1. How to write and test a Python program

Before we start writing the exercises, I want to discuss how to write programs that are documented and tested using the following principles. Specifically, we're going to:

- Write our first Python program to say "Hello, World!"
- Handle command-line arguments using `argparse`
- Run tests for your code with `pytest`
- Learn about `$PATH`
- Use tools like `yapf` or `black` to format your code
- Use tools like `flake8` and `pylint` to find problems in your code
- Use the `new.py` program to create new programs

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 towards making a version that will greet a name that is passed as an argument. It will also print a helpful message when we ask for it, and we're going to use tests to make sure it does everything correctly. In the `01_hello` directory, you'll see there several versions of a "hello" program we'll write. There is also a program called `test.py` that we're going use to test the program.

Start off by creating a text file called `hello.py` in that directory. If you are working in VSCode 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:

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

It's time to run our new program! You can open a terminal window in VSCode or PyCharm or some other terminal. Be sure to navigate to the directory where your `hello.py` program is located. We can run it with the command `python3 hello.py` to have Python version 3 execute the commands in the file called `hello.py`. You should see this:

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

Here is how it looks in the repl.it interface:

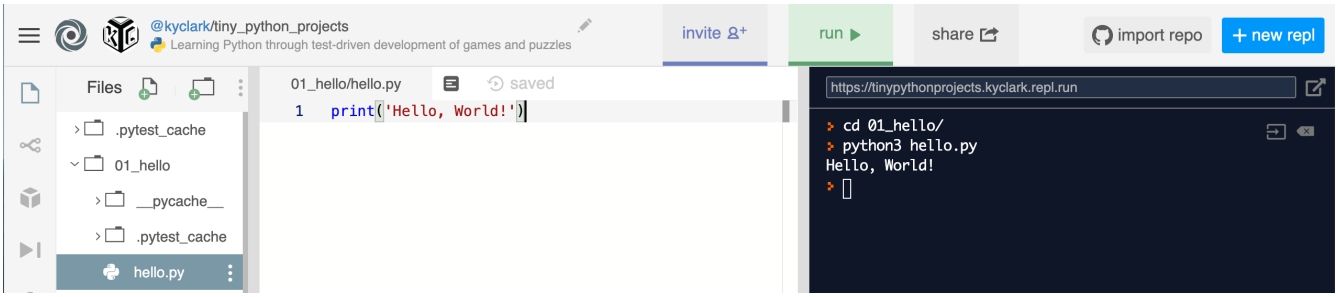


Figure 1. 1. Writing and running our first program using repl.it.

If that was your first Python program, congratulations!

1.1. Comment lines

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

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

If you run it 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.

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

If you get the error `pytest: command not found`, then you need to install the `pytest` module. Refer to the "Installing modules" section of the introduction.

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

test.py::test_exists PASSED                                              [ 20%] ①
test.py::test_runnable PASSED                                            [ 40%] ②
test.py::test_executable FAILED                                          [ 60%] ③
test.py::test_usage FAILED                                               [ 80%] ④
test.py::test_input FAILED                                               [100%] ⑤

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

① The first test always checks that the expected file exists. Here the test was looking for `hello.py`.

② The second test tried to run the program with `python3 hello.py` and then checked if the program printed "Hello, World!" If you miss even one character like forgetting the comma, then the test will point out the error, so read carefully!

③ The third test checks that the program is "executable." This test failed, and so next we're going to talk about how to make that pass.

④ The fourth test asked the program for help and didn't get anything. We're going to add the ability to print a "usage" statement that describes how to use our program.

⑤ 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.

I've written all the tests in an order that I hope helps you to write the program in a logical fashion. If you can'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 two 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%]  ①

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

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

        out = getoutput({prg})
>       assert out.strip() == 'Hello, World!'                                    ②
E       AssertionError: assert '/bin/sh: ./h...ission denied' == 'Hello, World!' ③
E         - /bin/sh: ./hello.py: Permission denied                               ④
E         + Hello, World!                                                        ⑤

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

① Notice that this test fails and so no more tests are run. This is because we ran `pytest` with the `-x` option.

② The `>` at the beginning of this line shows the source of the subsequent errors.

③ The `E` at the beginning of this line show that this is an "Error" you should read. The `AssertionError` is saying that `test.py` program is trying to execute the command `./hello.py` to see if it will produce the text "Hello, World!"

④ The `-` character is showing the actual output from the command is "Permission denied".

⑤ The `+` character is showing the test expected to get "Hello, World!"

Let's talk about how to fix this error!

1.3. Adding the shebang line

One thing we have learned about Python programs is that they live in plain text files that we 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 the language that 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 "shuh" and the `!` as the "bang!"). Just as with any other comment, Python will ignore the shebang, but the operating system (like Mac 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:

```
1 #!/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 which, of course, will have different values from mine, but we both (probably) have both of these ideas.

We can use the `env` command to find and run programs. If you run `env python3`, it will run the `python3` program if it can find one. Here's is 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 our environment. If Python has not been installed, then it won't be able to find it, but it also might be the case that Python has been installed more than once! We 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, we 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 above, 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
```

Then my program would not work when I run it on another computer that has `python3` installed in a different location. I doubt it would work on your computer, either. This why we 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:

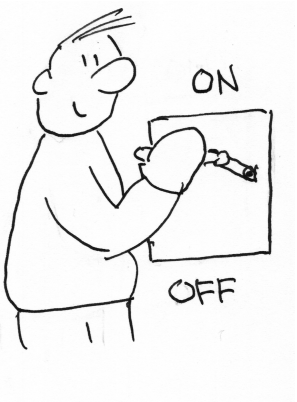
```
1 #!/usr/bin/env python3 ①
2 # Purpose: Say hello   ②
3 print('Hello, World!') ③
```

① The shebang line telling the operating system to use the `/usr/bin/env` program to find `python3` to interpret this program.

② A comment line documenting the purpose of the program.

③ A Python command to print some text to the screen.

1.4. 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 is that we could copy our program to a place where other programs live and execute it from anywhere on our computer!

To do this, the first step is to make our program "executable" using the command `chmod` (*change mode*). Think of it like turning your program "on." Run this command to make `hello.py` executable:

```
$ chmod +x hello.py ①
```

① The `+x` will *add* an "executable" attribute to the file.

Now you can run the program like so:

```
$ ./hello.py ①
Hello, World!
```

① The `./` is the current directory and is necessary to run a program when you are in the same directory as the program.

1.5. Understanding $PATH

One of the biggest reasons to set the shebang line and make your program executable is so that you can install and 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, Mac, and Linux all have a `$PATH` variable that is a list of directories where the OS will look in to find a program. For instance, here is the `$PATH` 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 the colon (`:`). Notice that the directory where `python3` lives is the first one in the `$PATH`. It's a pretty long string. so I broke it with the `\` character to make it easier to read. If you copy your `hello.py` program to any of the directories listed in your `$PATH`, and then you can execute the 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 upperleftmost 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 bathroooms and bedroom closets? Or would you start off by looking in places like the keyhooks beside the front door and then move on to search the pockets of your favorite jacket and your purse or backpack and then maybe under the couch cushions and so forth?

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

Often we may find ourselves working on a computer that won't allow us 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 can create this directory on the repl.it machine, copy a program to it, and then add it to my `$PATH`:

```
$ mkdir ~/bin                ①
$ cp 01_hello/hello.py ~/bin ②
$ PATH=~/bin:$PATH           ③
$ which hello.py             ④
/home/runner/bin/hello.py    ⑤
```

① Use the `mkdir` ("make directory") command to create `~/bin`.

② Use the `cp` command to copy the `01_hello/hello.py` program to the `~/bin` directory.

③ Put the `~/bin` directory first in our `$PATH`.

④ Use the `which` command to look for the `hello.py` program. If the steps above worked, the OS should now be able to find the program in one of the directories listed in our `$PATH`.

Now I can be in any directory:

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

And I can run it:

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

While the shebang and the executable stuff all 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 create one for our program as it is, there are no inputs and the output is always "Hello, World!"

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* change the code to:

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

But that would mean we'd have to change the code everytime we wanted to make it greet a different name. We'd like to have a way 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 ①

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

① Note that the `name` is called a *positional* argument.

To do this, we're going to use the `argparse` module. Modules are files of code we can bring into our programs. We can also learn to create modules to share our code with other people. There are hundreds to thousands of modules that 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 to look like the below. I really recommend you type everything yourself and don't copy and paste:

```
1 #!/usr/bin/env python3 ①
2 # Purpose: Say hello   ②
3
4 import argparse        ③
5
6 parser = argparse.ArgumentParser(description='Say hello') ④
7 parser.add_argument('name', help='Name to greet')         ⑤
8 args = parser.parse_args()                                ⑥
9 print('Hello, ' + args.name + '!')                       ⑦
10
11 #!/usr/bin/env python3
12 # Purpose: Say hello
13
14 import argparse
15
16 parser = argparse.ArgumentParser(description='Say hello')
17
18 args = parser.parse_args()
19 print('Hello, ' + args.name + '!')
```

① The shebang line tells the OS which program to use to execute this program.

② This comment documents the purpose of the program.

③ We must import the `argparse` module in order to use it.

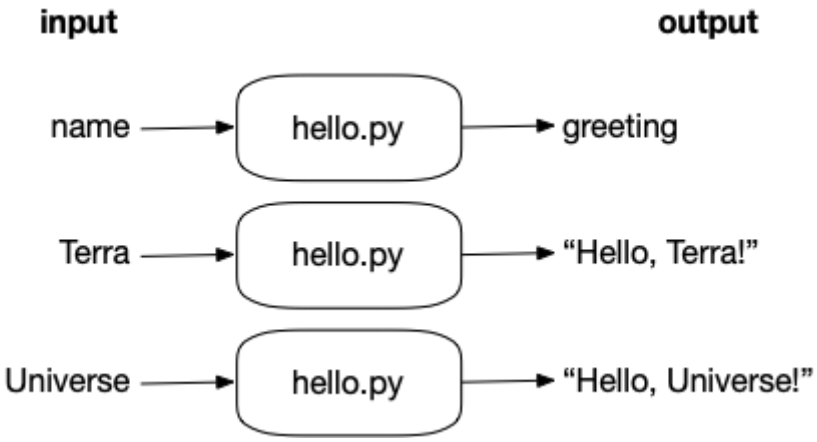
④ The `parser` will figure out all the arguments. The `description` appears in the help message.

⑤ We need to tell the parser to expect a `name` that will be the object of our salutations.

⑥ We ask the `parser` to parse any arguments to the program.

⑦ Print the greeting using the `args.name` value.

Here is 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              ①
usage: hello.py [-h] name ②
hello.py: error: the following arguments are required: name ③
```

① We run the program with no arguments, but the program now expects a single argument (a "name").

② Since the program doesn’t get the expected argument, it stops and prints a "usage" to let the user know how to properly invoke the program

③ The error message tells the user that they have not supplied a required parameter called `name`.

We changed our 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 `-h` and `--help` arguments and verify that you see the help messages. Our 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

What if I’d like run my 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',        ①
                    default='World', help='Name to greet')
args = parser.parse_args()
print('Hello, ' + args.name + '!')
```

① The only change to this program is to add `-n` for the "short" and `--name` for the "long" option names. We also indicate a `default` value. The `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:

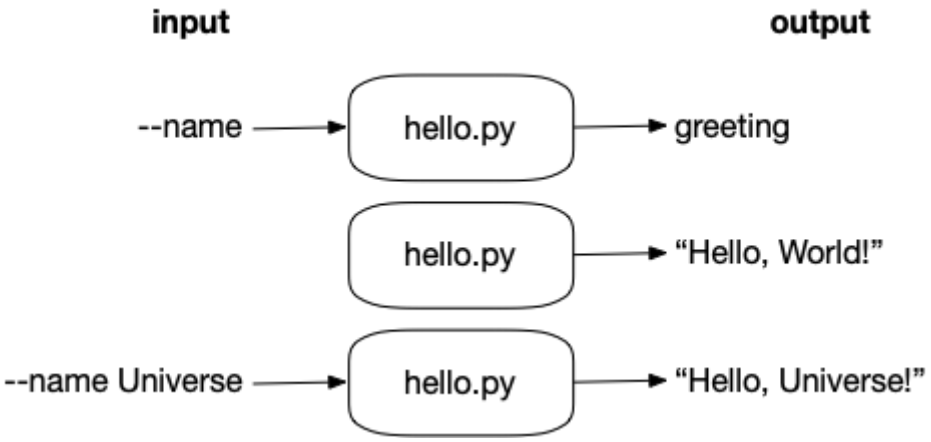
```
$ ./hello.py --name Terra
Hello, Terra!
```

And now 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 ①
```

① Notice that the argument is now an *option* 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.

Here is a string diagram that describes our program:



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 may have default values. Parameters that *don’t* start with dashes are "positional" and are usually required and so 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 close 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 `argparse` allows us to accept arguments when our program runs and will also create documentation for 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 then there is an idiom in Python programs to call the `main()` function at the end of the code like this:

```
1 #!/usr/bin/env python3
2 # Purpose: Say hello
3
4 import argparse
5
6 def main():                ①
7     parser = argparse.ArgumentParser(description='Say hello')
8     parser.add_argument('-n', '--name', metavar='name',
9                         default='World', help='Name to greet')
10    args = parser.parse_args()
11    print('Hello, ' + args.name + '!')
12
13 if __name__ == '__main__': ②
14    main()                  ③
```

① The `def` is to "define" a function. The name of the function is `main()`. The empty parentheses show that this function accepts no arguments.

② Every program or module in Python has a name which can be accessed through the variable `__name__`. When the program is executing, then `__name__` is set to "__main__". [1]

③ If this is true, then call the `main()` function.

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

1.11. Adding the `get_args()` function

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

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 how the program looks now:

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

import argparse

def get_args():                 ①
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('-n', '--name', metavar='name',
                        default='World', help='Name to greet')
    return parser.parse_args() ②

def main():                     ③
    args = get_args()           ④
    print('Hello, ' + args.name + '!')

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

① The `get_args()` function dedicated to getting the arguments. All the `argparse` code now lives here.

② We need to call `return` to send the results of parsing the arguments back to the `main()` function.

③ The `main()` function is much shorter now.

④ Call the `get_args()` function to get parsed arguments. If there is a problem with the arguments or if the user asks for `--help`, then the program never gets to this point because `argparse` will cause it to exit. If our program does make it this far, then 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()`.

1.11.1. Checking style and errors

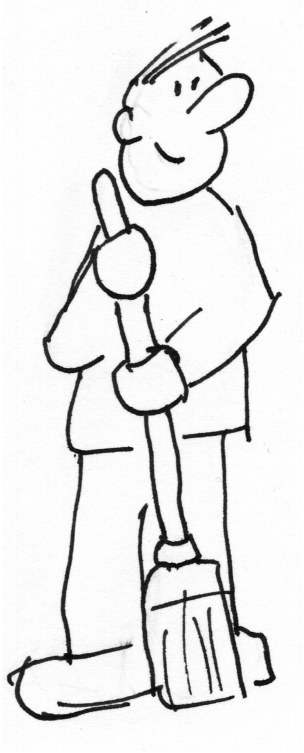


Figure 1. 2. `pylint` makes your code linty fresh!

Our program works really well now. We can use tools like `flake8` and `pylint` tools 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 or comment 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 multi-line 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
"""                                                    ①
Author:  Ken Youens-Clark <kyclark@gmail.com>
Purpose: Say hello
"""

import argparse


# -------------------------------------------------- ②
def get_args():
    """Get the command-line arguments"""              ③

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


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

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


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

① Triple-quoted, multi-line 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, the purpose of the script so that any future person using my 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.

② A big horizontal "line" comment to help me find the functions. You can omit these if you don't like them.

③ 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.

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 VSCode 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. Because this is something you will do literally hundreds of times, 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
```

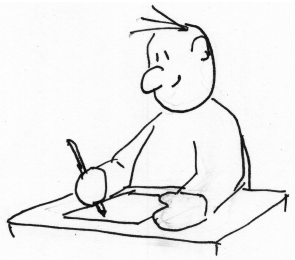
If you have the program `make` installed on your computer, then you can run `make test` when you are in the `01_hello` directory. The `make` program will look for a `Makefile` in your current working directory and then look for a recipe called "test." Here it will find that the command to run for the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

```
$ 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`, then you can always run `pytest -xv test.py` yourself. They both accomplish the same task.

The bigger point to make, though, is that we were able to use our tests to verify that our program still does exactly what it was supposed to do. As you write these programs, you may want to try different solutions. The tests give you the freedom to rewrite your programs and still know that they are working.

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 program. You'll be using `argparse` in every program for this book, and you'll learn about how we can use it to convert text to numbers and validate and open files and much more. There are so many options that I created a Python program called `new.py` that helps you start writing new Python programs that use `argparse`.

I have put my `new.py` program into the `bin` directory of the GitHub repo. I suggest you start every new program with this program. For instance, you could create a new version of `hello.py` using the `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` will not overwrite an existing file unless we tell it to, so you can use this without worrying that you might erase your work. Try using it to create a different program 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 new program:

```
 1 #!/usr/bin/env python3 ①
 2 """                    ②
 3 Author : Ken Youens-Clark <kyclark@gmail.com>
 4 Date   : 2020-02-28
 5 Purpose: Rock the Casbah
 6 """
 7
 8 import argparse        ③
 9 import os
10 import sys
11
12
13 # --------------------------------------------------
14 def get_args():       ④
15     """Get command-line arguments"""
16
17     parser = argparse.ArgumentParser(
18         description='Rock the Casbah',
19         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
20
21     parser.add_argument('positional', ⑤
22                         metavar='str',
23                         help='A positional argument')
24
25     parser.add_argument('-a',         ⑥
```

```
26                          '--arg',
27                          help='A named string argument',
28                          metavar='str',
29                          type=str,
30                          default='')
31
32      parser.add_argument('-i',           ⑦
33                          '--int',
34                          help='A named integer argument',
35                          metavar='int',
36                          type=int,
37                          default=0)
38
39      parser.add_argument('-f',           ⑧
40                          '--file',
41                          help='A readable file',
42                          metavar='FILE',
43                          type=argparse.FileType('r'),
44                          default=None)
45
46      parser.add_argument('-o',           ⑨
47                          '--on',
48                          help='A boolean flag',
49                          action='store_true')
50
51      return parser.parse_args()          ⑩
52
53
54  # --------------------------------------------------
55  def main():         ⑪
56      """Make a jazz noise here"""
57
58      args = get_args()           ⑫
59      str_arg = args.arg          ⑬
60      int_arg = args.int
61      file_arg = args.file
62      flag_arg = args.on
63      pos_arg = args.positional
64
65      print(f'str_arg = "{str_arg}"')
66      print(f'int_arg = "{int_arg}"')
67      print('file_arg = "{}"'.format(file_arg.name if file_arg else ''))
68      print(f'flag_arg = "{flag_arg}"')
69      print(f'positional = "{pos_arg}"')
70
71
72  # --------------------------------------------------
73  if __name__ == '__main__':    ⑭
74      main()                    ⑮
```

① The shebang line should use the `env` program to find the `python3` program.

② This docstring is for the program as a whole.

③ These lines import various modules the program needs.

④ The `get_args()` function is responsible for parsing and validating arguments.

⑤ Define a "positional" argument like our first version of `hello.py` that had a `name` argument.

⑥ Define an "optional" argument like when we changed to the `--name` option.

⑦ Define an optional argument that must be an integer value.

⑧ Define an optional argument that must be a file.

⑨ Define a "flag" option that is either "on" when present or "off" when absent. We'll learn more about these later.

⑩ Return the parsed arguments to `main()`. If there are any problem like the `--int` value was some text rather than a number like `42`, then `argparse` will print an error message and the "usage" for the user.

⑪ Define the `main()` function where the program starts.

⑫ The first thing our `main()` functions will always do is to call `get_args()` to get the arguments.

⑬ Each argument's value is accessible through the "long" name of the argument. It is not a requirement to have both a short and long name, but it is common and tends to make your program more readable.

⑭ When the program is being executed, the `__name__` value will be equal to the text "__main__".

⑮ If the condition is true, then call the `main()` function.

The arguments that this program will accept are:

1. 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.
2. An automatic `-h` or `--help` flag that will cause `argparse` to print the usage.
3. A string option called either `-a` or `--arg`
4. A named option argument called `-i` or `--int`
5. A file option called `-f` or `--file`
6. A boolean (off/on) flag called `-o` or `--on`

Looking at the above, we can see that they `new.py` program has done the following for you:

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

The result is a program that you can immediately execute and which will produce documentation on how to run it. After you use `new.py` to start 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 the "Crow's Nest" chapter, you can 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`, then I have included a sample of the above program as `template/template.py` that you can copy. For instance, in the "Crow's Nest" chapter you should create the program `02_crowsnest/crowsnest.py` Either you can do this with `new.py` from the top level of the repository:

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

Or 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 I don't want you to have to start every program from scratch! I think it's much easier to start from a complete, working program and modify it.

You could copy the `new.py` program to your `~/bin` directory, and then you would be able to use it from any directory to create a new program.

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

1.15. Summary

- A Python program is plain text that lives in a file. We 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 the `env` program 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`.

[1] See https://docs.python.org/3/library/*main*.html for more information.