Chapter 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:

- Process and document command-line arguments using `argparse`
- Write and run tests for your code with `pytest`
- Use tools like `yapf` or `black` to format your code
- Annotate variables and functions with type hints and then use `mypy` to check correctness
- Use tools like `flake8` and `pylint` to find problems in your code

How to write the world's best "Hello, World!"

It's pretty common to write "Hello, World!" as your first program in any language, so let's start there. We're going to start as simply as possible and end with a program you'd be proud to submit for a code review.

In your repo, go into the `hello` directory. There you'll see 14 versions of the "hello" program we'll write along with the `test.py` we'll use to test the final version. Start off by creating a text file called `hello.py` in that directory and add this line:


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

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!
```

If that was your first Python program, congratulations! You've done well, and we're all really proud of you.

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:

```
# Purpose: Say hello
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.

The shebang

We could write similar programs in other languages like bash, Ruby, and Perl, so it's common to document the language inside the program with a special comment character combination of `#!`. The nickname for this is "shebang" (pronounced "shuh-bang" — I always think of the `#` as "shuh" and the `!` as the "bang!"). As usual, Python will ignore this, but the OS will use it to decide which program to use to interpret the rest of the file.

Here is the shebang you should add:

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

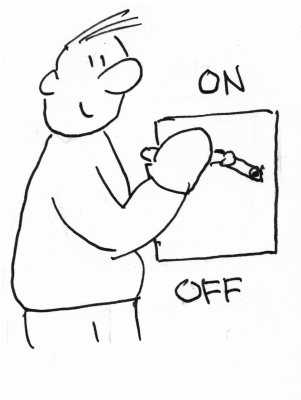
The `env` program will tell you about your "environment." If you run it like `env`, you should see lots of lines of output with lines like `USER=kyclark`. If you run it like `env python3`, it will search for and run the `python3` program. On my systems, I like to use the Anaconda Python distribution, and this is usually installed in my "home" directory. On my Mac laptop, this means that my `python3` is actually `/Users/kyclark/anaconda3/bin/python3`, but on another system like one of my Linux web servers will be an entirely different location. If I hardcoded the Mac location of `python3` in the shebang line, then my program won't work when I run it on Linux, so I always use the `env` program to find `python3`.

Now your program should look like this:

```
#!/usr/bin/env python3  1
# Purpose: Say hello    2
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.

Making a program executable



Now I'd like to "run" the script. On a Unix-like system, we can make any file "executable" with the command `chmod` (*change mode*). Think of it like turning your program "on." Run this command to make `hello.py` executable — to *add* (`+`) the *executable* (`x`) bit:

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

Now you can run the program like so:

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

That looks way cooler.

String diagrams

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!"

Adding a parameter

It's not terribly interesting for our program to always say "Hello, World!" Let's have it enthusiastically greet some name that we will pass as an *argument*. That is, we'd like our program to work like this:

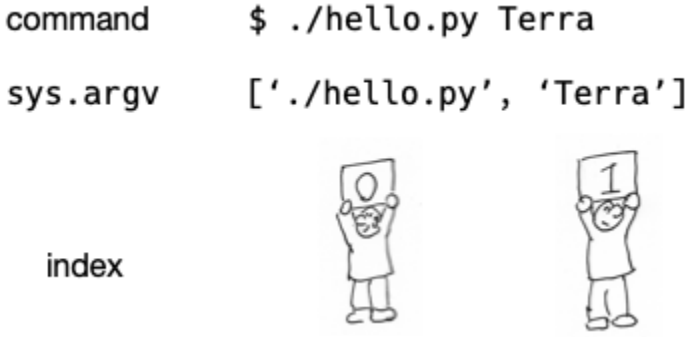
```
$ ./hello.py Terra
Hello, Terra!
```

The arguments to a program are available through the Python `sys` (system) module's `argv` (argument vector). We have to add `import sys` to our program to use it. Change your program so it looks like the code below. Don't worry if you don't understand what you're typing — we'll cover all this in detail later!

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys                    1
 5
 6  args = sys.argv[1:]           2
 7  if args:                      3
 8      name = args[0]            4
 9      print(f'Hello, {name}!')  5
10  else:                         6
11      print('Hello, World!')    7
```

1. Allows us to use code from the `sys` module.
2. `argv` is a list, slice from index 1.
3. If there are any arguments,
4. Then get the `name` from index 0.
5. Print `name` using an f-string,
6. Otherwise,
7. Print the default greeting.

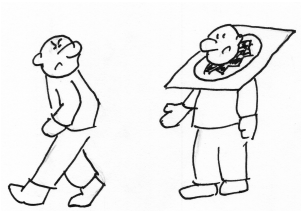
Line 6 probably looks a bit funny, so let's talk about that. The argument vector `argv` is a record of how the program was run starting with the path to the program itself and followed by any arguments. If you run it like `./hello.py Terra`, then `argv` will look like this:



Python, like so many other programming languages, uses the *index* `0` for the first element in a list. The element at `argv[0]` will always be the path to the program itself.

So on line 6, we use `sys.argv[1:]` to say we want a *slice* of that list starting from the index `1` and going to the end of the list and assign that to the variable called `args`. On line 7, we check if there is something in `args` — that is, there were some arguments to the program. If so, on line 8 we assign the variable `name` to the first element in `args`. If there are no `args`, we will print the default greeting on line 11. In the "Picnic" exercise, we'll talk much more about lists and slices.

Now our program will say "Hello, World!" when there are no arguments:



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

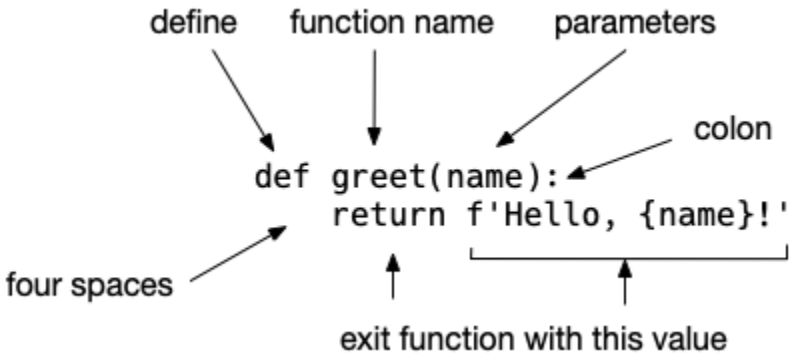
Or will extend warm salutations to whatever value is given as an argument:

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

Writing and testing functions

Our `hello.py` program now behaves differently depending on how it is run. How can we *test* our program to be sure it does the right thing? Let's find the part of our program that is variable and put that into a smaller unit of code that we can test. The unit will be a *function*, and the thing that changes is the greeting that we produce.

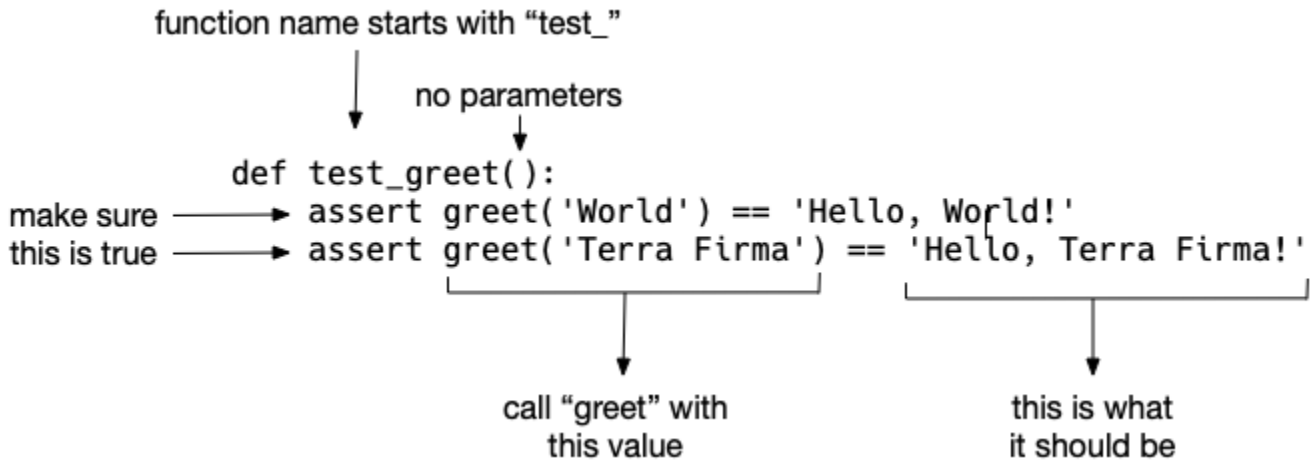
Create a function called `greet` with the `name` as a parameter.



The `def` is to *define* a function, and the function name follows after that. The function's parameters (the things that can change when it is run) go inside the parentheses. Here we have just one, and it's called `name`. The colon (`:`) lets Python know that a "block" of statements will follow, all of which need to be indented to the same level. Indentation can be done with tabs (pressing the `Tab` key) or using spaces, but you are not allowed to use a mix of tabs and spaces. Whatever you choose, be consistent. I use 4 spaces which seems to be in line with community standards.

The `return` will leave the function, optionally returning any value that follows. If no value follows the `return`, then the special value `None` will be returned to the caller. Some languages will return the last value of the function, but not Python. You must explicitly `return` what you want. You may return multiple values by separating them with commas, and you are also allowed to have more than one `return` in a function.

Notice that our new `greet` function does not `print` the greeting to the user. Its only job is to create the greeting so that we can write a test for it like so:



Here is what our whole program looks like now:

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

import sys

def greet(name):                    1
    return f'Hello, {name}!'

def test_greet():                   2
    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'

args = sys.argv[1:]                 3
name = args[0] if args else 'World' 4
print(greet(name))                  5
```

1. A function to create a greeting.
2. A function to test the `greet` function.
3. The program starts here.
4. An `if` expression is a shorthand way to write `if`/`else` that fits on one line. The `name` will be set to `args[0]` if is something in `args`, otherwise use the value `'World'`. We'll talk more about these in chapter 2 (Crow's Nest) when we are dealing with *binary* (two) choices.
5. The `print` and `greet` functions are *two separate ideas*. I know that I can count on Python to reliably `print`, but I need to test my `greet` function to ensure it works properly.

I personally like to use the `pytest` module to run my tests, but you may prefer to use the `unittest` module, especially if you have a background in Java. `pytest` will execute every function that has a name starting with `test_`, so I call my function `test_greet`. This function will call the `greet` function two times, once with the argument `'World'` and once with the argument `'Terra Firma'` to `assert` that the value returned from the function is what I would expect.

To see the tests in action, run `pytest -xv hello.py`:

```
$ pytest -xv hello.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.6.5, py-1.8.0, pluggy-0.12.0 -- /Users/kyclark/anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/kyclark/work/manning/tiny_python_projects/hello
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-2.7.1
collected 1 item

hello.py::test_greet PASSED                                              [100%]

========================== 1 passed in 0.03 seconds ===========================
```

The `-v` flag is to have `pytest` run in the "verbose" mode, and the `-x` flag to `pytest` tells it to stop at the first failing test. It's common to combine short flags in this way (`-xv` or `-vx`, the order doesn't matter). Here the `-x` flag causes `pytest` to stop on the first failing test. I include a `Makefile` with each exercise so that you can run `make test` to execute this for you. If you don't have `make` or don't want to use it, execute `pytest -xv test.py`.

Adding a `main` function

Our program is pretty good, but it's not quite idiomatic Python. Do you notice how we have two functions and then some other code just sort of hanging out (lines 13-15) flush left? It's good practice to put *all* your code inside functions, and it's very common to create a function called `main` where your program starts:

```
def main():
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))
```

We still have to tell Python to run this function, though, and the idiom for that in Python is to put these as the *last* two lines in your program:

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

Python reads your whole program from top to bottom. When it gets to these last lines, it will see if the special variable called `__name__` (one of the many special "double-under" or "dunder" variables) shows that we are in the "main" namespace. If you use `import` to bring your code into another piece of code, then it would *not* be in the "main" namespace. When you execute the code like a program, then then the namespace will be "main" and so the `main` function will be run.

Here is our whole program now:

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

import sys

def greet(name):
    return f'Hello, {name}!'

def test_greet():
    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'

def main():                 1
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))

if __name__ == '__main__':  2
    main()                  3
```

1. A new `main` entry function.
2. Program starts here.
3. Call "main" function if in "main" namespace

Note that the call to run `main()` must occur in the file *after* the definition of the `main` function. Python will throw an exception if you try to call a function that hasn't yet been defined. Try moving these to the *top* of your program and running it again, and you'll see an error:

```
NameError: name 'main' is not defined
```

Why is testing important?

I love to bake, especially breads and cakes. A simple but effective test for cakes is to insert a toothpick in the middle. If the toothpick comes out with gooey batter stuck to it, then the cake it not done. If it comes out clean, it's ready. When I'm cooking bread, I usually insert a thermometer to measure the internal temperature. It's really important to tests like these so you don't serve undercooked food!

Ask yourself if you would you fly on a plane or ride on an elevator if you knew it had never been tested? While lives may not depend so directly on the software you write, you still want to write code that is free from errors.

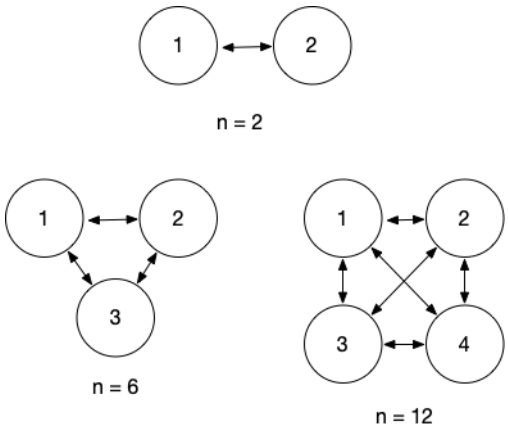


Figure 1. Complexity increases quickly!

As programs get longer, the tendency is that they get much harder to maintain. It's like adding a new team member. When there are just two people, there are only two ways to talk. Every new member increases the number of channels exponentially. Each new line of code or function you write interacts with all the other code in ways that become far too complicated for you to remember.

Tests help us check that our code still does what we think it does. Imagine we want to translate our `hello.py` to Spanish. We need to change the "Hello" to "Hola," but we misspell it "Halo":

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

import sys

def greet(name):
    return f'Halo, {name}!'                 1

def test_greet():
    assert greet('World') == 'Hola, World!'  2
    assert greet('Terra Firma') == 'Hola, Terra Firma!'

def main():
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))

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

1 "Hola" is misspelled as "Halo."

2 Check for the correct spelling.

If you run `pytest` on this, you will see this failure:

```
================================== FAILURES ===================================
_____________________________________ test_greet ______________________________________

    def test_greet():
>       assert greet('World') == 'Hola, World!'
E       AssertionError: assert 'Halo, World!' == 'Hola, World!'
E         - Halo, World!
E         ?   --
E         + Hola, World!
E         ?  ++

hello07.py:10: AssertionError
========================== 1 failed in 0.07 seconds ===========================
```

I would encourage you to write many functions, each of which does a few things as possible. Each function should have a `test_`, either in the same source file (I usually place it immediately after the function it's testing) or in a separate file (which I usually call `unit.py` for "unit" tests).

Often I will even write my `test_` function *before* I write the function itself, and I will imagine how I would want the function to work under a variety of conditions. As a general rule, I usually will test a function with:

- No arguments
- One argument
- Several arguments
- Bad arguments

I will then test if the function works or fails as I expect it. For instance, what would you have `greet` do if given no arguments or given a *list* of arguments instead of one?

Adding type hints

Each variable in Python has a `type`. For instance, we execute `python3` on the command line to interact directly with the Python interpreter. You will see a `>>>` prompt that is waiting for you to type some Python code: Create a `name` variable set to "World":

```
>>> name = 'World'
```

Now use the `type` command to see how Python represents this data:

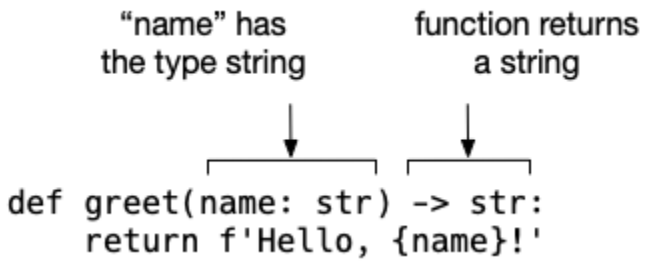
```
>>> type(name)
<class 'str'>
```

Anything in quotes (single or double) is a "string," which in Python is represented by the class `str`. Python has many other types, for instance, the number `10` is an `int` or "integer":

```
>>> type(10)
<class 'int'>
```

We can add "type hints" to our code to ensure that we use the right types of data, like not trying to divide an `int` by a `str` which would cause an *exception* and crash our code.

Shown is how `greet` would look with type hints. You can read `name: str` as "name has the type string." Additionally, the `greet` function itself has been annotated with `-> str` to indicate that it will return a `str` value. If a function (like `main`) returns nothing, add the special `None` as the return type.



Here is what the entire program looks like with type hints. ***Can you find the error?***

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import sys
 5
 6  def greet(name: str) -> str:  1
 7      return f'Hello, {name}!'
 8
 9  def test_greet() -> None:     2
10      assert greet('World') == 'Hello, World!'
11      assert greet('Terra Firma') == 'Hello, Terra Firma!'
12
13  def main() -> None:           3
14      args = sys.argv[1:]
15      print(greet(args))
16
17  if __name__ == '__main__':
18      main()
```

1 The `name` argument is a `str`, and the function returns a `str`.

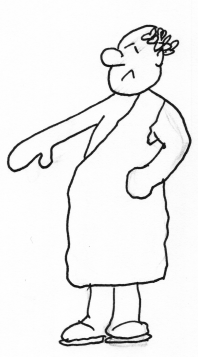
2 The function returns `None`.

3 The function returns `None`.

Did you find the error? What happens when you run this program? Read on to see how we can find the error.

Verification with `mypy`

Python itself completely ignores the type hints, but we can use the `mypy` program to find a problem in the code. If you don't have `mypy`, you can install it like so:



```
$ python3 -m pip install mypy
```

If we run it on `hello.py`, we see that it finds a type error:

```
$ mypy hello.py
hello.py:15: error: Argument 1 to "greet" has incompatible type "List[str]"; expected "str"
```

Can you see that we tried to pass `args` (which is a list of strings) to `greet` instead of a single `str` value? If you run the above, it will print the entire list `['World']` instead of the value inside the list:

```
$ ./hello.py World
Hello, ['World']!
```

This is not an error that our `test_greet` would have found because it's not a problem with the `greet` function but with *how the function is called*. Here is the correct `main` function:

```
def main() -> None:
    args = sys.argv[1:]
    name = args[0] if args else 'World'
    print(greet(name))
```

Manually validating and documenting arguments

We're now going to change our program to require exactly one argument. If we don't get that, we need to print a "usage" statement that explains how to use the program:

```
$ ./hello.py
usage: hello.py NAME
```

And likewise if run with more than one argument:

```
$ ./hello.py Terra Firma
usage: hello.py NAME
```

When given one argument, it should work as expected:

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

Here is the new version to make that work:

```
 1  #!/usr/bin/env python3
 2  # Purpose: Say hello
 3
 4  import os
 5  import sys
 6
 7  def greet(name: str) -> str:
 8      return f'Hello, {name}!'
 9
10  def test_greet() -> None:
11      assert greet('World') == 'Hello, World!'
12      assert greet('Terra Firma') == 'Hello, Terra Firma!'
13
14  def main() -> None:
15      args = sys.argv[1:]
16      if len(args) != 1:                              1
17          prg_name = os.path.basename(sys.argv[0])    2
18          print(f'usage: {prg_name} NAME')            3
19          sys.exit(1)                                 4
20      else:
21          print(greet(args[0]))
22
23  if __name__ == '__main__':
24      main()
```

1. `args` must have 1 element.
2. `sys.argv[0]` has program name. Use `basename` to remove the path.
3. Print "usage" statement.
4. Exit with error (not a 0 value).

Here I `import os` so that I can use the `os.path.basename` function to get the "basename" of `sys.argv[0]` which, you'll recall, is the path of the program itself. The "basename" of a path is the filename itself:

```
>>> import os
>>> os.path.basename('/path/to/hello.py')
'hello.py'
```

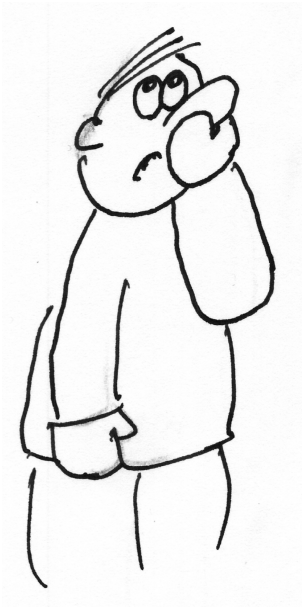
On line 15, I check if the `len` (length) of the `args` is *not* 1. If so, I'll print a "usage" statement on how to run the program. Note that I call `sys.exit(1)` to indicate a *non-zero exit value* because `0` (which is the default) indicates "zero errors." Any exit value that is not `0` indicates a failure.

Most Unix tools will also respond to `-h` or `--help` to show a "usage" statement. What happens when we try that?

```
$ ./hello.py --help
Hello, --help!
```

Well, that didn't work out. Just because we gave `--help` as an argument didn't mean that our program interpreted it correctly. Our program has been written to process *positional* arguments only, and the `--help` argument is a "flag," meaning a value that is `True` when present and `False` when absent. Here is a pretty ugly way to code this:

```
def main() -> None:
    args = sys.argv[1:]
    val = args[0] if args else ''
    if len(args) != 1 or val == '-h' or val == '--help':
        prg_name = os.path.basename(sys.argv[0])
        print(f'usage: {prg_name} NAME')
    else:
        print(greet(val))
```



Now we are setting a `val` ("value") variable equal to the first argument if one is present. Then we can check `len(args)` or if the `val` is equal to either the string `'-h'` or `'--help'`. I can verify that it works:

```
$ ./hello.py -h
usage: hello.py NAME
$ ./hello.py --help
usage: hello.py NAME
```

That big compound `if` test is pretty ugly. It's what I would call a "code smell" — it's trying to do too many things at once. We can write something much more elegant.

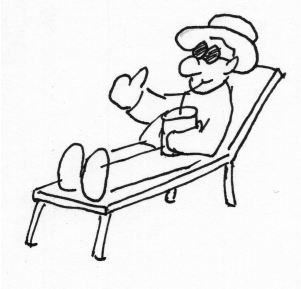
Documenting and validating arguments using `argparse`

The Python language has a standard module called `argparse` that will parse all the command line arguments, options, and flags. You have to invest a bit of time to learn it, but it will save you so much time in return. Don't worry if you don't understand this. Chapter 2 will show you many examples of how to use `argparse` that you can copy and paste for your own programs. By the end of this book, you will be an `argparse` pro!

Here is a new version that will make our code much cleaner:

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

import argparse

def greet(name: str) -> str:
    return f'Hello, {name}!'

def test_greet() -> None:
    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'

def main() -> None:
    parser = argparse.ArgumentParser(description='Say hello')  1
    parser.add_argument('name', help='Name to greet')          2
    args = parser.parse_args()                                 3
    print(greet(args.name))                                    4

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

1. Create argument parser.
2. Add the `name` parameter.
3. Get the parsed `args`.
4. Print the greeting given the `args.name` value.



Now instead of directly dealing with `sys.argv`, we describe to `argparse` that we want a single argument called `name` and let it do the hard work of parsing and validating the arguments. This will be a *positional* argument because `name` does *not* start with a dash. We will set `args` equal to the result of our `parser` doing the work to `parse_args`.

Now if you run the program with no arguments, the "usage" statement will be generated by `argparse`. Though you can't see it directly, the exit value is also set to something other than `0` to indicate failure because we failed to provide the required argument:

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

And both the `-h` and `--help` flags will trigger a longer help document that looks like a Unix `man` page.

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

Say hello

positional arguments:
  str         The name to greet

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

Adding `get_args`

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, and it makes the `main` function stay much shorter if this is separated. Getting the command-line arguments is a functional unit 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() -> argparse.Namespace:  1
    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('name', metavar='str', help='The name to greet')
    return parser.parse_args()

def main() -> None:
    args = get_args()                  2
    print(greet(args.name))

def greet(name: str) -> str:
    return f'Hello, {name}!'

def test_greet() -> None:
    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'

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

1 The `get_args` function dedicated to getting the command-line arguments

2 Call `get_args` function to get parsed arguments.

Checking style and errors

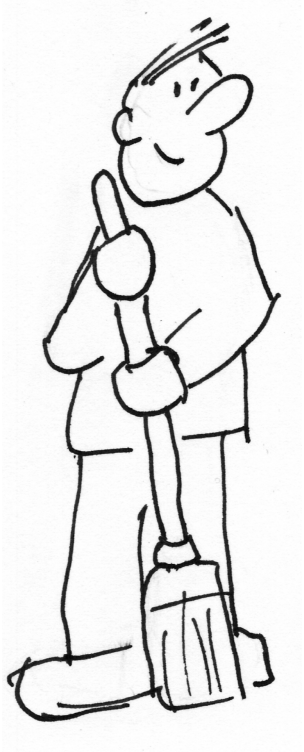
We now have the bones of a pretty respectable program. I like to use the `flake8` and `pylint` tools to give me suggestions on how to improve my programs. This is a process called "linting," and the tools are called "linters." You can use the `pip` module to install them like so:

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

I find that `flake8` is unhappy with readability as it tells me to put 2 lines after each function definition:

```
$ flake8 hello.py
hello.py:6:1: E302 expected 2 blank lines, found 1
hello.py:11:1: E302 expected 2 blank lines, found 1
hello.py:15:1: E302 expected 2 blank lines, found 1
hello.py:18:1: E302 expected 2 blank lines, found 1
hello.py:22:1: E305 expected 2 blank lines after class or function definition, found 1
```

The `pylint` program has other things to complain about, namely that my functions are missing documentation ("docstrings"):



```
$ pylint hello.py
************* Module hello
hello.py:1:0: C0111: Missing module docstring (missing-docstring)
hello.py:6:0: C0111: Missing function docstring (missing-docstring)
hello.py:11:0: C0111: Missing function docstring (missing-docstring)
hello.py:15:0: C0111: Missing function docstring (missing-docstring)
hello.py:18:0: C0111: Missing function docstring (missing-docstring)

-------------------------------------------------------------------
Your code has been rated at 6.67/10 (previous run: 6.67/10, +0.00)
```

A docstring is a string that occurs just after the `def` of the function. It can be a single line of text enclosed in single or double quotes. It's also common to use several lines, and Python allows you to use triple-quotes for strings that have line breaks: You can assign them to a variable:

```
>>> multi_line = """
... I should have been a pair of ragged claws.
... Scuttling across the floors of silent seas.
... """
```

Or you can use them in place of the `#` for multi-line comments. For instance, I usually document my whole program by putting a docstring just after the shebang. Inside I put my name, email address, the purpose of the script, and the date.

To fix the formatting issues, I ran the whole program through the formatting program called `yapf` (Yet Another Python Formatter). Another popular formatter is `black`. It really doesn't matter which one you choose, just choose one and use it consistently.

Here is a version that will silence all of our critics. Note that I like to add comments followed by dashes as visual separators between my functions, but this purely personal taste — you can omit these. I think this program is nicely formatted and more readable than previous versions:

```
#!/usr/bin/env python3
"""
Purpose: Say hello        1
Author:  Ken Youens-Clark
"""

import argparse


# -------------------------------------------------- 2
def get_args() -> argparse.Namespace:
    """Get command-line arguments""" 3

    parser = argparse.ArgumentParser(description='Say hello')
    parser.add_argument('name', metavar='str', help='The name to greet')
    return parser.parse_args()


# --------------------------------------------------
def main() -> None:
    """Start here"""

    args = get_args()
    print(greet(args.name))


# --------------------------------------------------
def greet(name: str) -> str:
    """Create a greeting"""

    return f'Hello, {name}!'


# --------------------------------------------------
def test_greet() -> None:
    """Test greet"""

    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'


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

1 Triple-quoted, multi-line docstring for program/module.

2 A big horizontal "line" to help me see the functions.

3 Triple-quotes can be used on a single line, too.

Making the argument optional

Let's return to making the argument to our program optional so that we can run with and without an argument. We'd like to run the program with no argument and have it default to using "World" for the `name` like so:

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

If we make `name` an *optional* argument, it can no longer be a *positional* argument. We create the option `--name` that will be followed with the *name* to greet:

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

The "short" name is a single dash and a single letter like `-n`, and the "long" name is two dashes followed by a longer name like `--name`. Here is what the usage looks like now:

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

Say hello

optional arguments:
  -h, --help          show this help message and exit
  -n str, --name str  The name to greet (default: World)
```

And here is the code. Note the use of the `formatter_class` to have `argparse` show the default values for arguments in the "usage" output:

```
#!/usr/bin/env python3
"""
Purpose: Say hello
Author:  Ken Youens-Clark
"""

import argparse


# --------------------------------------------------
def get_args() -> argparse.Namespace:
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Say hello',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)  1

    parser.add_argument('-n',             2
                        '--name',         3
                        default='World',  4
                        metavar='str',    5
                        help='The name to greet')

    return parser.parse_args()


# --------------------------------------------------
def main() -> None:
    """Start here"""

    args = get_args()
    print(greet(args.name))


# --------------------------------------------------
def greet(name: str) -> str:
    """Create a greeting"""

    return f'Hello, {name}!'


# --------------------------------------------------
def test_greet() -> None:
    """Test greet"""

    assert greet('World') == 'Hello, World!'
    assert greet('Terra Firma') == 'Hello, Terra Firma!'


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

1 Show default values in "usage."

2 Short option name

3 Long option name

4 The default value.

5 Hint to user of the data type.

Testing `hello.py`

Included in the `hello` directory is a `test.py` that will run the program to ensure it creates the correct output. We can run it either using `make test` or `pytest -xv test.py` [1] I will elide some of the output:

```
$ make test
pytest -xv test.py
============================= test session starts ==============================
...
collected 4 items

test.py::test_exists PASSED                                              [ 25%] 1
test.py::test_usage PASSED                                               [ 50%] 2
test.py::test_default PASSED                                             [ 75%] 3
test.py::test_input PASSED                                               [100%] 4

============================== 4 passed in 0.40s ===============================
```

1. The first test in every `test.py` checks to see if the expected program file exists. Here it is checking if `hello.py` exists in the same directory as the `test.py` program.
2. The second test always runs the program with `-h` and `--help` to see if it produces anything that looks like a "usage."
3. This test runs the program with no arguments to see if it produces the default output, `Hello, World!`
4. This test runs the program with the `-n` and `--name` options and different values to see if it accepts a `name` option and creates the correct output.

In the parlance of testing, the `test_greet` function that we wrote to test the `greet` function might be called a "unit test" as it tests one "unit" (a function) of code, while the `test.py` might be called an "integration test" because it's looking at the program as a whole from the outside to see if it performs correctly.

It may be a bit overwhelming to look at the `test.py` , but it will be instructive as the goal will be for you to learn how to write your own tests for your programs:

```
#!/usr/bin/env python3
"""tests for hello.py"""

import os
from subprocess import getstatusoutput, getoutput

prg = './hello.py'  1


# --------------------------------------------------
def test_exists():  2
    """exists"""

    assert os.path.isfile(prg)


# --------------------------------------------------
def test_usage():   3
    """usage"""

    for flag in ['-h', '--help']:
        rv, out = getstatusoutput(f'{prg} {flag}')
        assert rv == 0
        assert out.lower().startswith('usage')


# --------------------------------------------------
def test_default():  4
    """Says 'Hello, World!' by default"""

    out = getoutput(prg)
    assert out.strip() == 'Hello, World!'


# --------------------------------------------------
def test_input():    5
    """test for input"""

    for val in ['Universe', 'Multiverse']:
        for option in ['-n', '--name']:
            rv, out = getstatusoutput(f'{prg} {option} {val}')
            assert rv == 0
            assert out.strip() == f'Hello, {val}!'
```

1 This is a name of the expected program.

2 Tests are run in the order that they are defined in the program. This will be the first test to check if the `prg` exists.

3 Run the program with the `-h` and `--help` flags to see if the output looks like a "usage" statement.

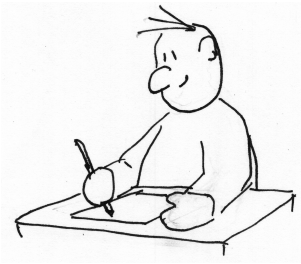
4 This test will run the program with no arguments to see if it prints `Hello, World!`

5 Run the program using both `-n` and `--name` with two different values to see if the program accepts the `name` option and prints the correct value.

In addition to running the integration `test.py` programs I have provided, I will suggest you include and run unit tests in your programs. I will also suggest writing your own functions and unit tests as well as extending the provided `test.py` to cover functionality that you add to your programs. I encourage you to study the `test.py` programs as you will learn as much or more from them as from the exercises you write!

Starting a new program with `new.py`

In my own practice, I almost never start writing a Python program from an empty page. I created a Python program called `new.py` that helps me start writing new Python programs. As most of my programs need to take parameters, I always use the `argparse` to interpret the command-line options. I have put my `new.py` program into the `bin` ("binaries" even though these are just text files) directory of the GitHub repo, and I recommend you start every new program with this program.



A central tenet of this book is to create *documented* and *testable* programs. Anything that *can* change about a program should be passed as an *argument* to the program when it is run. For instance, if a program will read an input file for data, the name of that file should be an argument like `-f input.txt`. Then the fact that the program takes an input file is now visible through the `--help` and we can pass in different files and test if the program processes the files correctly. It is *not* a requirement that you use `new.py` and `argparse`, however. As long as your programs process command-line arguments in the same way as `argparse` and always produce a usage on `-h` or `--help`, your programs should pass the test suites. The template that `new.py` provides is meant only to make it faster and more convenient to create new programs.

Here is how I would use `new.py` to create a new `hello.py` program:

```
$ new.py hello.py
Done, see new script "hello.py."
```

This is what will be produced:

```
#!/usr/bin/env python3
"""
Author : Ken Youens-Clark <kyclark@gmail.com>
Date   : 2019-10-21
Purpose: Rock the Casbah
"""

import argparse
import os
import sys


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

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

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

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

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

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

    parser.add_argument('-o',
                        '--on',
                        help='A boolean flag',
                        action='store_true')

    return parser.parse_args()


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

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

```
64
65      print('str_arg = "{}"'.format(str_arg))
66      print('int_arg = "{}"'.format(int_arg))
67      print('file_arg = "{}"'.format(file_arg.name))
68      print('flag_arg = "{}"'.format(flag_arg))
69      print('positional = "{}"'.format(pos_arg))
70
71
72  # --------------------------------------------------
73  if __name__ == '__main__':
74      main()
```

The arguments that this program will accept are:

1. A single positional argument of the type `str`. *Positional* means it is not preceeded 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 named string argument called either `-a` or `--arg`
4. A named interger argument called `-i` or `--int`
5. A named file argument called `-f` or `--file`
6. A boolean (off/on) flag called `-o` or `--on`

Each option here has both a "short" and "long" name. It is not a requirement to have both, but it is common and tends to make your program more readable.

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
```

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 next chapter you should create the program `crowsnest/crowsnest.py`. (That is, from the root directory of the repository, go into the `crowsnest` directory and create a file called `crowsnest.py`.)

Either you can do this with `new.py`:

```
$ cd crowsnest
$ new.py crowsnest.py
```

Or the `cp` (copy) command:

```
$ cp template/template.py 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.

Summary

- 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.
- Small, limited functions are easy to write `test_` functions which can be run by `pytest`.
- It is often helpful to write the tests before you write the functions to imagine how they ought to work.
- You should run your tests after any change to your program to ensure that everything still works.
- Code formatters like `yapf` and `black` will automatically format your code to community standards, making it easier to read and debug.
- The `mypy` tool can check for errors in your code that has been annotated with type hints.
- 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. The `make` program looks for a file called `Makefile` in the directory where we run `make`. Inside that file is a "target" called `test` that says the command that should be run is `pytest -xv test.py`. Makefiles are a way to create workflows of actions, each of which can depend on each other. I'm using a Makefile here to create a shortcut to run the tests.

Last updated 2020-01-15 15:49:17 -0700