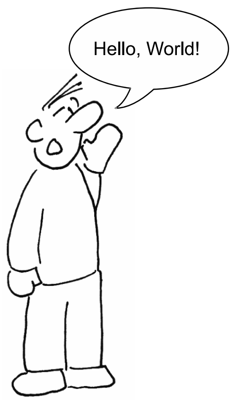
1 How to write and test a Python program

Before you start working on the exercises, I want to discuss how to write programs that are documented and tested. Specifically, we're going to

- Write a Python program to say "Hello, World!"
- Handle command-line arguments using `argparse`
- Run tests for the code with Pytest.
- Learn about `$PATH`
- Use tools like YAPF and Black to format the code
- Use tools like Flake8 and Pylint to find problems in the code
- Use the new.py program to create new programs



1.1 Creating your first program

It's pretty common to write "Hello, World!" as your first program in any language, so let's start there. We're going to work toward making a version that will greet whichever name is passed as an argument. It will also print a helpful message when we ask for it, and we'll use tests to make sure it does everything correctly.

In the 01_hello directory, you'll see several versions of the hello program we'll write. There is also a program called test.py that we'll use to test the program.

Start off by creating a text file called hello.py in that directory. If you are working in VS Code or PyCharm, you can use File > Open to open the 01_hello directory as a project. Both tools have something like a File > New menu option that will allow you to create a new file in that directory. It's very important to create the hello.py file *inside* the 01_hello directory so that the test.py program can find it.

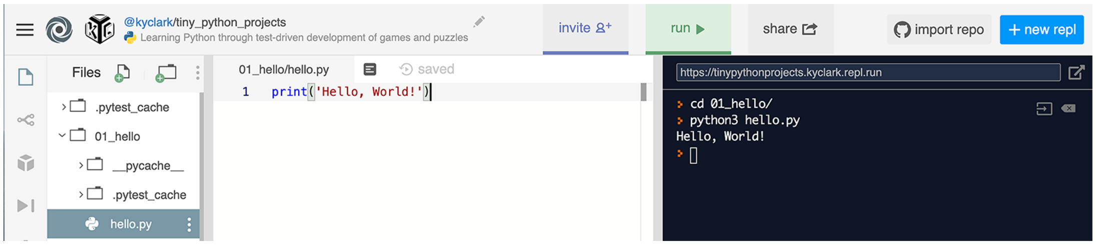
Once you've started a new file, add this line:

```
print('Hello, World!')
```

It's time to run your new program! Open a terminal window in VS Code or PyCharm or in some other terminal, and navigate to the directory where your hello.py program is located. You can run it with the command `python3 hello.py`—this causes Python version 3 to execute the commands in the file named hello.py. You should see this:

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

Figure 1.1 shows how it looks in the Repl.it interface.


@kyclark/tiny_python_projects
Learning Python through test-driven development of games and puzzles
invite
run
share
import repo
+ new repl
Files
.pytest_cache
01_hello
__pycache__
.pytest_cache
hello.py
01_hello/hello.py
saved
https://tinypythonprojects.kyclark.repl.run

```
print('Hello, World!')
```

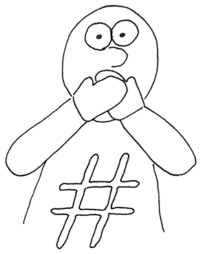
```
> cd 01_hello/
> python3 hello.py
Hello, World!
>
```

Figure 1.1 Writing and running our first program using Repl.it

If that was your first Python program, congratulations!

1.2 Comment lines

In Python, the # character and anything following it is ignored by Python. This is useful for adding comments to your code or temporarily disabling lines of code when testing and debugging. It's always a good idea to document your programs, indicating the purpose of the program or the author's name and email address, or both. We can use a comment for that:



```
# Purpose: Say hello
print('Hello, World!')
```

If you run this program again, you should see the same output as before because the "Purpose" line is ignored. Note that any text to the left of the # is executed, so you can add a comment to the end of a line if you like.

1.3 Testing your program

The most fundamental idea I want to teach you is how to test your programs. I've written a test.py program in the 01_hello directory that we can use to test our new hello.py program.

We will use pytest to execute all the commands and tell us how many tests we passed. We'll include the -v option, which tells `pytest` to create "verbose" output. If you run it like this, you should see the following output as the first several lines. After that will follow many more lines showing you more information about the tests that didn't pass.

> **NOTE** If you get the error "pytest: command not found," you need to install the `pytest` module. Refer to the "Installing modules" section in the book's introduction.

The second test tries to run the program with python3 hello.py and then checks if the program printed "Hello, World!" If you miss even one character, like forgetting a comma, the test will point out the error, so read carefully!

```
$ pytest -v test.py
============================= test session starts ==============================
...
collected 5 items

test.py::test_exists PASSED                                              [ 20%]
test.py::test_runnable PASSED                                            [ 40%]
test.py::test_executable FAILED                                          [ 60%]
test.py::test_usage FAILED                                               [ 80%]
test.py::test_input FAILED                                               [100%]

=================================== FAILURES ===================================
```

The first test always checks that the expected file exists. Here the test looks for hello.py.

The fourth test asks the program for help and doesn't get anything. We're going to add the ability to print a "usage" statement that describes how to use our program.

The last test checks that the program can greet a name that we'll pass as an argument. Since our program doesn't yet accept arguments, we'll need to add that, too.

The third test checks that the program is "executable." This test fails, so next we'll talk about how to make that pass.

I've written the tests in an order that I hope will help you write the program in a logical fashion. If the program doesn't pass one of the tests, there's no reason to continue running the tests after it. I recommend you always run the tests with the flags -x, to stop on the first failing test, and -v, to print verbose output. You can combine these like -xv or -vx. Here's what our tests look like with those options:

```
$ pytest -xv test.py
============================= test session starts ==============================
...
collected 5 items
```

```
test.py::test_exists PASSED                                              [ 20%]
test.py::test_runnable PASSED                                            [ 40%]
test.py::test_executable FAILED                                          [ 60%]
```

This test fails. No more tests are run because we ran pytest with the -x option.

```
=================================== FAILURES ===================================
_______________________________ test_executable ________________________________

    def test_executable():
        """Says 'Hello, World!' by default"""

        out = getoutput({prg})
>       assert out.strip() == 'Hello, World!'
E       AssertionError: assert '/bin/sh: ./h...ission denied' == 'Hello, World!'
E         - /bin/sh: ./hello.py: Permission denied
E         + Hello, World!

test.py:30: AssertionError
!!!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!!!!
========================= 1 failed, 2 passed in 0.09s ==========================
```

The angle bracket (>) at the beginning of this line shows the source of the subsequent errors.

The hyphen character (-) is showing that the actual output from the command is "Permission denied."

The plus character (+) shows that the test expected to get "Hello, World!"

The "E" at the beginning of this line shows that this is an "Error" you should read. The AssertionError is saying that the test.py program is trying to execute the command ./hello.py to see if it will produce the text "Hello, World!"

Let's talk about how to fix this error.

1.4 Adding the #! (shebang) line

One thing you have learned so far is that Python programs live in plain text files that you ask `python3` to execute. Many other programming languages, such as Ruby and Perl, work in the same way—we type Ruby or Perl commands into a text file and run it with the right language. It's common to put a special comment line in programs like these to indicate which language needs to be used to execute the commands in the file.

This comment line starts off with `#!`, and the nickname for this is "shebang" (pronounced "shuh-bang"—I always think of the `#` as the "shuh" and the `!` as the "bang!"). Just as with any other comment, Python will ignore the shebang, but the operating system (like macOS or Windows) will use it to decide which program to use to run the rest of the file.

Here is the shebang you should add:

```
#!/usr/bin/env python3
```

The `env` program will tell you about your "environment." When I run `env` on my computer, I see many lines of output like `USER=kyclark` and `HOME=/Users/kyclark`. These values are accessible as the variables `$USER` and `$HOME`:

```
$ echo $USER
kyclark
$ echo $HOME
/Users/kyclark
```

If you run `env` on your computer, you should see your login name and your home directory. They will, of course, have different values from mine, but we both (probably) have both of these concepts.

You can use the `env` command to find and run programs. If you run `env python3`, it will run a `python3` program if it can find one. Here's what I see on my computer:

```
$ env python3
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `env` program is looking for `python3` in the environment. If Python has not been installed, it won't be able to find it, but it's also possible that Python has been installed more than once. You can use the `which` command to see which `python3` it finds:

```
$ which python3
/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

If I run this on Repl.it, I can see that `python3` exists in a different place. Where does it exist on your computer?

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

Just as my `$USER` name is different from yours, my `python3` is probably different from yours. If the `env` command is able to find a `python3`, it will execute it. As shown previously, if you run `python3` by itself, it will open a REPL.

If I were to put my `python3` path as the shebang line, like so,

```
#!/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

my program would not work on another computer that has `python3` installed in a different location. I doubt it would work on your computer, either. This is why you should always use the `env` program to find the `python3` that is specific to the machine on which it's running.

Now your program should look like this:

```
#!/usr/bin/env python3
# Purpose: Say hello
print('Hello, World!')
```

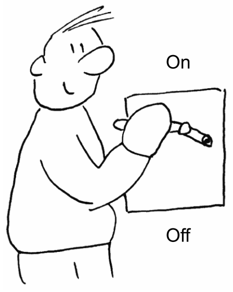
The shebang line tells the operating system to use /usr/bin/env to find python3 to interpret this program.

A comment line documenting the purpose of the program

A Python command to print some text to the screen

1.5 Making a program executable

So far we've been explicitly telling python3 to run our program, but since we added the shebang, we can execute the program directly and let the OS figure out that it should use python3. The advantage of this is that we could copy our program to a directory where other programs live and execute it from anywhere on our computer.



The first step in doing this is to make our program "executable" using the command chmod (*change mode*). Think of it as turning your program "on." Run this command to make hello.py executable:

```
$ chmod +x hello.py
```

The +x will add an "executable" attribute to the file.

Now you can run the program like so:

```
$ ./hello.py
Hello, World!
```

The ./ is the current directory, and it's necessary to run a program when you are in the same directory as the program.

1.6 Understanding $PATH

One of the biggest reasons to set the shebang line and make your program executable is so that you can install your Python programs just like other commands and programs. We used the which command earlier to find the location of python3 on the Repl.it instance:

```
$ which python3
/home/runner/.local/share/virtualenvs/python3/bin/python3
```

How was the env program able to find it? Windows, macOS, and Linux all have a $PATH variable, which is a list of directories the OS will look in to find a program. For instance, here is the $PATH for my Repl.it instance:

```
> echo $PATH
/home/runner/.local/share/virtualenvs/python3/bin:/usr/local/bin:\
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

The directories are separated by colons (:). Notice that the directory where python3 lives is the first one in $PATH. It's a pretty long string, so I broke it with the \ character to make it easier to read. If you copy your hello.py program to any of the directories listed in your $PATH, you can execute a program like hello.py without the leading ./ and without having to be in the same directory as the program.

Think about $PATH like this: If you lose your keys in your house, would you start looking in the upper-left kitchen cabinet and work your way through each cabinet, and then all the drawers where you keep your silverware and kitchen gadgets, and then move on to your bathrooms and bedroom closets? Or would you start by looking in places where you normally put your keys, like the key hooks beside the front door, and then move on to search the pockets of your favorite jacket and your purse or backpack, and then maybe look under the couch cushions, and so forth?

The `$PATH` variable is a way of telling your computer to only look in places where executable programs can be found. The only alternative is for the OS to search *every directory*, and that could take several minutes or possibly even hours! You can control both the names of the directories in the `$PATH` variable and their relative order so that the OS will find the programs you need.

It's very common for programs to be installed into /usr/local/bin, so we could try to copy our program there using the `cp` command. Unfortunately, I do not have permission to do this on Repl.it:

```
> cp 01_hello/hello.py /usr/local/bin
cp: cannot create regular file '/usr/local/bin/hello.py': Permission denied
```

But I can do this on my own laptop:

```
$ cp hello.py /usr/local/bin/
```

I can verify that the program is found:

```
$ which hello.py
/usr/local/bin/hello.py
```

And now I can execute it from any directory on my computer:

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

1.6.1 *Altering your $PATH*

Often you may find yourself working on a computer that won't allow you to install programs into your `$PATH`, such as on Repl.it. An alternative is to alter your `$PATH` to include a directory where you can put your programs. For instance, I often create a bin directory in my home directory, which can often be written with the tilde (~).

On most computers, `~/bin` would mean "the bin directory in my home directory." It's also common to see `$HOME/bin` where `$HOME` is the name of your home directory. Here is how I create this directory on the Repl.it machine, copy a program to it, and then add it to my `$PATH`:

```
$ mkdir ~/bin
$ cp 01_hello/hello.py ~/bin
$ PATH=~/bin:$PATH
$ which hello.py
/home/runner/bin/hello.py
```

Use the mkdir ("make directory") command to create ~/bin.

Use the cp command to copy the 01_hello/hello.py program to the ~/bin directory.

Put the ~/bin directory first in $PATH.

Use the which command to look for the hello.py program. If the previous steps worked, the OS should now be able to find the program in one of the directories listed in $PATH.

Now I can be in any directory,

```
$ pwd
/home/runner/tinypythonprojects
```

and I can run it:

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

Although the shebang and the executable stuff may seem like a lot of work, the payoff is that you can create a Python program that can be installed onto your computer or anyone else's and run just like any other program.

1.7 Adding a parameter and help

Throughout the book, I'll use string diagrams to visualize the inputs and outputs of the programs we'll write. If we created one for our program now (as in figure 1.2), there would be no inputs, and the output would always be "Hello, World!"

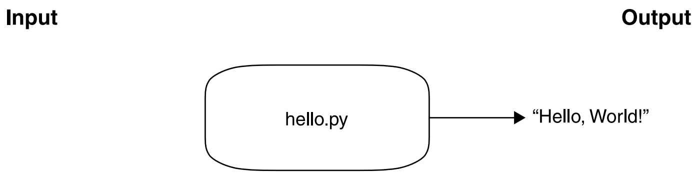


Figure 1.2 A string diagram representing our hello.py program that takes no inputs and always produces the same output

It's not terribly interesting for our program to always say "Hello, World!" It would be nice if it could say "Hello" to something else, like the entire universe. We could do this by changing the code as follows:

```
print('Hello, Universe')
```

But that would mean we'd have to change the code every time we wanted to make it greet a different name. It would be better to change the *behavior* of the program without always having to change *the program itself.*

We can do that by finding the parts of the program that we want to change—like the name to greet— and providing that value as as an *argument* to our program. That is, we'd like our program to work like this:

```
$ ./hello.py Terra
Hello, Terra!
```

How would the person using our program know to do this? *It's our program's responsibility to provide a help message!* Most command-line programs will respond to arguments like -h and --help with helpful messages about how to use the programs. We need our program to print something like this:

```
$ ./hello.py -h
usage: hello.py [-h] name

Say hello

positional arguments:
  name        Name to greet

optional arguments:
  -h, --help  show this help message and exit
```

Note that name is called a positional argument.

To do this, we can use the argparse module. Modules are files of code we can bring into our programs. We can also create modules to share our code with other people. There are hundreds to thousands of modules you can use in Python, which is one of the reasons why it's so exciting to use the language.

The argparse module will "parse" the "arguments" to the program. To use it, change your program as follows. I recommend you type everything yourself and don't copy and paste.

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

parser = argparse.ArgumentParser(description='Say hello')
parser.add_argument('name', help='Name to greet')
args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

The shebang line tells the OS which program to use to execute this program.

This comment documents the purpose of the program.

We must import the argparse module to use it.

The parser will figure out all the arguments. The description appears in the help message.

We need to tell the parser to expect a name that will be the object of our salutations.

We ask the parser to parse any arguments to the program.

We print the greeting using the args.name value.

Figure 1.3 shows a string diagram of our program now.

Now when you try to run the program like before, it triggers an error and a "usage" statement (notice that "usage" is the first word of the output):

```
$ ./hello.py
usage: hello.py [-h] name
hello.py: error: the following arguments are required: name
```

We run the program with no arguments, but the program now expects a single argument (a "name").

Since the program doesn't get the expected argument, it stops and prints a "usage" message to let the user know how to properly invoke the program.

The error message tells the user that they have not supplied a required parameter called "name."

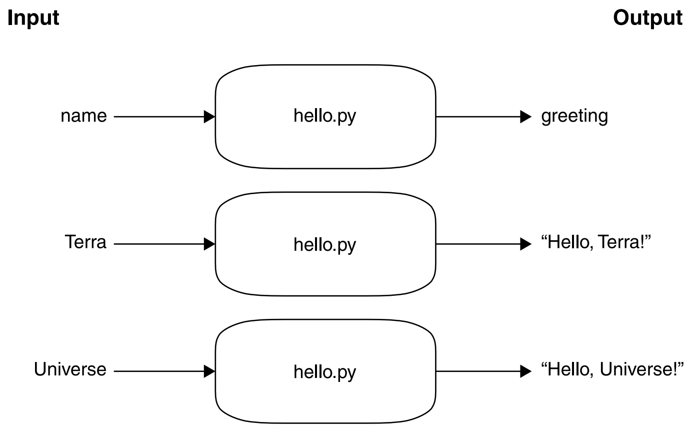


Figure 1.3 Now our string diagram shows that our program can take an argument and produce a message based on that value.

We've changed the program so that it requires a name or it won't run. That's pretty cool! Let's give it a name to greet:

```
$ ./hello.py Universe
Hello, Universe!
```

Try running your program with both the `-h` and `--help` arguments, and verify that you see the help messages.

The program works really well now and has nice documentation, all because we added those few lines using `argparse`. That's a big improvement.

1.8 Making the argument optional

Suppose we'd like to run the program like before, with no arguments, and have it print "Hello, World!" We can make the `name` optional by changing the name of the argument to `--name`:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

parser = argparse.ArgumentParser(description='Say hello')
parser.add_argument('-n', '--name', metavar='name',
                    default='World', help='Name to greet')
args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

The only change to this program is adding -n and --name for the "short" and "long" option names. We also indicate a default value. "metavar" will show up in the usage to describe the argument.

Now we can run it like before:

```
$ ./hello.py
Hello, World!
```

Or we can use the `--name` option:

```
$ ./hello.py --name Terra
Hello, Terra!
```

And our help message has changed:

```
$ ./hello.py -h
usage: hello.py [-h] [-n NAME]

Say hello

optional arguments:
  -h, --help            show this help message and exit
  -n name, --name name  Name to greet
```

The argument is now optional and no longer a positional argument. It's common to provide both short and long names to make it easy to type the options. The metavar value of "name" appears here to describe what the value should be.

Figure 1.4 shows a string diagram that describes our program.

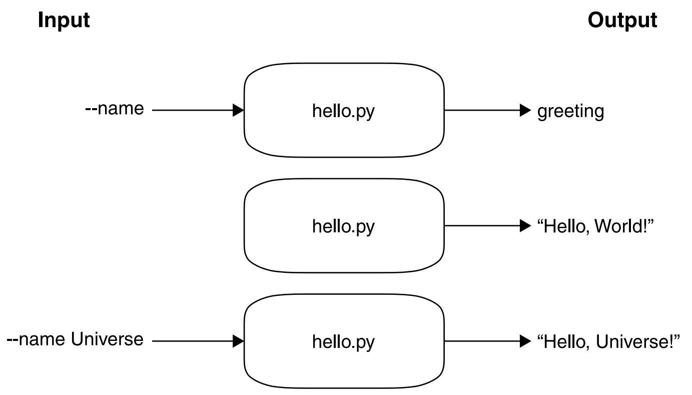


Figure 1.4 The `name` parameter is now optional. The program will greet a given name or will use a default value when it's missing.

Our program is really flexible now, greeting a default value when run with no arguments or allowing us to say "hi" to something else. Remember that parameters that start with dashes are *optional*, so they can be left out, and they may have default values. Parameters that *don't* start with dashes are *positional* and are usually required, so they do not have default values.

Table 1.1 Two kinds of command-line parameters

Type	Example	Required	Default
Positional	`name`	Yes	No
Optional	`-n` (short), `--name` (long)	No	Yes

1.9 Running our tests

Let's run our tests again to see how we are doing:

```
$ make test
pytest -xv test.py
============================ test session starts =============================
...
collected 5 items

test.py::test_exists PASSED                                              [ 20%]
test.py::test_runnable PASSED                                            [ 40%]
test.py::test_executable PASSED                                          [ 60%]
test.py::test_usage PASSED                                               [ 80%]
test.py::test_input PASSED                                               [100%]

============================= 5 passed in 0.38s ==============================
```

Wow, we're passing all our tests! I actually get excited whenever I see my programs pass all their tests, even when I'm the one who wrote the tests. Before we were failing on the usage and input tests. Adding the `argparse` code fixed both of those because it allows us to accept arguments when our program runs, and it will also create documentation about how to run our program.

1.10 Adding the main() function

Our program works really well now, but it's not quite up to community standards and expectations. For instance, it's very common for computer programs—not just ones written in Python—to start at a place called `main()`. Most Python programs define a function called `main()`, and there is an idiom to call the `main()` function at the end of the code, like this:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

def main():
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    args = parser.parse_args()
    print('Hello, ' + args.name + '!')

if __name__ == '__main__':
    main()
```

def defines a function, named main() in this case. The empty parentheses show that this function accepts no arguments.

Every program or module in Python has a name that can be accessed through the variable __name__. When the program is executing, __name__ is set to "__main__".[1]

If this is true, call the main() function.

[1] See Python's documentation of `main` for more information: https://docs.python.org/3/library/__main__.html.

As our programs get longer, we'll start creating more functions. Python programmers approach this in different ways, but in this book I will always create and execute a `main()` function to be consistent. To start off, we'll always put the main part of our program inside the `main()` function.

1.11 Adding the get_args() function

As a matter of personal taste, I like to put all the `argparse` code into a separate place that I always call `get_args()`. Getting and validating arguments is one concept in my mind, so it belongs by itself. For some programs, this function can get quite long.

I always put `get_args()` as the first function so that I can see it immediately when I read the source code. I usually put `main()` right after it. You are, of course, welcome to structure your programs however you like.

Here is what the program looks like now:

```
#!/usr/bin/env python3
# Purpose: Say hello

import argparse

def get_args():
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    return parser.parse_args()

def main():
    args = get_args()
    print('Hello, ' + args.name + '!')

if __name__ == '__main__':
    main()
```

The get_args() function is dedicated to getting the arguments. All the argparse code now lives here.

We need to call return to send the results of parsing the arguments back to the main() function.

The main() function is much shorter now.

Call the get_args() function to get parsed arguments. If there is a problem with the arguments or if the user asks for --help, the program never gets to this point because argparse will cause it to exit. If our program does make it this far, the input values must have been OK.

Nothing has changed about the way the program works. We're just organizing the code to group ideas together—the code that deals with `argparse` now lives in the `get_args()` function, and everything else lives in `main()`. Just to be sure, go run the test suite!

1.11.1 Checking style and errors

Our program works really well now. We can use tools like Flake8 and Pylint to check if our program has problems. These tools are called *linters*, and their job is to suggest ways to improve a program. If you haven't installed them yet, you can use the `pip` module to do so now:

```
$ python3 -m pip install flake8 pylint
```

The Flake8 program wants me to put two blank lines between each of the function `def` definitions:

```
$ flake8 hello.py
hello.py:6:1: E302 expected 2 blank lines, found 1
hello.py:12:1: E302 expected 2 blank lines, found 1
hello.py:16:1: E305 expected 2 blank lines after class or function definition,
     found 1
```

And Pylint says that the functions are missing documentation ("docstrings"):

```
$ pylint hello.py
************* Module hello
hello.py:1:0: C0114: Missing module docstring (missing-module-docstring)
hello.py:6:0: C0116: Missing function or method docstring (missing-function-
     docstring)
hello.py:12:0: C0116: Missing function or method docstring (missing-function-
     docstring)

-------------------------------------------------------------------
Your code has been rated at 7.00/10 (previous run: -10.00/10, +17.00)
```

A *docstring* is a string that occurs just after the `def` of the function. It's common to have several lines of documentation for a function, so programmers often will use Python's triple quotes (single or double) to create a multiline string. Following is what the program looks like when I add docstrings. I have also used YAPF to format the program and fix the spacing problems, but you are welcome to use Black or any other tool you like.

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Say hello
"""

import argparse
```

Triple-quoted, multiline docstring for the entire program. It's common practice to write a long docstring just after the shebang to document the overall purpose of the function. I like to include at least my name, email address, and the purpose of the script so that any future person using the program will know who wrote it, how to get in touch with me if they have problems, and what the program is supposed to do.

```
# --------------------------------------------------
def get_args():
    """Get the command-line arguments"""

    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--
     name', default='World', help='Name to greet')
    return parser.parse_args()
```

A big horizontal "line" comment to help me find the functions. You can omit these if you don't like them.

The docstring for the get_args() function. I like to use triple quotes even for a single-line comment, as they help me to see the docstring better.

```
# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    print('Hello, ' + args.name + '!')
```

The main() function is simply where the program begins, so there's not much to say in the docstring. I think it's (at least a little) funny to always put "Make a jazz noise here," but you can put whatever you like.

```
# --------------------------------------------------
if __name__ == '__main__':
    main()
```

To learn how to use YAPF or Black on the command line, run them with the `-h` or `--help` flag and read the documentation. If you are using an IDE like VS Code or PyCharm, or if you are using the Repl.it interface, there are commands to reformat your code.

1.12 Testing hello.py

We've made many changes to our program—are we sure it still works correctly? Let's run our test again.

This is something you will do literally hundreds of times, so I've created a shortcut you might like to use. In every directory, you'll find a file called Makefile that looks like this:

```
$ cat Makefile
.PHONY: test

test:
    pytest -xv test.py
```

If you have the program `make` installed on your computer, you can run `make test` when you are in the 01_hello directory. The `make` program will look for a Makefile in your current working directory and then look for a recipe called "test." There it will find that the command to run for the "test" target is `pytest -xv test.py`, so it will run that command for you.

```
$ make test
pytest -xv test.py
```

```
============================= test session starts ==============================
...
collected 5 items

test.py::test_exists PASSED                                              [ 20%]
test.py::test_runnable PASSED                                            [ 40%]
test.py::test_executable PASSED                                          [ 60%]
test.py::test_usage PASSED                                               [ 80%]
test.py::test_input PASSED                                               [100%]

============================== 5 passed in 0.75s ===============================
```

If you do not have `make` installed, you might like to install it and learn about how Makefiles can be used to execute complicated sets of commands. If you do not want to install or use `make`, you can always run `pytest -xv test.py` yourself. They both accomplish the same task.

The important point is that we were able to use our tests to verify that our program still does exactly what it is supposed to do. As you write programs, you may want to try different solutions. Tests give you the freedom to rewrite a program (also called "refactoring your code") and know that it still works.

1.13 Starting a new program with new.py

The `argparse` module is a standard module that is always installed with Python. It's widely used because it can save us so much time in parsing and validating the arguments to our programs. You'll be using `argparse` in every program for this book, and you'll learn how you can use it to convert text to numbers, to validate and open files, and much more. There are so many options that I created a Python program called new.py that will help you start writing new Python programs that use `argparse`.

I have put this new.py program into the bin directory of the GitHub repo. I suggest you use it to start every new program you write. For instance, you could create a new version of hello.py using new.py. Go to the top level of your repository and run this:

```
$ bin/new.py 01_hello/hello.py
"01_hello/hello.py" exists.  Overwrite? [yN] n
Will not overwrite. Bye!
```

The new.py program will not overwrite an existing file unless you tell it to, so you can use it without worrying that you might erase your work. Try using it to create a program with a different name:

```
$ bin/new.py 01_hello/hello2.py
Done, see new script "01_hello/hello2.py."
```

Now try executing that program:

```
$ 01_hello/hello2.py
usage: hello2.py [-h] [-a str] [-i int] [-f FILE] [-o] str
hello2.py: error: the following arguments are required: str
```

Let’s look at the source code of the new program:

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2020-02-28
Purpose: Rock the Casbah
"""
```

The shebang line should use the env program to find the python3 program.

This docstring is for the program as a whole.

```
import argparse
import os
import sys
```

These lines import various modules that the program needs.

```
# --------------------------------------------------
def get_args():
    """Get command-line arguments"""
```

The get_args() function is responsible for parsing and validating arguments.

```
    parser = argparse.ArgumentParser(
        description='Rock the Casbah',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('positional',
                        metavar='str',
                        help='A positional argument')
```

Define a “positional” argument like our first version of hello.py that had a name argument.

```
    parser.add_argument('-a',
                        '--arg',
                        help='A named string argument',
                        metavar='str',
                        type=str,
                        default='')
```

Define an “optional” argument like when we changed to use the --name option.

```
    parser.add_argument('-i',
                        '--int',
                        help='A named integer argument',
                        metavar='int',
                        type=int,
                        default=0)
```

Define an optional argument that must be an integer value.

```
    parser.add_argument('-f',
                        '--file',
                        help='A readable file',
                        metavar='FILE',
                        type=argparse.FileType('r'),
                        default=None)
```

Define an optional argument that must be a file.

```
    parser.add_argument('-o',
                        '--on',
```

Define a “flag” option that is either “on” when present or “off” when absent. You’ll learn more about these later.

```
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args()
```

Return the parsed arguments to main(). If there are any problems, like if the --int value is some text rather than a number like 42, argparse will print an error message and the "usage" for the user.

```
# --------------------------------------------------
def main():
    """Make a jazz noise here"""
```

Define the main() function where the program starts.

```
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    file_arg = args.file
    flag_arg = args.on
    pos_arg = args.positional
```

The first thing our main() functions will always do is call get_args() to get the arguments.

Each argument's value is accessible through the long name of the argument. It is not required to have both a short and long name, but it is common and tends to make your program more readable.

```
    print(f'str_arg = "{str_arg}"')
    print(f'int_arg = "{int_arg}"')
    print('file_arg = "{}"'.format(file_arg.name if file_arg else ''))
    print(f'flag_arg = "{flag_arg}"')
    print(f'positional = "{pos_arg}"')


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

When the program is being executed, the __name__ value will be equal to the text "__main__."

If the condition is true, this calls the main() function.

This program will accept the following arguments:

- A single positional argument of the type `str`. *Positional* means it is not preceded by a flag to name it but has meaning because of its position relative to the command name.
- An automatic `-h` or `--help` flag that will cause `argparse` to print the usage.
- A string option called either `-a` or `--arg`.
- A named option argument called `-i` or `--int`.
- A file option called `-f` or `--file`.
- A Boolean (off/on) flag called `-o` or `--on`.

Looking at the preceding list, you can see that new.py has done the following for you:

- Created a new Python program called hello2.py
- Used a template to generate a working program complete with docstrings, a `main()` function to start the program, a `get_args()` function to parse and document various kinds of arguments, and code to start the program running in the `main()` function
- Made the new program executable so that it can be run like `./hello2.py`

The result is a program that you can immediately execute and that will produce documentation on how to run it. After you use new.py to create your new program, you should open it with your editor and modify the argument names and types to suit the needs of your program. For instance, in chapter 2 you'll be able to delete everything but the positional argument, which you should rename from `'positional'` to something like `'word'` (because the argument is going to be a word).

Note that you can control the "name" and "email" values that are used by new.py by creating a file called .new.py (note the leading dot!) in your home directory. Here is mine:

```
$ cat ~/.new.py
name=Ken Youens-Clark
email=kyclark@gmail.com
```

1.14 Using template.py as an alternative to new.py

If you don't want to use new.py, I have included a sample of the preceding program as template/template.py, which you can copy. For instance, in chapter 2 you will need to create the program 02_crowsnest/crowsnest.py.

You can do this with new.py from the top level of the repository:

```
$ bin/new.py 02_crowsnest/crowsnest.py
```

Or you can the use `cp` (copy) command to copy the template to your new program:

```
$ cp template/template.py 02_crowsnest/crowsnest.py
```

The main point is that you won't have to start every program from scratch. I think it's much easier to start with a complete, working program and modify it.

> **NOTE** You can copy new.py to your ~/bin directory. Then you can use it from any directory to create a new program.

Be sure to skim the appendix—it has many examples of programs that use `argparse`. You can copy many of those examples to help you with the exercises.

Summary

- A Python program is plain text that lives in a file. You need the `python3` program to interpret and execute the program file.
- You can make a program executable and copy it to a location in your `$PATH` so that you can run it like any other program on your computer. Be sure to set the shebang to use `env` to find the correct `python3`.
- The `argparse` module will help you document and parse all the parameters to your program. You can validate the types and numbers of arguments, which can be positional, optional, or flags. The usage will be automatically generated.

- We will use the `pytest` program to run the test.py programs for each exercise. The `make test` shortcut will execute `pytest -xv test.py`, or you can run this command directly.
- You should run your tests often to ensure that everything works.
- Code formatters like YAPF and Black will automatically format your code to community standards, making it easier to read and debug.
- Code linters like Pylint and Flake8 can help you correct both programmatic and stylistic problems.
- You can use the new.py program to generate new Python programs that use `argparse`.