[previous assignment](http://www.thinkful.com/courses/PIP-001/unit/2/lesson/1/assignment/3) [next assignment](http://www.thinkful.com/courses/PIP-001/unit/2/lesson/1/assignment/5)

 mark complete

[LESSON 1 - WORKING WITH SPREADSHEETS](http://www.thinkful.com/courses/PIP-001/unit/2/lesson/1)ASSIGNMENT 4

**Building a command line snippets app**

Estimated Time: 60 minutes

In the next two assignments we are going to code a simple command line application that will build on your knowledge of the csv and the argparse libraries. The application will allow you to store and retrieve snippets of text. You can use this to save notes about how Python works as you continue to work through the course.

Using logging

As we build the application we are going to be introducing a new module called logging. This allows us to track what our program is doing and makes an excellent tool for debugging code. The first thing we need to do is to configure the logging module using the logging.basicConfig function.

Create a new folder for this project called 'snippets-app' and from the command line cd into it. Do a git init to start version control, and go ahead and create a remote repository on Github to link this local repo to.

Now inside of your 'snippets-app' folder, create a new file called *snippets.py* and add the following code:

import logging

import csv

# Set the log output file, and the log level

logging.basicConfig(filename="output.log", level=logging.DEBUG)

The filename argument describes where we want the log to be saved. In this case, it will write to a file called "output.log" in the current directory. The level argument sets the log level. The logging module splits log messages into one of five severity levels:

* **DEBUG**: Detailed information, typically of interest only when diagnosing problems.
* **INFO**: Confirmation that things are working as expected.
* **WARNING**: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
* **ERROR**: Due to a more serious problem, the software has not been able to perform some function.
* **CRITICAL**: A serious error, indicating that the program itself may be unable to continue running.

When we set the log level to DEBUG all of the messages will be logged. If it is set to WARNING only log messages with a severity of WARNING or higher will be logged.

Saving snippets

Now let's start looking at how to save snippets. We are going to store the snippets in a CSV file using the csvmodule. So lets try to write a function that:

1. Opens a CSV file for writing
2. Writes a snippet of text, with an associated name to the CSV file

First we can create a *function stub*: a function that is defined but does nothing. This will let us think about the inputs that we need in order to save a snippet. In your *snippets.py* file try adding the following stub:

def put(name, snippet, filename):

""" Store a snippet with an associated name in the CSV file """

pass

Notice how our function takes three arguments as inputs; the name of the snippet, the snippet text and a filename where we should store the snippet.

Now we can start to add some functionality. First let's focus on opening the file so we can write our snippets to it:

import logging

import csv

# Set the log output file, and the log level

logging.basicConfig(filename="output.log", level=logging.DEBUG)

def put(name, snippet, filename):

""" Store a snippet with an associated name in the CSV file """

logging.info("Writing {}:{} to {}".format(name, snippet, filename))

logging.debug("Opening file")

with open(filename, "a") as f:

writer = csv.writer(f)

First we use the info and debug functions from the logging module to write messages to the log with a corresponding severity.

Next we open the file, giving the argument "a" to the open function to indicate that we want to create the file if it doesn't exist, or append to it otherwise.

We then use the csv library to create a writer object. The writer object will allow us to add a new row to the CSV file for each snippet we create. We pass our file in as the csvfile argument to the csv.writer function, indicating that we want the writer to use our opened file. Notice how we have added an import statement to allow us to use the csv library.

Finally we can think about adding code that actually adds the snippet to the file:

def put(name, snippet, filename):

""" Store a snippet with an associated name in the CSV file """

logging.info("Writing {}:{} to {}".format(name, snippet, filename))

logging.debug("Opening file")

with open(filename, "a") as f:

writer = csv.writer(f)

logging.debug("Writing snippet to file".format(name, snippet))

writer.writerow([name, snippet])

logging.debug("Write sucessful")

return name, snippet

We use the writerow method of the writer object to add a new row to the CSV file. We pass in a list the the method which contains the name of the snippet and the snippet value. These will be stored as two separate columns in the file.

Finally we return a tuple containing the information that we have added. Returning the information is a fairly common pattern when adding, removing or updating information using a function. We will see shortly how this helps to clean up our code.

Before moving on, now is a good time to add and commit your work so far.

Trying it out

Now that we have a function to save our snippets, let's try it out. Open up a terminal in the same directory as your snippets.py folder, and then open up the Python interpreter. Now try using your function to add a couple of snippets to a file:

>>> import snippets

>>> snippets.put("spam", "eggs", "snippets.csv")

('spam', 'eggs')

>>> snippets.put("green eggs", "ham", "snippets.csv")

('green eggs', 'ham')

You should see that the *snippets.csv* file has been created in the same directory. Try opening it up and taking a look at the contents. There should be two rows, one containing spam and eggs, and the other containing green eggs and ham. Neat huh?

There should also be a file called *output.log* which has been created. Have a quick look at it to make sure that your debugging messages are being created correctly.

Adding a command line interface

Although we now know that our function works, we don't want to have to open up the Python console every time we want to add a snippet. So let's use the argparse module to add a command line interface to our function.

The first stage of this is to work out roughly how we want our interface to look. The put function we have just written takes three arguments, so it seems to make sense to allow three command line arguments. As we plan to allow snippets to be retrieved and set, it probably makes sense to add a fourth argument specifying the command type (storage or retrieval). So let's sketch out a few options for how we could use our app.

# Store an item using fully qualified names

python snippets.py --type "put" --name "spam" --snippet "eggs" --filename "snippets.csv"

# Store an item using abbreviations

python snippets.py -t "put" -n "spam" -s "eggs" -f "snippets.csv"

# Have a default argument for filename, so we can leave it empty

python snippets.py -t "put" -n "spam" -s "eggs"

# Use positional rather than optional arguments

python snippets.py put spam eggs snippets.csv

# As above, but with a defualt argument for filename

python snippets.py put spam eggs

Each option has advantages and disadvantages. For an app as simple as this the use of optional arguments everywhere seems too verbose, even with the abbreviations. A default filename seems to make sense so you can concentrate on simple adding and retrieving the snippets without worrying about where they are stored. So for this assignment we will settle on the last option as our ideal syntax.

Now that we have settled on an interface, lets start to build the parser using the argparse module. As a first step we can create a make\_parser function:

def make\_parser():

""" Construct the command line parser """

logging.info("Constructing parser")

description = "Store and retrieve snippets of text"

parser = argparse.ArgumentParser(description=description)

return parser

Here we create an instance of the ArgumentParser object from the argparse module. Note that you will need to add import argparse at the top of your script so we can access this object. We then simply return the parser object so we can use it later to parse the command line arguments.

Now let's add a main function to the bottom of *snippets.py* which will create our parser and try to parse the command line arguments.

def main():

""" Main function """

logging.info("Starting snippets")

parser = make\_parser()

arguments = parser.parse\_args(sys.argv[1:])

if \_\_name\_\_ == "\_\_main\_\_":

main()

The main function uses our make\_parser function to create the command line parser. We then call the parse\_argsmethod of the parser, passing in all of the command line arguments we have provided except the first one. As we learned in Unit 1, the first argument of argv always contains the name of our program, which we don't want to try to parse. Also don't forget to import the sys module at the top of your program so you can access the argvvariable.

Also at this point we have added a main block to the app that calls our main function. This means that we can try running our script from the command line. Open up a terminal, navigate to your project folder, and enter python snippets.py --help. You should see some help text which has been automatically generated for us by the parser object.

Before moving on, be sure to commit your changes.

Making our interface accept arguments

Now that we have a parser let's start building up our interface. As a first step we are going to split our parser into subparsers - one for the storage command and one for the retrieval command. This is similar to how apps like gitwork, with different commands taking different arguments.

def make\_parser():

""" Construct the command line parser """

logging.info("Constructing parser")

description = "Store and retrieve snippets of text"

parser = argparse.ArgumentParser(description=description)

subparsers = parser.add\_subparsers(help="Available commands")

# Subparser for the put command

logging.debug("Constructing put subparser")

put\_parser = subparsers.add\_parser("put", help="Store a snippet")

return parser

Here we use the add\_subparsers method of the parser to allow us to add multiple subparsers. We then use the add\_parser method of the subparsers object to create our new subparser. This subparser works in the same way as a standard ArgumentParser instance, allowing us to add arguments. So let's try to expand our function even further to add in the arguments for a put command:

def make\_parser():

""" Construct the command line parser """

logging.info("Constructing parser")

description = "Store and retrieve snippets of text"

parser = argparse.ArgumentParser(description=description)

subparsers = parser.add\_subparsers(help="Available commands")

# Subparser for the put command

logging.debug("Constructing put subparser")

put\_parser = subparsers.add\_parser("put", help="Store a snippet")

put\_parser.add\_argument("name", help="The name of the snippet")

put\_parser.add\_argument("snippet", help="The snippet text")

put\_parser.add\_argument("filename", default="snippets.csv", nargs="?",

help="The snippet filename")

put\_parser.set\_defaults(command="put")

return parser

Here we use the add\_argument method to add the three positional arguments. For the filename argument we give two additional options to the add\_argument method. We use the default argument to set the default filename. The nargs="?" option says that we can leave out the argument if we just want to use the default value.

Finally we use the set\_defaults method to add a default argument which is contained every time we perform a put command. This is helpful as it allows us to work out which command was entered after the parsing has taken place.

Now let's try it out. In a terminal we can run the program a few different ways to check that our parser works:

# Shows the help for which subcommands are available

python snippets.py --help

# Show the help for the put subcommand

python snippets.py put --help

# Standard put command

python snippets.py put spam eggs

# Put command with an optional filename

python snippets.py put spam eggs test.py

# Put command missing an argument (this should fail)

python snippets.py put spam

We've reached another milestone, so be sure to commit these changes.

Putting it all together

Now that we have both our functionality and our interface, all that remains is to join the two together. Let's extend our main function:

def main():

""" Main function """

logging.info("Starting snippets")

parser = make\_parser()

arguments = parser.parse\_args(sys.argv[1:])

# Convert parsed arguments from Namespace to dictionary

arguments = vars(arguments)

command = arguments.pop("command")

if command == "put":

name, snippet = put(\*\*arguments)

print "Stored '{}' as '{}'".format(snippet, name)

There are a few things which might be new to you in this code. Firstly we use the built in vars function to convert the Namespace object returned by parse\_args into a dictionary. This is purely for convenience, as the dictionary object is somewhat easier to work with.

Next we use the pop method of the dictionary. This removes and returns the command item which we added earlier to distinguish between the two commands.

We then call our put function, using \*\*arguments as an argument. The use of the double-star operator in this context is known as *unpacking*. It converts the key-value pairs in the dictionary into keyword arguments to the function. For example:

# Code without argument unpacking

put(name="spam", snippet="eggs", filename="snippets.py")

# Identical code which uses argument unpacking

arguments = {

"name": "spam",

"snippet": "eggs",

"filename": "snippets.py"

}

put(\*\*arguments)

Try running you app and using it to add items to a CSV file. Open up the file to make sure that the items are being added correctly. If so, congratulations - that's the first part of our app finished.

You'll be on your own to add to this code base in the next assignment, so be sure to commit your changes.