Lecture 8: Reading/Writing in data I – Command line

Now that we have covered the basics, we can start making programs do something interesting. In general, programs analyze **something**. Perhaps you want to analyze some of the data you have created, some sequencing data you have downloaded from a DNA sequence repository, or analyze a fluorescent micrograph. The first thing that you need to do is get the data in to the program. There are two main ways to accomplish this:

1. Data is typed into the command line
2. Data is stored in a file on your local computer or on a computer accessible through the internet.

Prompt the user for input

input() - You should be familiar with one way to get data from the user using the input() function. Input takes in whatever the user enters and stores it as a string. We have used this already and won't cover this further.

Input when we run the script

Arguments and flags – We have seen that when we run executables/commands in the command line, we can also supply arguments and flags that modify how a program can run. Similarly, we can supply arguments and flags we run a Python program. For example, normally when we run a script (my_script.py), we would run it as:

```
$ python3 my_script.py
```

However, we can also run it with additional arguments like so:

```
$ python3 my_script.py argument1 file2 3 -f
```

These additional arguments are available to the Python script through the sys module as the argv attribute. Argv is a list of strings that includes the name of the python script you are running. To explore this, create a python script called arg_printer.py with the following:

```
import sys
print(sys.argv)
```

Try it out now and see what happens:

```
$ python3 arg_printer.py 1 argument2 useful_file.txt
['arg_printer.py', '1', 'argument2', 'useful_file.txt']

$ python3 arg_printer.py
['arg_printer.py']

$ python3 arg_printer.py 1 2 3 4 5
['arg_printer.py', '1', '2', '3', '4', '5']
```

As you can see, sys.argv is a list of strings of variable size. The first element is always the name of the script.

Let's say you wanted to analyze the protein sequences within an arbitrary file. The easiest way to do this is to enter the name of the file when you run the program:

```
$ python3 protein_reader.py proteins.fa
```

While you could use this approach to add arguments to your program. In general, DO NOT DO THIS. Instead use a custom module designed for taking in arguments to your script.

Input using the argparse module

argparse is a built in module that is extremely useful for organizing and taking in arguments from the user. In general, arguments can be complicated: some are required and some our optional; some are integers and some our strings. Argparse can help you enforce and error check this with minimal effort.

```
>>> import argparse
>>> argparse.ArgumentParser
<class 'argparse.ArgumentParser'>
```

The workhorse of the argparse module is the ArgumentParser class. This class allows you to create an ArgumentParser object that will hold information on all the arguments your program can take in. First you need to create an ArgumentParser object. You can include description information that will be printed to the screen that describes what the program does.

```
>>> parser = argparse.ArgumentParser(description = “This program does something cool”)

>>> type(parser)

<class 'argparse.ArgumentParser'>

>>> dir(parser)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_action_groups', '_actions', '_add_action', '_add_container_actions', '_check_conflict', '_check_value', '_defaults', '_get_args', '_get_formatter', '_get_handler', '_get_kwargs', '_get_nargs_pattern', '_get_option_tuples', '_get_optional_actions', '_get_optional_kwargs', '_get_positional_actions', '_get_positional_kwargs', '_get_value', '_get_values', '_handle_conflict_error', '_handle_conflict_resolve', '_has_negative_number_optionals', '_match_argument', '_match_arguments_partial', '_mutually_exclusive_groups', '_negative_number_matcher', '_option_string_actions', '_optionals', '_parse_known_args', '_parse_optional', '_pop_action_class', '_positionals', '_print_message', '_read_args_from_files', '_registries', '_registry_get', '_remove_action', '_subparsers', 'add_argument', 'add_argument_group', 'add_help', 'add_mutually_exclusive_group', 'add_subparsers', 'argument_default', 'conflict_handler', 'convert_arg_line_to_args', 'description', 'epilog', 'error', 'exit', 'format_help', 'format_usage', 'formatter_class', 'fromfile_prefix_chars', 'get_default', 'parse_args', 'parse_known_args', 'prefix_chars', 'print_help', 'print_usage', 'prog', 'register', 'set_defaults', 'usage']
```

There are a few things about the parser object that are worth pointing out:

1. ArgumentParser is not a method/function. It is a constructor that creates a custom object of the ArgumentParser class. It is similar to running int(6) or string(‘New string’) where you are explicitly creating objects of these types. **In Python, you can recognize constructors because the first letter is capitalized.** (Though this is by convention and not enforced by the interpreter syntax)
2. Besides magic methods and normal methods/attributes we have encountered before, there are also methods/attributes that start with a single underscore. These indicate private methods that are used by the class but SHOULD NOT be used by you in the code. Unlike other languages, Python does not prevent you from using private methods but you should not use these by convention.
3. There are two methods of this class that you will commonly use, add_argument and parse_args. As you get more advanced, you also might use other methods/classes to make more complicated scripts, such as those with subparsers and those with mutually exclusive options.

To see how this works, create a script with the following lines of code and run it.

```
import argparse

parser = argparse.ArgumentParser(description = 'This is a sample program')

args = parser.parse_args()
```

```
$python3 argparse_tut.py
$python3 argparse_tut.py -h
usage: argparse_tut.py [-h]

This is a sample program

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

$python3 argparse_tut.py --help
usage: argparse_tut.py [-h]

This is a sample program

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

Just with these three lines, our program is now able to take in two different flags and print out useful info. We can add positional arguments now using the add_argument() method. Like the name implies, positional arguments are those where the position matter. I.e. the interpreter expects you to enter the arguments in defined order. Add the following after initially creating the parser object:

```
parser.add_argument("loops", type=int, help="Enter the number of times you want a loop run")
```

```
$python3 argparse_tut.py
usage: argparse_tut.py [-h] loops
argparse_tut.py: error: the following arguments are required: loops

$python3 argparse_tut.py -h
usage: argparse_tut.py [-h] loops

This is a sample program

positional arguments:
  loops       Enter the number of times you want a loop run

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


$python3 argparse_tut.py String
usage: argparse_tut.py [-h] loops
argparse_tut.py: error: argument loops: invalid int value: 'String'

$python3 argparse_tut.py 2
```

Check out all the automatic error checking we get! Now we can access the appropriate variables as an attribute of the args object (which was returned by the parser). Add the following to the end of the script and run it a few times:

```
for i in range(0,args.loops):
  print("This is the " + str(i+1) + " time we ran the loop")
```

```
$python3 argparse_tut.py 2
This is the 1 time we ran the loop
This is the 2 time we ran the loop
$python3 argparse_tut.py 5
```

```
This is the 1 time we ran the loop
This is the 2 time we ran the loop
This is the 3 time we ran the loop
This is the 4 time we ran the loop
This is the 5 time we ran the loop
```

We can also add flagged arguments. Flagged arguments don't need to be in any order, but they need to be preceded by a certain flag. The flag can also be the only argument entered as well. It's easiest to see how these work with an example. Add the following to your script:

```
parser.add_argument('-f','--file', help='This is a required Input file', required=True, type=str)
```

```
parser.add_argument('-n','--number', help='This is an optional number to store', type=int)
```

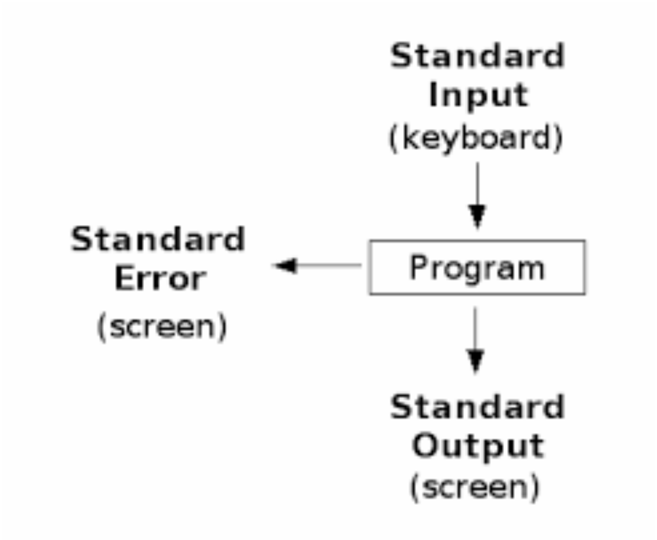
```
$python3 argparse_tut.py
usage: argparse_tut.py [-h] -f FILE [-n NUMBER] loops
argparse_tut.py: error: the following arguments are required: loops, -f/--file
$python3 argparse_tut.py -h
usage: argparse_tut.py [-h] -f FILE [-n NUMBER] loops

This is a sample program

positional arguments:
  loops                 Enter the number of times you want a loop run

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  This is a required Input file
  -n NUMBER, --number NUMBER
                        This is an optional number to store
```

If you just want to have an argument with just a flag:

```
parser.add_argument('-i', '--interaction_flag', help = 'Use this flag if you want to interact with the program', action='store_true')
```

Outputing information to the console

print() – We have been using the print function throughout this class but let's go over what it does to print to the console. Print sends it out put to a stream called standard output. Standard output is one of three standard streams. In computer programming, standard streams are preconnected input and output communication channels between a computer program and its environment when it begins execution. The three I/O connections are called standard input (stdin), standard output (stdout) and standarderror (stderr).



By default standard output is printed to your screen. If you add the following function to a file called print_demonstration.py we can explore how it works.

```
print("This info is printed to the console")
```

```
$ python3 print_demonstration.py
This info is printed to the console
```

However you can easily redirect the output from standard output to any file you would like using the '>' symbol:

```
$ python3 print_demonstration.py > info.txt
$ more info.txt
This info is printed to the console
```

Depending on your settings, if you try to write it to an existing file, you will get the following error:

```
$ python3 print_demonstration.py > info.txt
-bash: info.txt: cannot overwrite existing file
```

In bash, you can force overwrite of the file using the '>|' symbols or append to an existing file using the '>>' symbols:

```
$ python3 print_demonstration.py >| info.txt
$ more info.txt
This info is printed to the console
$ python3 print_demonstration.py >> info.txt
$ python3 print_demonstration.py >> info.txt
$ more info.txt
This info is printed to the console
This info is printed to the console
This info is printed to the console
```

Now let's change the code in the script to produce an error but still redirect standard out to

```
print("This info is printed to the console. Sometimes we make errors
though." + 2)
```

```
$ python3 print_demonstration.py >> info.txt
Traceback (most recent call last):
  File "print_demonstration.py", line 1, in <module>
    print("This info is printed to the console. Sometimes we make errors though."
+ 2)
TypeError: Can't convert 'int' object to str implicitly
```

Note how the error was still printed to the screen and not printed to the info.txt. This is because the error message was sent to a separate stream – stderr so it isn't redirected from the console. If you also want to redirect the standard error stream to a file you use the same redirect commands but place a 2 in front of them:

```
$ python3 print_demonstration.py >> info.txt 2> errors.txt
$ more errors.txt
Traceback (most recent call last):
  File "print_demonstration.py", line 1, in <module>
    print("This info is printed to the console. Sometimes we make errors though."
+ 2)
TypeError: Can't convert 'int' object to str implicitly
```

In a python script you can also print to the standard error using the file keyword in the print function. Add the following to your code:

```
import sys

print("This info is printed to the console.")
print("Sometimes we make errors though.", file = sys.stderr)
```

```
$python3 print_demonstration.py >> info.txt
Sometimes we make errors though.
```

Redirecting your standard out to a file is a reasonable way to output data to a file. The main limitation is that you can only redirect to a single file.