Intro to Python

> "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 a change to the "shebang" line. I'm going to use `env` to find `python3` so I won't have a hard-coded path that my user will have to change. 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. (One difference between versions 2 and 3 of Python was that the parens to `print` were not necessary in version 2).

Variables

Let's use the REPL (Read-Evaluate-Print-Loop, pronounced "reh-pull") to play:

```
$ ipython
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: name = 'Duderino'

In [2]: print('Hello,', name)
Hello, Duderino
```

Here I'm showing that it's easy to create a variable called `name` which we assign the value "Duderino." Unlike bash, we don't have to worry about spaces around the `=`. Just as in bash, we can use it in a `print` statement, but we can't directly stick it into the string:

```
In [3]: print('Hello, name')
Hello, name
```

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

```
In [4]: print('Hello, ' + name)
Hello, Duderino
```

Arguments

To say "hello" to an argument passed from the command line, we need the `sys` module. A module is a package of code we can use:

```
$ 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 in the zeroth position (`args[0]`), so the first "argument" to the script is in `args[1]`. It works as you would expect:

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

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:

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

If there are fewer than 2 arguments (remembering that the script name is in the "first" position), 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 Professor
Hello, Professor!
```

On line 7 above, you see we can use the `len` function to ask how long the `args` list is. You can play with the Python REPL to understand `len`. Both strings (like "foobar") and lists (like the arguments to our script) have a "length." Type `help(list)` in the REPL to read the docs on lists.

```
>>> len('foobar')
6
>>> len(['foobar'])
1
>>> len(['foo', 'bar'])
2
```

Here is the same functionality but using two new functions, `printf` (from the base package) and `os.path.basename`:

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

Notice the usage doesn't have a "./" on the script name because we used `basename` to clean it up.

main()

Lastly, let me introduce the `main` function. Many languages (e.g., Python, Perl, Rust, Haskell) have the idea of a "main" module/function where all the processing starts. If you define a "main" function, most people reading your code would understand that the program ought to begin there. I usually put my "main" as the first `def` (the keyword to "define" a function), and then use call it at the end of the script. It's a bit of a hack, but it seems to be standard Python.

```
$ cat -n hello_arg4.py
     1  #!/usr/bin/env python3
     2  """hello with args/main"""
     3
     4  import sys
     5  import os
     6
     7
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) != 2:
    13          script = os.path.basename(args[0])
    14          print('Usage: {} NAME'.format(script))
    15          sys.exit(1)
    16
    17      name = args[1]
    18      print('Hello, {}!'.format(name))
    19
    20
    21  main()
```

Function Order

Note that you cannot put line 21 first 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 say "hi" to any number of arguments:

```
$ cat -n hello_arg5.py
     1  #!/usr/bin/env python3
     2  """hello with to many"""
     3
     4  import sys
     5  import os
     6
     7
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) < 2:
    13          script = os.path.basename(args[0])
    14          print('Usage: {} NAME [NAME2 ...]'.format(script))
    15          sys.exit(1)
    16
```

```
    17      names = args[1:]
    18      print('Hello, {}!'.format(', '.join(name)))
    19
    20
    21  main()
$ ./hello_arg5.py foo
Hello, foo!
$ ./hello_arg5.py foo bar baz
Hello, foo, bar, baz!
```

Look at line 18 to see how we can `join` all the arguments on a comma-space, e.g.,:

```
>>> ', '.join(['foo', 'bar', 'baz'])
'foo, bar, baz'
>>> ':'.join("hello")
'h:e:l:l:o'
```

Notice the second example where we can treat a string like a list of characters.

The other interesting bit on line 16 is how to take a slice of a list. We want all the elements of `args` starting at position 1, so `args[1:]`. You can indicate a start and/or end position. It's best to play with it to understand:

```
>>> x = ['foo', 'bar', 'baz']
>>> x[1]
'bar'
>>> x[1:]
['bar', 'baz']
>>> a = "abcdefghijklmnopqrstuvwxyz"
>>> a[2:4]
'cd'
>>> a[:3]
'abc'
>>> a[3:]
'defghijklmnopqrstuvwxyz'
>>> a[-1]
'z'
>>> a[-3]
'x'
>>> a[-3:]
'xyz'
>>> a[-3:26]
'xyz'
>>> a[-3:27]
'xyz'
```

Conditionals

Above we saw a simple `if` condition, but what if you want to test for more then one condition? Here is a program that shows you how to take input directly from the user:

```
$ cat -n if-else.py
     1    #!/usr/bin/env python3
     2    """conditions"""
     3
     4    name = input('What is your name? ')
     5    age = int(input('Hi, ' + name + '. What is your age? '))
     6
     7    if age < 0:
     8        print("That isn't possible.")
     9    elif age < 18:
    10        print('You are a minor.')
    11    else:
    12        print('You are an adult.')
$ ./if-else.py
What is your name? Geoffrey
Hi, Geoffrey. What is your age? 47
You are an adult.
```

On line 4, we can put the first answer into the `name` variable; however, on line 5, I 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
```

Types

Which leads into the notion that Python, unlike bash, has types – variables can hold string, integers, floating-point numbers, lists, dictionaries, and more:

```
>>> type('foo')
<class 'str'>
>>> type(4)
<class 'int'>
```

```
>>> type(3.14)
<class 'float'>
>>> type(['foo', 'bar'])
<class 'list'>
>>> type(range(1,3))
<class 'range'>
>>> type({'name': 'Geoffrey', 'age': 47})
<class 'dict'>
```

As noted earlier, you can use `help` on any of the class names to find out more of what you can do with them.

So let's return to the `+` operator earlier and check out how it works with different types:

```
>>> 1 + 2
3
>>> 'foo' + 'bar'
'foobar'
>>> '1' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Python will crash if you try to "add" two different types together, but the type of the argument depends on the run-time conditions:

```
>>> x = 4
>>> y = 5
>>> x + y
9
>>> z = '1'
>>> x + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

To avoid such errors, you can coerce your data:

```
>>> int(x) + int(z)
5
```

Or check the types at run-time:

```
>>> for pair in [(1, 2), (3, '4')]:
...     n1, n2 = pair[0], pair[1]
...     if type(n1) == int and type(n2) == int:
...         print('{} + {} = {}'.format(n1, n2, n1 + n2))
...     else:
...         print('Cannot add {} ({}) and {} ({})'.format(n1, type(n1), n2, type(n2)))
```

```
...
1 + 2 = 3
Cannot add 3 (<class 'int'>) and 4 (<class 'str'>)
```

Loops

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  """hello with to many"""
     3
     4  import sys
     5  import os
     6
     7
     8  def main():
     9      """main"""
    10      args = sys.argv
    11
    12      if len(args) < 2:
    13          script = os.path.basename(args[0])
    14          print('Usage: {} NAME [NAME2 ...]'.format(script))
    15          sys.exit(1)
    16
    17      for name in args[1:]:
    18          print('Hello, ' + name + '!')
    19
    20
    21  main()
$ ./hello_arg6.py Salt Peppa
Hello, Salt!
Hello, Peppa!
```

You can use a `for` loop on anything that is like a list:

```
>>> for letter in "abc":
...     print(letter)
...
a
b
c
>>> for number in range(0, 5):
...     print(number)
...
```

```
0
1
2
3
4
>>> for word in ['foo', 'bar']:
...     print(word)
...
foo
bar
>>> for word in 'We hold these truths'.split():
...     print(word)
...
We
hold
these
truths
>>> for line in open('input1.txt'):
...     print(line, end='')
...
this is
some text
from a file.
```

In each case, we're iterating over the members of a list as produced from a string, a range, an actual list, a list produced by a function, and an open file, respectively. (That 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
- Docstring
- imports
- def main()
- main()

```
#!/usr/bin/env python3
"""program docstring"""

import sys
import os
```

```
def main():
    """main"""
    ...

main()
```

Rather than type this out each time, let's use a program to help us start writing new programs. In `/rsgrps/bh_class/bin` (which should be in your `$PATH` by now), you will see `new_py.py`. (If you are working locally on your laptop – which I **strongly** recommend you learn how – you can find the program in `biosys-analytics/bin` which you can either copy into a directory in your `$PATH` or add that directory to your `$PATH`).

Try this:

```
$ new_py.py foo
Done, see new script "foo.py."
$ cat foo.py
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-01-24
Purpose: Rock the Casbah
"""

import os
import sys


# --------------------------------------------------
def main():
    args = sys.argv[1:]

    if len(args) != 1:
        print('Usage: {} ARG'.format(os.path.basename(sys.argv[0])))
        sys.exit(1)

    arg = args[0]

    print('Arg is "{}"'.format(arg))


# --------------------------------------------------
main()
```

I will not require you to use this program to write new scripts, but I do suggest

it could save you time and errors. I wrote this for myself, and I use it every time I start a new program. I first wrote a program like this in the mid-90s using Perl and have always relied on stubbers since.

Notice that the ".py" extension was added for you. You may specify `foo.py` if you prefer.

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. Run with `-h|--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 named options.

```
$ new_py.py -fa foo
Done, see new script "foo.py."
[hpc:login3@~]$ cat foo.py
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-01-24
Purpose: Rock the Casbah
"""

import argparse
import sys
```

```
# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        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', '--flag', help='A boolean flag', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
```

```
def main():
    """Make a jazz noise here"""
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    flag_arg = args.flag
    pos_arg = args.positional

    print('str_arg = "{}"'.format(str_arg))
    print('int_arg = "{}"'.format(int_arg))
    print('flag_arg = "{}"'.format(flag_arg))
    print('positional = "{}"'.format(pos_arg))


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

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, off/on flag:

```
$ ./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.