> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

Hello

Let's use our familiar "Hello, World!" to get started:

```
$ cat -n hello.py
     1  #!/usr/bin/env python3
     2
     3  print('Hello, World!')
```

The first thing to notice is the "shebang" on line 1. I'm going to use `env` to find the first `python3` binary in the user's `$PATH`. In `bash`, we could use either `echo` or `printf` to print to the terminal (or a file). In Python, we have `print()` noting that we must use parentheses now to invoke functions.

Variables

It's not so interesting to just say "Hello, World!" all the time. Let's make a program that will say "hello" to some value that we pass in. This value can change each time we run the program, so it's common to call this a "variable."

Let's use the REPL (Read-Evaluate-Print-Loop, pronounced "reh-pul") to play with variables. Type `python` (or `python3` or `ipython`) to get into a REPL:

```
>>> name = 'Duderino'
>>> print('Hello,', name)
Hello, Duderino
```

Here I'm showing that we can create variable called `name` by assigning it some value like "Duderino." Unlike `bash`, we don't have to worry about spaces around the `=`. You can put any number of spaces around the equal sign, but it's most common (and readable) to put just one on each side. Notice that `print` will accept more than one argument and will put spaces between the arguments. You can tell it to use some other "separator" by indicating the `sep` keyword argument. Notice the Pythonic style is that there are *no* spaces around the `=` for keywords:

```
>>> print('Hello', name, sep=', ')
Hello, Duderino
```

It's not easy to tell, but `print` is also putting a newline on the end. We can change that with the `end` keyword argument:

```
>>> print('Hello', name, sep=', ', end='!')
Hello, Duderino!>>>
```

Unlike in `bash`, we cannot use a variable directly in a `print` statement or we get the equivalent of George Burns telling Gracie "Say 'Good night,' Gracie" and she says "Good night, Gracie!":

```
>>> print('Hello, name')
Hello, name
```

We could to use the `+` operator to concatenate it to the literal string "Hello,":

```
>>> print('Hello, ' + name)
Hello, Duderino
```

Types: Strings and Numbers

As you might expect, the "plus" operator `+` is also used to perform numeric addition:

```
>>> n = 10
>>> n + 1
11
```

The `name` variable above is of the type `str` (string) because we put the value in quotes (single or double, it doesn't matter).

```
>>> name = 'The Dude'
>>> type(name)
<class 'str'>
```

Numbers don't have quotes. Number can be integers (`int`) or floating-point numbers (`float`) if they have a decimal somewhere or you write them in scientific notation:

```
>>> type(10)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type(1.)
<class 'float'>
>>> type(2.864e-10)
<class 'float'>
```

The "plus" operator behaves completely differently with different *types* of arguments as long as the arguments are both strings or both numbers. Things go wobbly when you mix them:

```
>>> 'Hello, ' + 'Mr. Lebowski'
'Hello, Mr. Lebowski'
>>> 1 + 2
3
>>> 1 + 'Mr. Lebowski'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Lists

Before we go further, I will introduce a different variable type called a "list" as we are going to need that immediately. You create a list by putting values in `[]` (square brackets or just "brackets"):

```
>>> vals = ['foo', 'bar', 'baz']
>>> vals
['foo', 'bar', 'baz']
```

You can get the length of a list using the `len` function:

```
>>> len(vals)
3
```

But note that, like so many other languages, Python starts counting at `0`, so the first element is in the "zeroth" position. The value at position `1` is actually the *second* value.

```
>>> vals[0]
'foo'
>>> vals[1]
'bar'
```

We'll talk much more about lists in the next chapter. I needed to tell you that so I could tell you this next bit.

Command-line Arguments: sys.argv is a list

Now let's get our "hello" program to greet an argument passed from the command line. We discussed in the `bash` section that programs can take *positional* arguments, e.g. `ls` can accept the name of the directory you wish to list or `wc` can take the name of a file to count. *Positional* arguments mean the first argument, the second argument, and so on. In the command `ls ~`, the `~` (tilde which means `$HOME` in `bash`) is the one and only positional argument. In the command `ls /bin /usr/bin/`, there are two positional arguments, `/bin` and `/usr/bin/`.

Named options have some sort of prefix, e.g., `find` can take a `-maxdepth` argument to indicate how many levels deep to search. Lastly, commands may also take flags like the `-l` flags to `ls` that indicates you wish to see the "long" listing.

To get access to the positional arguments to our program, we need to `import sys` which is a package of code that will interact with the system. Those arguments will be a *list*:

```
$ cat -n hello_arg.py
     1    #!/usr/bin/env python3
     2
     3    import sys
     4
     5    args = sys.argv
     6    print('Hello, ' + args[1] + '!')
```

From the `sys` module, we call the `argv` function to get the "argument vector." This is a list, and, like `bash`, the name of the script is the first argument (in the zeroth position) – `args[0]`. That means the first *actual* "argument" to the script is in `args[1]`.

```
$ ./hello_arg.py Geddy
Hello, Geddy!
```

But there is a problem if we fail to pass any arguments:

```
$ ./hello_arg.py
Traceback (most recent call last):
  File "./hello_arg.py", line 6, in <module>
    print('Hello, ' + args[1] + '!')
IndexError: list index out of range
```

We tried to access something in `args` that doesn't exist, and so the entire program came to a halt ("crashed"). As in `bash`, we need to check how many arguments we have. Before I show you how to do that, let me explain something about *slicing* lists. Inside the `[]`, you can indicate a start and stop positon like so:

```
>>> vals
['foo', 'bar', 'baz']
>>> vals[1:]
['bar', 'baz']
>>> vals[1:2]
['bar']
```

In the last example, you see that the stop position is not inclusive. Even though you've seen that Python gets very upset by asking for a position in a list that does not exist, it has no problem giving you nothing when you ask for a *slice* that doesn't exist:

```
>>> vals[1000:]
[]
```

So we can use that to ask for `sys.argv[1:]` to get all the *actual* arguments to our program, skipping over the name of the program itself:

```
$ cat -n hello_arg2.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4
     5  args = sys.argv[1:]
     6
     7  if len(args) < 1:
     8      print('Usage:', sys.argv[0], 'NAME')
     9      sys.exit(1)
    10
    11  name = args[0]
    12  print('Hello, ' + name + '!')
```

If there are fewer than 1 argument, then we print a usage statement and use `sys.exit` to send the operating system a non-zero exit status, just like in `bash`. It works much better now:

```
$ ./hello_arg2.py
Usage: ./hello_arg2.py NAME
$ ./hello_arg2.py Alex
Hello, Alex!
```

Here is the same functionality but using some new functions, `str.format` so we can introduce a different way to join strings, and `os.path.basename` so we can get the name of the program without any leading path information like `./`:

```
$ cat -n hello_arg3.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4  import os
     5
     6  args = sys.argv[1:]
     7
     8  if len(args) != 1:
     9      script = os.path.basename(sys.argv[0])
    10      print('Usage: {} NAME'.format(script))
    11      sys.exit(1)
    12
    13  name = args[0]
    14  print('Hello, {}!'.format(name))
$ ./hello_arg3.py
Usage: hello_arg3.py NAME
$ ./hello_arg3.py Neil
Hello, Neil!
```

The main() thing

Many languages (e.g., Perl, Rust, Haskell) have the idea of a `main` module/function where all the processing starts. If you define a `main` function using `def main`, most people reading your code would understand that the program *ought* to begin there. (I say "ought" because Python won't actually make that happen. You still have to *call* the `main` function to make your program run!) I usually put my `main` first and then call it at the end of the script with this `__name__ == '__main__'` business. This looks a bit of a hack, but it is fairly Pythonic.

```
$ cat -n hello_arg4.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4  import os
     5
     6  def main():
     7      args = sys.argv[1:]
     8
     9      if len(args) != 1:
    10          script = os.path.basename(sys.argv[0])
    11          print('Usage: {} NAME'.format(script))
    12          sys.exit(1)
    13
    14      name = args[0]
    15      print('Hello, {}!'.format(name))
    16
    17
    18  if __name__ == '__main__':
    19      main()
$ ./hello_arg4.py
Usage: hello_arg4.py NAME
$ ./hello_arg4.py '2013 Rock and Roll Hall of Fame Inductees'
Hello, 2013 Rock and Roll Hall of Fame Inductees!
```

Function Order

Note that you cannot put call to `main()` before `def main` because you cannot call a function that hasn't been defined (lexically) in the program yet. To add insult to injury, this is a **run-time error** – meaning the mistake isn't caught by the compiler when the program is parsed into byte-code; instead the program just crashes.

```
$ cat -n func-def-order.py
     1    #!/usr/bin/env python3
```

```
     2
     3    print('Starting the program')
     4    foo()
     5    print('Ending the program')
     6
     7    def foo():
     8        print('This is foo')
$ ./func-def-order.py
Starting the program
Traceback (most recent call last):
  File "./func-def-order.py", line 4, in <module>
    foo()
NameError: name 'foo' is not defined
```

To contrast:

```
$ cat -n func-def-order2.py
     1    #!/usr/bin/env python3
     2
     3    def foo():
     4        print('This is foo')
     5
     6    print('Starting the program')
     7    foo()
     8    print('Ending the program')
$ ./func-def-order2.py
Starting the program
This is foo
Ending the program
```

Handle All The Args!

If we like, we can greet to any number of arguments:

```
$ cat -n hello_arg5.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4  import os
     5
     6  def main():
     7      names = sys.argv[1:]
     8
     9      if len(names) < 1:
    10          script = os.path.basename(sys.argv[0])
    11          print('Usage: {} NAME [NAME2 ...]'.format(script))
```

```
    12          sys.exit(1)
    13
    14      print('Hello, {}!'.format(', '.join(names)))
    15
    16  if __name__ == '__main__':
    17      main()
$ ./hello_arg5.py
Usage: hello_arg5.py NAME [NAME2 ...]
$ ./hello_arg5.py Geddy Alex Neil
Hello, Geddy, Alex, Neil!
```

Notice on line 14 to see how we can `join` all the arguments on a comma + space.

Conditionals

So far we've been using an `if` condition to see if we have enough arguments. If you want to test for more than one condition, you can use `elif` (else if) and `else` ("otherwise" or the "default" branch if all others fail). Here we'll use the `input` function to present the user with a prompt and get their input:

```
$ cat -n if-else.py
     1  #!/usr/bin/env python3
     2
     3  name = input('What is your name? ')
     4  age = int(input('Hi, ' + name + '. What is your age? '))
     5
     6  if age < 0:
     7      print("That isn't possible.")
     8  elif age < 18:
     9      print('You are a minor.')
    10  else:
    11      print('You are an adult.')
$ ./if-else.py
What is your name? Ken
Hi, Ken. What is your age? -4
That isn't possible.
$ ./if-else.py
What is your name? Lincoln
Hi, Lincoln. What is your age? 29
You are an adult.
```

On line 3, we can put the first answer directly into the `name` variable; however, on line 4, I need to convert the answer to an integer with `int` because I will need to compare it numerically, cf:

```
>>> 4 < 5
True
```

```
>>> '4' < 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
>>> int('4') < 5
True
```

Things go very badly if we blindly try to coerce a string into an `int`:

```
$ ./if-else.py
What is your name? Doreen
Hi, Doreen. What is your age? Ageless
Traceback (most recent call last):
  File "./if-else.py", line 4, in <module>
    age = int(input('Hi, ' + name + '. What is your age? '))
ValueError: invalid literal for int() with base 10: 'Ageless'
```

Later we'll talk about how to avoid problems like this.

Looping Over Lists

As in bash, we can use `for` loops in Python. Here's another way to greet all the people:

```
$ cat -n hello_arg6.py
     1  #!/usr/bin/env python3
     2
     3  import sys
     4  import os
     5
     6  def main():
     7      names = sys.argv[1:]
     8
     9      if len(names) < 1:
    10          prg = os.path.basename(sys.argv[0])
    11          print('Usage: {} NAME [NAME2 ...]'.format(prg))
    12          sys.exit(1)
    13
    14      for name in names:
    15          print('Hello, ' + name + '!')
    16
    17
    18  if __name__ == '__main__':
    19      main()
$ ./hello_arg6.py
Usage: hello_arg6.py NAME [NAME2 ...]
$ ./hello_arg6.py Salt Peppa
```

```
Hello, Salt!
Hello, Peppa!
```

You can use a `for` loop on anything that is like a list! A string is a list of characters:

```
>>> for letter in "abc":
...     print(letter)
...
a
b
c
```

The `range` function returns something that can be "iterated" like a list:

```
>>> for number in range(0, 5):
...     print(number)
...
0
1
2
3
4
```

Lists, of course:

```
>>> for word in ['foo', 'bar']:
...     print(word)
...
foo
bar
```

You can use the `str.split` function to split a string (the default is to split on spaces):

```
>>> for word in 'We hold these truths'.split():
...     print(word)
...
We
hold
these
truths
```

And we can use the `open` function to open a file and read each line using a `for` loop:

```
>>> for line in open('input1.txt'):
...     print(line, end='')
...
this is
some text
```

`from a file.`

The last example either needs to suppress the newline from `print` or do `rstrip()` on the line to remove it as the text coming from the file has a newline.

Stubbing new programs

Every program we've seen so far has had the same basic structure:

- Shebang
- import modules
- define main()
- call main()

Additionally we keep having to write the same few lines of code to get the arguments from `sys.argv[1:]` and then test that we have the right number and then print a usage and `sys.exit(1)`. Rather than type all that boilerplate or copy-paste from other programs, let's use a program to help us create new programs.

Included in the `bin` directory of the GitHub repo, there is a program called `new_py.py` that will stub out all this code for you. Make sure you either add that directory to you `$PATH` or copy that program into your existing `$PATH`, e.g., I like to have `$HOME/.local/bin` for programs like this:

```
$ which new_py.py
/Users/kyclark/.local/bin/new_py.py
```

Now run it with no arguments. As you might expect, it gives you a usage statement:

```
$ new_py.py
usage: new_py.py [-h] [-a] [-f] program
new_py.py: error: the following arguments are required: program
```

Give it the name of a new program (either with or without `.py`):

```
$ new_py.py foo
Done, see new script "foo.py."
$ cat -n foo.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : kyclark
     4  Date   : 2019-05-14
     5  Purpose: Rock the Casbah
     6  """
     7
     8  import os
     9  import sys
```

```
10
11
12  # --------------------------------------------------
13  def main():
14      args = sys.argv[1:]
15
16      if len(args) != 1:
17          print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
18          sys.exit(1)
19
20      arg = args[0]
21
22      print('Arg is "{}"'.format(arg))
23
24
25  # --------------------------------------------------
26  if __name__ == '__main__':
27      main()
```

What happens if you try to initialize a script when one already exists with that name?

```
$ new_py.py foo
"foo.py" exists.  Overwrite? [yN] n
Will not overwrite. Bye!
```

Unless you answer "y", the script will not be overwritten. You could also use the `-f|--force` flag to force the overwritting of an existing file.

Introducing argparse

Run `new_py.py` with `-h` or `--help` to see all the options:

```
$ new_py.py -h
usage: new_py.py [-h] [-a] [-f] program

Create Python script

positional arguments:
  program         Program name

optional arguments:
  -h, --help      show this help message and exit
  -a, --argparse  Use argparse (default: False)
  -f, --force     Overwrite existing (default: False)
```

Hey, what is `--argparse` about? Let's try it! I will combine the two short flag

`-a` and `-f` into `-fa` to "force" a new script that uses the `argparse` module to give us positional arguments, named arguments, and flags.

```
$ new_py.py -fa foo
Done, see new script "foo.py."
$ cat -n foo.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : kyclark
     4  Date   : 2019-05-14
     5  Purpose: Rock the Casbah
     6  """
     7
     8  import argparse
     9  import sys
    10
    11
    12  # --------------------------------------------------
    13  def get_args():
    14      """get command-line arguments"""
    15      parser = argparse.ArgumentParser(
    16          description='Argparse Python script',
    17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    18
    19      parser.add_argument(
    20          'positional', metavar='str', help='A positional argument')
    21
    22      parser.add_argument(
    23          '-a',
    24          '--arg',
    25          help='A named string argument',
    26          metavar='str',
    27          type=str,
    28          default='')
    29
    30      parser.add_argument(
    31          '-i',
    32          '--int',
    33          help='A named integer argument',
    34          metavar='int',
    35          type=int,
    36          default=0)
    37
    38      parser.add_argument(
    39          '-f', '--flag', help='A boolean flag', action='store_true')
    40
```

```
41      return parser.parse_args()
42
43
44  # --------------------------------------------------
45  def warn(msg):
46      """Print a message to STDERR"""
47      print(msg, file=sys.stderr)
48
49
50  # --------------------------------------------------
51  def die(msg='Something bad happened'):
52      """warn() and exit with error"""
53      warn(msg)
54      sys.exit(1)
55
56
57  # --------------------------------------------------
58  def main():
59      """Make a jazz noise here"""
60      args = get_args()
61      str_arg = args.arg
62      int_arg = args.int
63      flag_arg = args.flag
64      pos_arg = args.positional
65
66      print('str_arg = "{}"'.format(str_arg))
67      print('int_arg = "{}"'.format(int_arg))
68      print('flag_arg = "{}"'.format(flag_arg))
69      print('positional = "{}"'.format(pos_arg))
70
71
72  # --------------------------------------------------
73  if __name__ == '__main__':
74      main()
```

That's a good bit more code, but the advantage here is that we can now get quite detailed help documentation and very specific behavior from our arguments, e.g., one argument needs to be a string while another needs to be a number while another is a true/false flag. You can immediately run the program that was just created and see the usage:

```
$ ./foo.py -h
usage: foo.py [-h] [-a str] [-i int] [-f] str

Argparse Python script

positional arguments:
```

```
  str                A positional argument

optional arguments:
  -h, --help         show this help message and exit
  -a str, --arg str  A named string argument (default: )
  -i int, --int int  A named integer argument (default: 0)
  -f, --flag         A boolean flag (default: False)
```

All this without writing a line of Python! Quite useful.

In my experience, perhaps 20-50% of the effort to solve most of the exercises can be handled by using `argparse` well. You can specify an exact number of positional arguments, you can specify named arguments that must be constrained to a list of choices, you can force one argument to be an `int` and another to be a `float`, you can get Boolean flags or ensure that arguments are existing files that can be opened and read. I urge you to read the documentation for `argparse` thoroughly. I often find the REPL is quite useful for this:

```
>>> import argparse
>>> help(argparse)
```

Exercises

Now is the time for you to write your own programs. Go into the `exercises` directory and solve `hello_a`. The way you are intended to work is to read the README, then use `new_py.py` to stub out the named program, then use `make test` to pass the tests.

Exercise: hello

If you `ls` the directory, you should see something like this:

```
$ ls
Makefile     README.md    README.pdf   solution.py* test.py*
```

The README says:

Write a Python program named `hello.py` that warmly greets the names you provide. When there are two names, join them with "and." When there are three or more, join them on commas (INCLUDING THE OXFORD WE ARE NOT SAVAGES) and "and." If no names are supplied, print a usage.

```
$ ./hello.py
Usage: hello.py NAME [NAME...]
$ ./hello.py Alice
Hello to the 1 of you: Alice!
$ ./hello.py Mike Carol
```

```
Hello to the 2 of you: Mike and Carol!
$ ./hello.py Greg Peter Bobby Marcia Jane Cindy
Hello to the 6 of you: Greg, Peter, Bobby, Marcia, Jane, and Cindy!
```

The name of the program should be `hello.py`, so do this:

```
$ new_py.py hello
Done, see new script "hello.py."
$ cat -n hello.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : kyclark
     4  Date   : 2019-05-14
     5  Purpose: Rock the Casbah
     6  """
     7
     8  import os
     9  import sys
    10
    11
    12  # --------------------------------------------------
    13  def main():
    14      args = sys.argv[1:]
    15
    16      if len(args) != 1:
    17          print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
    18          sys.exit(1)
    19
    20      arg = args[0]
    21
    22      print('Arg is "{}"'.format(arg))
    23
    24
    25  # --------------------------------------------------
    26  if __name__ == '__main__':
    27      main()
```

Next you will run `make test`, but let's take a moment to understand what this command is doing. The `make` program will look for a file called `Makefile` (or `makefile`) and will then look for a "target" called `test`. Look at the `Makefile`:

```
$ cat -n Makefile
     1  .PHONY: doc test
     2
     3  doc:
     4      pandoc README.md -o README.pdf
     5
     6  test:
```

```
     7      pytest -v test.py
```

You see there is a line with `test:` that starts on the far left side of the file and there is some text indented *with a tab* underneath it. The indented bit will be run when we say `make test`, so you could just run `pytest -v test.py` directly if you wish or if your system doesn't have `make` installed. If you don't have `pytest` installed, then probably this will do it:

```
$ python3 -m pip install pytest
```

Or if you have the Anaconda distribution (highly recommended), then you could do this:

```
$ conda install pytest
```

Now, run `make test` and you should see that you are already passing one test – the usage! That is because `new_py.py` automatically adds that code for you. Of course, it's not accurate code:

```
$ ./hello.py
Usage: hello.py ARG
```

Our program is supposed to take multiple arguments, and we've been given the expected usage in the README, so first correct that.

Now look at the first failure:

```
=================================== FAILURES ===================================
_________________________________ test_01 _________________________________

    def test_01():
        """runs hello"""

        rv, out = getstatusoutput('{} {}'.format(prg, 'Alice'))
        assert rv == 0
>       assert out.rstrip() == 'Hello to the 1 of you: Alice!'
E       assert 'Arg is "Alice"' == 'Hello to the 1 of you: Alice!'
E         - Arg is "Alice"
E         + Hello to the 1 of you: Alice!
```

The lines starting with `E` are the errors we need to fix. It's telling us that the output from the program `out` should equal 'Hello to the 1 of you: Alice!' but instead it says 'Arg is "Alice"'. Confirm this is the case:

```
$ ./hello.py Alice
Arg is "Alice"
```

OK, go fix that.

```
$ ./hello.py Alice
Hello to the 1 of you: Alice!
```

And now run `make test`. You should find you now pass two tests! Here's the next failure:

```
================================== FAILURES ===================================
_________________________________ test_02 _________________________________

    def test_02():
        """runs hello"""

        rv, out = getstatusoutput('{} {}'.format(prg, ' Mike Carol'))
        assert rv == 0
>       assert out.rstrip() == 'Hello to the 2 of you: Mike and Carol!'
E       AssertionError: assert '' == 'Hello to the 2 of you: Mike and Carol!'
E         + Hello to the 2 of you: Mike and Carol!
```

Now when given the arguments "Mike" and "Carol", we need to fix the greeting. When you are passing that test, then move on to the next failure. In this way, you will eventually have a program that meets the specifications. At any point you can inspect the `solution.py` that I have provided if you get stuck or want to compare your version with what I wrote.

For what it's worth, this is the idea of "Test-Driven Development" which you can read about on the Internet.

Exercise: article.py

Make a Python program called `article.py` that prepends before the given "word" (it may not really be a word) the article "a" if the word begins with a consonant or "an" if it begins with a vowel. If given no arguments, the program should provide a usage statement.

```
$ ./article.py
Usage: article.py WORD
$ ./article.py foo
a foo
$ ./article.py oof
an oof
```