "918.67"}

#### Shutil and Glob: Python Documentation For This Video

* [The os.mkdir function](https://docs.python.org/3/library/os.html#os.mkdir)
* [The shutil module](https://docs.python.org/3/library/shutil.html)
* [The glob module](https://docs.python.org/3/library/glob.html)
* [The json module](https://docs.python.org/3/library/json.html)

**Creating a Directory If It Doesn’t Exist**

Before we start doing anything with the receipts, we want to have a processed directory to move them to so that we don’t try to process the same receipt twice. Our script can be smart enough to create this directory for us if it doesn’t exist when we first run the script. We’ll use the [os.mkdir](https://docs.python.org/3/library/os.html#os.mkdir) function; if the directory already exists we can catch the OSError that is thrown:

*~/receipts/process\_receipts.py*

import os

try:

os.mkdir("./processed")

except OSError:

print("'processed' directory already exists")

**Collecting the Receipts to Process**

From the shell, we’re able to collect files based on patterns, and that’s useful. For our purposes, we want to get every receipt from the new directory that matches this pattern:

receipt-[0-9]\*.json

That pattern translates to receipt-, followed by any number of digits, and ending with a .json file extension. We can achieve this exact result using the [glob.glob](https://docs.python.org/3/library/glob.html#glob.glob) function.

*~/receipts/process\_receipts.py* (partial)

receipts = glob.glob('./new/receipt-[0-9]\*.json')

subtotal = 0.0

Part of processing the receipts will entail adding up all of the values, so we’re going to start our script with a subtotal of 0.0.

**Reading JSON, Totaling Values, and Moving Files**

The remainder of our script is going to require us to do the following:

1. Iterate over the receipts
2. Reading each receipt’s JSON
3. Totaling the value of the receipts
4. Moving each receipt file to the processed directory after we’re finished with it

We used the json.dump function to write out a JSON file, and we can use the opposite function [json.load](https://docs.python.org/3/library/json.html#json.load) to read a JSON file. The contents of the file will be turned into a dictionary that we can us to access its keys. We’ll add the value to the subtotal before finally moving the file using [shutil.move](https://docs.python.org/3/library/shutil.html#shutil.move). Here’s our final script:

*~/receipts/process\_receipts.py*

import glob

import os

import shutil

import json

try:

os.mkdir("./processed")

except OSError:

print("'processed' directory already exists")

# Get a list of receipts

receipts = glob.glob('./new/receipt-[0-9]\*.json')

subtotal = 0.0

for path in receipts:

with open(path) as f:

content = json.load(f)

subtotal += float(content['value'])

name = path.split('/')[-1]

destination = f"./processed/{name}"

shutil.move(path, destination)

print(f"moved '{path}' to '{destination}'")

print("Receipt subtotal: $%.2f" % subtotal)

Let’s add some files that don’t match our pattern to the new directory before running our script:

touch new/receipt-other.json new/receipt-14.txt new/random.txt

Finally, let’s run our script twice and see what we get:

$ python3.6 process\_receipts.py

moved './new/receipt-0.json' to './processed/receipt-0.json'

moved './new/receipt-1.json' to './processed/receipt-1.json'

moved './new/receipt-2.json' to './processed/receipt-2.json'

moved './new/receipt-3.json' to './processed/receipt-3.json'

moved './new/receipt-4.json' to './processed/receipt-4.json'

moved './new/receipt-5.json' to './processed/receipt-5.json'

moved './new/receipt-6.json' to './processed/receipt-6.json'

moved './new/receipt-7.json' to './processed/receipt-7.json'

moved './new/receipt-8.json' to './processed/receipt-8.json'

moved './new/receipt-9.json' to './processed/receipt-9.json'

Receipt subtotal: $6932.04

$ python3.6 process\_receipts.py

'processed' directory already exists

Receipt subtotal: $0.00

*Note:* The subtotal that is printed for you will be different since our receipts are all randomly generated.

**Python Documentation For This Video**

* [The re module](https://docs.python.org/3/library/re.html)
* [The math module](https://docs.python.org/3/library/math.html)

**More Specific Patterns Using Regular Expressions (The re Module)**

Occasionally, we need to be very specific about string patterns that we use, and sometimes those are just not doable with basic globbing. As an exercise in this, let’s change our process\_receipts.py file to only return even numbered files (regardless of length). Let’s generate some more receipts and try to accomplish this from the REPL:

$ FILE\_COUNT=20 python3.6 gen\_receipts.py

$ python3.6

>>> import glob

>>> receipts = glob.glob('./new/receipt-[0-9]\*[24680].json')

>>> receipts.sort()

>>> receipts

['./new/receipt-10.json', './new/receipt-12.json', './new/receipt-14.json', './new/receipt-16.json', './new/receipt-18.json']

That glob was pretty close, but it didn’t give us the single-digit even numbers. Let’s try now using the [re](https://docs.python.org/3/library/re.html) (**R**egular **E**xpression) module’s [match](https://docs.python.org/3/library/re.html#re.match) function, the [glob.iglob](https://docs.python.org/3/library/glob.html#glob.iglob) function, and a list comprehension:

>>> import re

>>> receipts = [f for f in glob.iglob('./new/receipt-[0-9]\*.json') if re.match('./new/receipt-[0-9]\*[02468].json', f)]

>>> receipts

['./new/receipt-0.json', './new/receipt-2.json', './new/receipt-4.json', './new/receipt-6.json', './new/receipt-8.json', './new/receipt-10.json', './new/receipt-12.json', './new/receipt-14.json', './new/receipt-16.json', './new/receipt-18.json']

We’re using the [glob.iglob](https://docs.python.org/3/library/glob.html#glob.iglob) function instead of the standard glob function because we knew we were going to iterate through it and make modifications at the same time. This iterator allows us to avoid fitting the whole expanded glob.globlist into memory at one time.

Regular Expressions are a pretty big topic, but once you’ve learned them, they are incredibly useful in scripts and also when working with tools like grep. The [re](https://docs.python.org/3/library/re.html) module gives us quite a few powerful ways to use regular expressions in our python code.

**Improved String Replacement**

One actual improvement that we can make to our process\_receipts.py file is that we can use a single function call to go from our path variable to the destination that we want. This section:

*~/receipts/process\_receipts.py* (partial)

name = path.split('/')[-1]

destination = f"./processed/{name}"

Becomes this using the [str.replace](https://docs.python.org/3/library/stdtypes.html#str.replace) method:

destination = path.replace('new', 'processed')

This is a useful refactoring to make because it makes the intention of our code more clear.

**Working With Numbers Using math**

Depending on how we want to process the values of our receipts, we might want to manipulate the numbers that we are working with by rounding; going to the next highest integer, or the next lowest integer. These sort of “rounding” actions are pretty common, and some of them require the [math](https://docs.python.org/3/library/math.html) module:

>>> import math

>>> math.ceil(1.1)

2

>>> math.floor(1.1)

1

>>> round(1.1111111111, 2)

1.11

We can utilize the built-in [round](https://docs.python.org/3/library/functions.html#round) function to clean up the printing of the subtotal at the end of the script. Here’s the final version of process\_receipts.py:

*~/receipts/process\_receipts.py*

import glob

import os

import shutil

import json

try:

os.mkdir("./processed")

except OSError:

print("'processed' directory already exists")

subtotal = 0.0

for path in glob.iglob('./new/receipt-[0-9]\*.json'):

with open(path) as f:

content = json.load(f)

subtotal += float(content['value'])

destination = path.replace('new', 'processed')

shutil.move(path, destination)

print(f"moved '{path}' to '{destination}'")

print(f"Receipt subtotal: ${round(subtotal, 2)}")

**BONUS: Truncate Float Without Rounding**

I mentioned in the video that you can do some more complicated math to print a number to a specified number of digits without rounding. Here’s an example a function that would do the truncation (for those curious):

>>> import math

>>> def ftruncate(f, ndigits=None):

... if ndigits and (ndigits > 0):

... multiplier = 10 \*\* ndigits

... num = math.floor(f \* multiplier) / multiplier

... else:

... num = math.floor(f)

... return num

>>> num = 1.5441020468646993

>>> ftruncate(num)

1

>>> ftruncate(num, 2)

1.54

>>> ftruncate(num, 8)

1.54410204

Creating Larger Project :  
  
In this last segment, we’re tackling a single, large problem over multiple videos. We’ll dig into development practices that we can utilize to ensure the success of our projects.

Our approach will include:

1. Project Planning
2. Documentation
3. Test Driven Development (TDD)

Through Test Driven Development, we’ll run into a wide variety of errors and establish a familiarity with the stack trace that will make debugging projects in the future easier.

**Links For This Video**

* [db\_setup.sh](https://raw.githubusercontent.com/linuxacademy/content-python3-sysadmin/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 Project**

We have many database servers that we manage, and we want to create a single tool that we can use to easily back up the databases to either AWS S3 or locally. We would like to be able to:

1. Specify the database URL to backup.
2. Specify a “driver” (local or s3)
3. Specify the backup “destination”. This will be a file path for local and a bucket name for s3.
4. Depending on the “driver”, create a local backup of the database or upload the backup to an S3 bucket.

**Setting up PostgreSQL Lab Server**

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 server to create and run our database. Create a “CentOS 7” cloud server and run the following on it:

$ curl -o db\_setup.sh https://raw.githubusercontent.com/linuxacademy/content-python3-sysadmin/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 9.6.6.

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-centos96-9.6-3.noarch.rpm

$ sudo yum install pgdg-centos96-9.6-3.noarch.rpm epel-release

$ sudo yum update

$ sudo yum install 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;"

With this prep work finished, we’re ready to start planning the project itself.

To start out our project, we’re going to set up our source control, our virtualenv, and finally start documenting how we want the project to work.

#### Creating the Repo and Virtualenv

Since we’re building a project that will likely be more than a single file, we’re going to create a full project complete with source control and dependencies. We’ll start by creating the directory to hold our project, and we’re going to place this in a code directory:

$ rm ~/requirements.txt

$ mkdir -p ~/code/pgbackup

$ cd ~/code/pgbackup

We’ve talked about pip and virtualenvs, and how they allow us to manage our dependency versions. For a development project, we will leverage a new tool to manage our project’s virtualenv and install dependencies. This tool is called [pipenv](https://docs.pipenv.org/). Let’s install [pipenv](https://docs.pipenv.org/) for our user and create a Python 3 virtualenv for our project:

$ pip3.6 install --user pipenv

$ pipenv --python $(which python3.6)

Rather than creating a requirements.txt file for us, pipenv has created a Pipfile that it will use to store virtualenv and dependency information. To activate our new virtualenv, we use the command pipenv shell, and to deactivate it we use exit instead of deactivate.

Next, let’s set up [git](https://git-scm.com/) as our source control management tool by initializing our repository. We’ll also add a .gitignore file [from GitHub](https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore) so that we don’t later track files that we don’t mean to.

$ git init

$ curl https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore -o .gitignore

#### Sketch out the README.rst

One great way to start planning out a project is to start by documenting it from the top level. This is the documentation that we would give to someone who wanted to know how to use the tool but didn’t care about creating the tool. This approach is sometimes called “README Driven Development”. Whenever we write documentation in a Python project, we should be using [reStructuredText](http://docutils.sourceforge.net/rst.html). We use this specific markup format because there are tools in the Python ecosystem that can read this text and render documentation in a standardized way. Here’s our READEME.rst file:

~/code/pgbackup/README.rst

pgbackup

========

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

Preparing for Development

-------------------------

1. Ensure ``pip`` and ``pipenv`` are installed

2. Clone repository: ``git clone git@github.com:example/pgbackup``

3. ``cd`` into repository

4. Fetch development dependencies ``make install``

5. Activate virtualenv: ``pipenv shell``

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

Running Tests

-------------

Run tests locally using ``make`` if virtualenv is active:

::

$ make

If virtualenv isn’t active then use:

::

$ pipenv run make

#### Our Initial Commit

Now that we’ve created our README.rst file to document what we plan on doing with this project, we’re in a good position to stage our changes and make our first commit:

$ git add --all .

$ git commit -m 'Initial commit'

To start out our project, we’re going to set up our source control, our virtualenv, and finally start documenting how we want the project to work.

#### Creating the Repo and Virtualenv

Since we’re building a project that will likely be more than a single file, we’re going to create a full project complete with source control and dependencies. We’ll start by creating the directory to hold our project, and we’re going to place this in a code directory:

$ rm ~/requirements.txt

$ mkdir -p ~/code/pgbackup

$ cd ~/code/pgbackup

We’ve talked about pip and virtualenvs, and how they allow us to manage our dependency versions. For a development project, we will leverage a new tool to manage our project’s virtualenv and install dependencies. This tool is called [pipenv](https://docs.pipenv.org/). Let’s install [pipenv](https://docs.pipenv.org/) for our user and create a Python 3 virtualenv for our project:

$ pip3.6 install --user pipenv

$ pipenv --python $(which python3.6)

Rather than creating a requirements.txt file for us, pipenv has created a Pipfile that it will use to store virtualenv and dependency information. To activate our new virtualenv, we use the command pipenv shell, and to deactivate it we use exit instead of deactivate.

Next, let’s set up [git](https://git-scm.com/) as our source control management tool by initializing our repository. We’ll also add a .gitignore file [from GitHub](https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore) so that we don’t later track files that we don’t mean to.

$ git init

$ curl https://raw.githubusercontent.com/github/gitignore/master/Python.gitignore -o .gitignore

#### Sketch out the README.rst

One great way to start planning out a project is to start by documenting it from the top level. This is the documentation that we would give to someone who wanted to know how to use the tool but didn’t care about creating the tool. This approach is sometimes called “README Driven Development”. Whenever we write documentation in a Python project, we should be using [reStructuredText](http://docutils.sourceforge.net/rst.html). We use this specific markup format because there are tools in the Python ecosystem that can read this text and render documentation in a standardized way. Here’s our READEME.rst file:

~/code/pgbackup/README.rst

pgbackup

========

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

Preparing for Development

-------------------------

1. Ensure ``pip`` and ``pipenv`` are installed

2. Clone repository: ``git clone git@github.com:example/pgbackup``

3. ``cd`` into repository

4. Fetch development dependencies ``make install``

5. Activate virtualenv: ``pipenv shell``

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

Running Tests

-------------

Run tests locally using ``make`` if virtualenv is active:

::

$ make

If virtualenv isn’t active then use:

::

$ pipenv run make

#### Our Initial Commit

Now that we’ve created our README.rst file to document what we plan on doing with this project, we’re in a good position to stage our changes and make our first commit:

$ git add --all .

$ git commit -m 'Initial commit'

With our project structured, we’re finally ready to start implementing the logic to create database backups. We’re going to tackle this project using “Test Driven Development”, so let’s learn the basics of TDD now.

**Documentation For This Video**

* [The pytest package](https://docs.pytest.org/en/latest/)
* [The pytest.raises function](https://docs.pytest.org/en/latest/assert.html#assertions-about-expected-exceptions)

**Installing pytest**

For this course, we’re using [pytest](https://docs.pytest.org/en/latest/) as our testing framework. It’s a simple tool, and although there is a unit testing framework built into Python, I think that pytest is a little easier to understand. Before we can use it though, we need to install it. We’ll use pipenv and specify that this is a “dev” dependency:

(pgbackup-E7nj\_BsO) $ pipenv install --dev pytest

...

Adding pytest to Pipfile's [dev-packages]…

Locking [dev-packages] dependencies…

Locking [packages] dependencies…

Updated Pipfile.lock (5c8539)!

Now the line that we wrote in our Makefile that utilized the pytest, CLI will work.

**Writing Our First Tests**

The first step of TDD is writing a failing test. In our case, we’re going to go ahead and write a few failing tests. Using [pytest](https://docs.pytest.org/en/latest/), our tests will be functions with names that start with test\_. As long as we name the functions properly, the test runner should find and run them.

We’re going to write three tests to start:

1. A test that shows that the CLI fails if no driver is specified.
2. A test that shows that the CLI fails if there is no destination value given.
3. A test that shows, given a driver and a destination, that the CLI’s returned Namespace has the proper values set.

At this point, we don’t even have any source code files, but that doesn’t mean that we can’t write code that demonstrates how we would like our modules to work. The module that we want is called cli, and it should have a create\_parserfunction that returns an ArgumentParser configured for our desired use.

Let’s write some tests that exercise cli.create\_parser and ensure that our ArgumentParser works as expected. The name of our test file is important; make sure that the file starts with test\_. This file will be called test\_cli.py.

*~/code/pgbackup/tests/test\_cli.py*

import pytest

from pgbackup import cli

url = "postgres://bob:password@example.com:5432/db\_one"

def test\_parser\_without\_driver():

"""

Without a specified driver the parser will exit

"""

with pytest.raises(SystemExit):

parser = cli.create\_parser()

parser.parse\_args([url])

def test\_parser\_with\_driver():

"""

The parser will exit if it receives a driver

without a destination

"""

parser = cli.create\_parser()

with pytest.raises(SystemExit):

parser.parse\_args([url, "--driver", "local"])

def test\_parser\_with\_driver\_and\_destination():

"""

The parser will not exit if it receives a driver

with a destination

"""

parser = cli.create\_parser()

args = parser.parse\_args([url, "--driver", "local", "/some/path"])

assert args.driver == "local"

assert args.destination == "/some/path"

**Running Tests**

Now that we’ve written a few tests, it’s time to run them. We’ve created our Makefile already, so let’s make sure our virtualenv is active and run them:

$ pipenv shell

(pgbackup-E7nj\_BsO) $ make

PYTHONPATH=./src pytest

======================================= test session starts =======================================

platform linux -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0

rootdir: /home/user/code/pgbackup, inifile:

collected 0 items / 1 errors

============================================= ERRORS ==============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ERROR collecting tests/test\_cli.py \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ImportError while importing test module '/home/user/code/pgbackup/tests/test\_cli.py'.

Hint: make sure your test modules/packages have valid Python names.

Traceback:

tests/test\_cli.py:3: in

from pgbackup import cli

E ImportError: cannot import name 'cli'

!!!!!!!!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 errors during collection !!!!!!!!!!!!!!!!!!!!!!!!!!!!!

===================================== 1 error in 0.11 seconds =====================================

make: \*\*\* [test] Error 2

We get an ImportError from our test file because there is no module in pgbackup named cli. This is awesome because it tells us what our next step is. We need to create that file.

We now have some breaking tests to help guide us to the implementation of our client module. Let’s follow the errors that we see to get our tests passing.

**Documentation For This Video**

* [The argparse package](https://docs.python.org/3/library/argparse.html)
* [The argparse.Action class](https://docs.python.org/3/library/argparse.html#argparse.Action)
* [The pytest package](https://docs.pytest.org/en/latest/)
* [The pytest.fixture function](https://docs.pytest.org/en/latest/fixture.html)
* [Python decorators](https://docs.python.org/3/glossary.html#term-decorator)

**Moving Through Failing Tests**

Our current test failure is from there not being a cli.py file within the src/pgbackup directory. Let’s do just enough to move onto the next error:

*(partial make output)*

(pgbackup-E7nj\_BsO) $ touch src/pgbackup/cli.py

(pgbackup-E7nj\_BsO) $ make

PYTHONPATH=./src pytest

======================================= test session starts =======================================

platform linux -- Python 3.6.4, pytest-3.3.2, py-1.5.2, pluggy-0.6.0

rootdir: /home/user/code/pgbackup, inifile:

collected 3 items

tests/test\_cli.py FFF [100%]

============================================ FAILURES =============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_parser\_without\_driver \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_parser\_without\_driver():

"""

Without a specified driver the parser will exit

"""

with pytest.raises(SystemExit):

> parser = cli.create\_parser()

E AttributeError: module 'pgbackup.cli' has no attribute 'create\_parser'

tests/test\_cli.py:12: AttributeError

...

Now we’re getting an AttributeError because there is not an attribute/function called create\_parser. Let’s implement a version of that function that creates an ArgumentParser that hasn’t been customized:

*~/code/pgbackup/src/pgbackup/cli.py*

from argparse import ArgumentParser

def create\_parser():

parser = ArgumentParser()

return parser

Once again, let’s run our tests:

*(partial make output)*

(pgbackup-E7nj\_BsO) $ make

...

self = ArgumentParser(prog='pytest', usage=None, description=None, formatter\_class=, conflict\_handler='error', add\_help=True)

status = 2

message = 'pytest: error: unrecognized arguments: postgres://bob:password@example.com:5432/db\_one --driver local /some/path\n'

def exit(self, status=0, message=None):

if message:

self.\_print\_message(message, \_sys.stderr)

> \_sys.exit(status)

E SystemExit: 2

/usr/local/lib/python3.6/argparse.py:2376: SystemExit

-------------------------------------- Captured stderr call ---------------------------------------

usage: pytest [-h]

pytest: error: unrecognized arguments: postgres://bob:password@example.com:5432/db\_one --driver local /some/path

=============================== 1 failed, 2 passed in 0.14 seconds ================================

Interestingly, two of the tests succeeded. Those two tests were the ones that expected there to be a SystemExit error. Our tests sent unexpected output to the parser (since it wasn’t configured to accept arguments), and that caused the parser to error. This demonstrates why it’s important to write tests that cover a wide variety of use cases. If we hadn’t implemented the third test to ensure that we get the expected output on success, then our test suite would be green!

**Creating Our First Class**

For this course, we haven’t created any custom classes because it’s not something that we’ll do all the time, but in the case of our CLI, we need to. Our idea of having a flag of --driver that takes two distinct values isn’t something that any existing [argparse.Action](https://docs.python.org/3/library/argparse.html#argparse.Action) can do. Because of this, we’re going to follow along with the documentation and implement our own custom DriverAction class. We can put our custom class in our cli.py file and use it in our [add\_argument](https://docs.python.org/3/library/argparse.html#argparse.ArgumentParser.add_argument) call.

*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

**Adding More Tests**

Our CLI is coming along, but we probably want to raise an error if the end-user tries to use a driver that we don’t understand. Let’s add a few more tests that do the following:

1. Ensure that you can’t use a driver that is unknown, like azure.
2. Ensure that the drivers for s3 and local don’t cause errors.

*test/test\_cli.py* (partial)

def test\_parser\_with\_unknown\_drivers():

"""

The parser will exit if the driver name is unknown.

"""

parser = cli.create\_parser()

with pytest.raises(SystemExit):

parser.parse\_args([url, "--driver", "azure", "destination"])

def test\_parser\_with\_known\_drivers():

"""

The parser will not exit if the driver name is known.

"""

parser = cli.create\_parser()

for driver in ['local', 's3']:

assert parser.parse\_args([url, "--driver", driver, "destination"])

**Adding Driver Type Validation**

Since we already have a custom DriverAction, we can feel free to customize this to make our CLI a little more intelligent. The only drivers that we are going to support (for now) are s3 and local, so let’s add some logic to our action to ensure that the driver given is one that we can work with:

known\_drivers = ['local', 's3']

class DriverAction(Action):

def \_\_call\_\_(self, parser, namespace, values, option\_string=None):

driver, destination = values

if driver.lower() not in known\_drivers:

parser.error("Unknown driver. Available drivers are 'local' & 's3'")

namespace.driver = driver.lower()

namespace.destination = destination

**Removing Test Duplication Using pytest.fixture**

Before we consider this unit of our application complete, we should consider cleaning up some of the duplication in our tests. We create the parser using create\_parser in every test but using [pytest.fixture](https://docs.pytest.org/en/latest/fixture.html) we can extract that into a separate function and inject the parser value into each test that needs it.

Here’s what our parser function will look like:

*tests/test\_cli.py* (partial)

import pytest

@pytest.fixture

def parser():

return cli.create\_parser()

We haven’t run into this yet, but the @pytest.fixture on top of our function definition is what’s known as a [“decorator”](https://docs.python.org/3/glossary.html#term-decorator). A [“decorator”](https://docs.python.org/3/glossary.html#term-decorator) is a function that returns a modified version of the function. We’ve seen that if we don’t use parentheses that our functions aren’t called, and because of that we’re able to pass functions into other functions as arguments. This particular decorator will register our function in the list of fixtures that can be injected into a pytest test. To inject our fixture, we will add an argument to our test function definition that has the same name as our fixture name, in this case, parser. Here’s the final test file:

*tests/test\_cli.py*

import pytest

from pgbackup import cli

url = "postgres://bob@example.com:5432/db\_one"

@pytest.fixture()

def parser():

return cli.create\_parser()

def test\_parser\_without\_driver(parser):

"""

Without a specified driver the parser will exit

"""

with pytest.raises(SystemExit):

parser.parse\_args([url])

def test\_parser\_with\_driver(parser):

"""

The parser will exit if it receives a driver

without a destination

"""

with pytest.raises(SystemExit):

parser.parse\_args([url, "--driver", "local"])

def test\_parser\_with\_driver\_and\_destination(parser):

"""

The parser will not exit if it receives a driver

with a destination

"""

args = parser.parse\_args([url, "--driver", "local", "/some/path"])

assert args.driver == "local"

assert args.destination == "/some/path"

def test\_parser\_with\_unknown\_drivers(parser):

"""

The parser will exit if the driver name is unknown.

"""

with pytest.raises(SystemExit):

parser.parse\_args([url, "--driver", "azure", "destination"])

def test\_parser\_with\_known\_drivers(parser):

"""

The parser will not exit if the driver name is known.

"""

for driver in ['local', 's3']:

assert parser.parse\_args([url, "--driver", driver, "destination"])

Now, all of our tests should pass, and we’re in a good spot to make a commit.

The simplest way that we can get all of the information that we need out of a PostgreSQL is to use the pg\_dump utility that Postgres itself provides. Since that code exists outside of our codebase, it’s not our job to ensure that the pg\_dump tool itself works, but we do need to write tests that can run without an actual Postgres server running. For this, we will need to “stub” our interaction with pg\_dump.

**Documentation For This Video**

* [The pytest-mock package](https://github.com/pytest-dev/pytest-mock/#usage)
* [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)

**Install pytest-mock**

Before we can learn how to use mocking in our tests, we need to install the [pytest-mock package](https://github.com/pytest-dev/pytest-mock/#usage). This will pull in a few packages for us, and mainly provide us with a mocker fixture that we can inject into our tests:

(pgbackup-E7nj\_BsO) $ pipenv install --dev pytest-mock

**Writing Tests With Mocking**

We’re going to put all of the Postgres related logic into its own module called pgdump, and we’re going to begin by writing our tests. We want this module to do the following:

1. Make a call out to pg\_dump using [subprocess.Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen).
2. Returns the subprocess that STDOUT can be read from.

We know how to use the [subprocess](https://docs.python.org/3/library/subprocess.html) module, but we haven’t used [subprocess.Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen) yet. Behind the scenes, the functions that we already know use [Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen), and wait for it to finish. We’re going to use this instead of run, because we want to continue running code instead of waiting, right until we need to write the contents of proc.stdout to a file or S3.

To ensure that our code runs the proper third-party utilities, we’re going to use [mocker.patch](http://www.voidspace.org.uk/python/mock/patch.html#mock.patch) on the subprocess.Popenconstructor. This will substitute in a different implementation that holds onto information like the number of times the function is called and with what arguments. Let’s see what this looks like in practice:

*tests/test\_pgdump.py*

import pytest

import subprocess

from pgbackup import pgdump

url = "postgres://bob:password@example.com:5432/db\_one"

def test\_dump\_calls\_pg\_dump(mocker):

"""

Utilize pg\_dump with the database URL

"""

mocker.patch('subprocess.Popen')

assert pgdump.dump(url)

subprocess.Popen.assert\_called\_with(['pg\_dump', url], stdout=subprocess.PIPE)

The arguments that we’re passing to assert\_called\_with will need to match what is being passed to subprocess.Popenwhen we exercise pgdump.dump(url).

We now have tests for our pgdump implementation, and we have a basic understanding of mocking. Let’s start following the errors to completion.

**Documentation For This Video**

* [The pytest-mock package](https://github.com/pytest-dev/pytest-mock/#usage)
* [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 mocker.patch function](http://www.voidspace.org.uk/python/mock/patch.html#mock.patch)
* [The pytest.raises function](https://docs.pytest.org/en/latest/assert.html#assertions-about-expected-exceptions)
* [The sys.exit function](https://docs.python.org/3/library/subprocess.html#sys.exit)

**Initial Implementation**

Our first error is from not having a src/pgbackup/pgdump.py file, so let’s be sure to create that. We can guess that we’ll also have an error for the missing function, so let’s skip ahead a little and implement that:

*src/pgbackup/pgdump.py*

import subprocess

def dump(url):

return subprocess.Popen(['pg\_dump', url], stdout=subprocess.PIPE)

This will get our tests to passing, but what happens when the pg\_dump utility isn’t installed?

**Adding Tests For Missing PostgreSQL Client**

Let’s add another test that tells our [subprocess.Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen) to raise an OSError instead of succeeding. This is the kind of error that we will receive if the end-user of our package doesn’t have the pg\_dump utility installed. To cause our stub to raise this error we need to set the side\_effect attribute when we call [mocker.patch](http://www.voidspace.org.uk/python/mock/patch.html#mock.patch). We’ll pass in an OSError to this attribute. Finally, we’ll want to exit with a status code of 1 if we catch this error and pass the error message through. That means we’ll need to use [pytest.raises](https://docs.pytest.org/en/latest/assert.html#assertions-about-expected-exceptions) again to ensure we receive a SystemExit error. Here’s what the final tests look like for our pgdump module:

*tests/test\_pgdump.py*

import pytest

import subprocess

from pgbackup import pgdump

url = "postgres://bob:password@example.com:5432/db\_one"

def test\_dump\_calls\_pg\_dump(mocker):

"""

Utilize pg\_dump with the database URL

"""

mocker.patch('subprocess.Popen')

assert pgdump.dump(url)

subprocess.Popen.assert\_called\_with(['pg\_dump', url], stdout=subprocess.PIPE)

def test\_dump\_handles\_oserror(mocker):

"""

pgdump.dump returns a reasonable error if pg\_dump isn't installed.

"""

mocker.patch('subprocess.Popen', side\_effect=OSError("no such file"))

with pytest.raises(SystemExit):

pgdump.dump(url)

**Implementing Error Handling**

Since we know that [subprocess.Popen](https://docs.python.org/3/library/subprocess.html#subprocess.Popen) can raise an OSError, we’re going to wrap that call in a try block, print the error message, and use [sys.exit](https://docs.python.org/3/library/subprocess.html#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)

**Manual Testing**

We can have a certain amount of confidence in our code because we’ve written tests that cover our expected cases, but since we used patching, we don’t know that it works. Let’s manually load our code into the python REPL to test it out:

(pgbackup-E7nj\_BsO) $ 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.

The last few pieces of logic that we need to implement pertain to how we store the database dump. We’ll have a strategy for storing locally and on AWS S3, and it makes sense to put both of these in the same module. Let’s use TDD to implement the local storage strategy of our storage module.

**Documentation For This Video**

* [The tempfile package](https://docs.python.org/3/library/tempfile.html)
* [The tempfile.TemporaryFile class](https://docs.python.org/3/library/tempfile.html#tempfile.TemporaryFile)
* [The tempfile.NamedTemporaryFile class](https://docs.python.org/3/library/tempfile.html#tempfile.NamedTemporaryFile)

**Writing Local File Tests**

Working with files is something that we already already know how to do, and local storage is no different. If we think about what our local storage driver needs to do, it really needs two things:

1. Take in one “readable” object and one, local, “writeable” object.
2. Write the contents of the “readable” object to the “writeable” object.

Notice that we didn’t say files, that’s because we don’t need our inputs to be file objects. They need to implement some of the same methods that a file does, like [read](https://docs.python.org/3/library/io.html#io.BufferedWriter.read) and [write](https://docs.python.org/3/library/io.html#io.BufferedWriter.write), but they don’t have to be file objects.

For our testing purposes, we can use the [tempfile](https://docs.python.org/3/library/tempfile.html) package to create a [TemporaryFile](https://docs.python.org/3/library/tempfile.html#tempfile.TemporaryFile) to act as our “readable” and another [NamedTemporaryFile](https://docs.python.org/3/library/tempfile.html#tempfile.NamedTemporaryFile) to act as our “writeable”. We’ll pass them both into our function, and assert after the fact that the contents of the “writeable” object match what was in the “readable” object:

*tests/test\_storage.py*

import tempfile

from pgbackup import storage

def test\_storing\_file\_locally():

"""

Writes content from one file-like to another

"""

infile = tempfile.TemporaryFile('r+b')

infile.write(b"Testing")

infile.seek(0)

outfile = tempfile.NamedTemporaryFile(delete=False)

storage.local(infile, outfile)

with open(outfile.name, 'rb') as f:

assert f.read() == b"Testing"

**Implement Local Storage**

The requirements we looked at before are close to what we need to do in the code. We want to call [close](https://docs.python.org/3/library/io.html#io.IOBase.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()

The last unit that we need to implement before we can combine all of our modules into our final tool is the storage strategy for AWS S3.

**Documentation For This Video**

* [The boto3 package](https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj)
* [The pytest-mock package](https://github.com/pytest-dev/pytest-mock/#usage)
* [The Mock class](https://docs.python.org/3/library/unittest.mock.html#unittest.mock.Mock)

**Installing boto3**

To interface with AWS (S3 specifically), we’re going to use the wonderful [boto3](https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj) package. We can install this to our virtualenv using pipenv:

(pgbackup-E7nj\_BsO) $ pipenv install boto3

**Configuring AWS Client**

The [boto3](https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj) package works off of the same configuration file that you can use with the official aws CLI. To get our configuration right, let’s leave our virtualenv and install the awscli package for our user. From there, we’ll use its configure command to set up our config file:

(pgbackup-E7nj\_BsO) $ exit

$ mkdir ~/.aws

$ pip3.6 install --user awscli

$ aws configure

$ exec $SHELL

The exec $SHELL portion reload the shell to ensure that the configuration changes are picked up. Before moving on, make sure to reactivate our development virtualenv:

$ pipenv shell

**Writing S3 test**

Following the approach that we’ve been using, let’s write tests for our S3 interaction. To limit the explicit dependencies that we have, we’re going to have the following parameters to our storage.s3 function:

* A client object that has an upload\_fileobj method.
* A boto3 client meets this requirement, but in testing, we can pass in a “mock” object that implements this method.
* A file-like object (responds to read).
* An S3 bucket name as a string.
* The name of the file to create in S3.

We need an infile for all of our tests, so let’s extract a fixture for that also.

*tests/test\_storage.py* (partial)

import tempfile

import pytest

from pgbackup import storage

@pytest.fixture

def infile():

infile = tempfile.TemporaryFile('r+b')

infile.write(b"Testing")

infile.seek(0)

return infile

# Local storage tests...

def test\_storing\_file\_on\_s3(mocker, infile):

"""

Writes content from one readable to S3

"""

client = mocker.Mock()

storage.s3(client,

infile,

"bucket",

"file-name")

client.upload\_fileobj.assert\_called\_with(

infile,

"bucket",

"file-name")

**Implementing S3 Strategy**

Our test gives a little too much information about how we’re going to implement our storage.s3 function, but it should be pretty simple for us to implement now:

*src/pgbackup/storage.py* (partial)

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-E7nj\_BsO) $ echo "UPLOADED" > example.txt

(pgbackup-E7nj\_BsO) $ PYTHONPATH=./src python

>>> import boto3

>>> from pgbackup import storage

>>> client = boto3.client('s3')

>>> infile = open('example.txt', 'rb')

>>> storage.s3(client, infile, 'pyscripting-db-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-E7nj\_BsO) $ rm example.txt

(pgbackup-E7nj\_BsO) $ git add .

(pgbackup-E7nj\_BsO) $ git commit -m 'Implement S3 interactions'

We’ve successfully written the following:

* CLI parsing
* Postgres Interaction
* Local storage driver
* AWS S3 storage driver

Now we need to wire up an executable that can integrate these parts. Up to this point we’ve used TDD to write our code. These have been “unit tests” because we’re only ever testing a single unit of code. If we wanted to write tests that ensure our application worked from start to finish, we could do that and they would be “integration” tests. Given that our code does a lot with the network, and we would have to do a lot of mocking to write integration tests, we’re not going to write them. Sometimes the tests aren’t worth the work that goes into them.

**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)

**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 made 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)

install\_requires=['boto3'],

entry\_points={

'console\_scripts': [

'pgbackup=pgbackup.cli:main',

],

}

Notices 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 in the cli module, and it needs to do the following:

1. Import the [boto3](https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Client.upload_fileobj) package.
2. Import our pgdump and storage modules.
3. Create a parser and parse the arguments.
4. Fetch the database dump.
5. Depending on the driver type do one of the following:
   * 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-E7nj\_BsO) $ pip install -e .

(pgbackup-E7nj\_BsO) $ pgbackup --driver local ./local-dump.sql postgres://demo:password@54.245.63.9:80/sample

(pgbackup-E7nj\_BsO) $ 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 filename, let’s put all database URL interactions in the pgdump module with a function name of dump\_file\_name. This is a pure function that takes an input and produces an output, so it’s a prime function for us to unit test. Let’s write our tests now:

*tests/test\_pgdump.py* (partial)

def test\_dump\_file\_name\_without\_timestamp():

"""

pgdump.db\_file\_name returns the name of the database

"""

assert pgdump.dump\_file\_name(url) == "db\_one.sql"

def test\_dump\_file\_name\_with\_timestamp():

"""

pgdump.dump\_file\_name returns the name of the database

"""

timestamp = "2017-12-03T13:14:10"

assert pgdump.dump\_file\_name(url, timestamp) == "db\_one-2017-12-03T13:14:10.sql"

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 print a timestamp with [time.strftime](https://docs.python.org/3/library/time.html#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.

For our internal tools, there’s a good chance that we won’t be open sourcing every little tool that we write, but we will want it to be distributable. The newest and preferred way to distribute a python tool is to build a ‘wheel’.

Let’s set up our tool now to be buildable as a wheel so that we can distribute it.

**Documentation For This Video**

* [The wheel documentation](https://wheel.readthedocs.io/en/stable/#defining-the-python-version)

**Adding a setup.cfg**

Before we can generate our [wheel](https://wheel.readthedocs.io/en/stable/#defining-the-python-version), we’re going to want to configure setuptools to not build the wheel for Python 2. We can’t build for Python 2 because we used string interpolation. We’ll put this configuration in a setup.cfg:

*setup.cfg*

[bdist\_wheel]

python-tag = py36

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

(pgbackup-E7nj\_BsO) $ python setup.py bdist\_wheel

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

(pgbackup-E7nj\_BsO) $ pip uninstall pgbackup

(pgbackup-E7nj\_BsO) $ pip install dist/pgbackup-0.1.0-py36-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-E7nj\_BsO) $ python

>>> import boto3

>>> f = open('dist/pgbackup-0.1.0-py36-none-any.whl', 'rb')

>>> client = boto3.client('s3')

>>> client.upload\_fileobj(f, 'pyscripting-db-backups', 'pgbackup-0.1.0-py36-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-E7nj\_BsO) $ exit

$ pip3.6 install --user https://s3.amazonaws.com/pyscripting-db-backups/pgbackup-0.1.0-py36-none-any.whl

$ pgbackup --help