Command line arguments

Up until now, to pass arguments to a function from the command line, we've relied on the `input()` function. We'll look at a couple different ways of passing arguments from the command line starting with **`sys.argv`**. `sys.argv` is a list that contains the argument passed to Python via the command line.

Copy the following code into a file:

```
import sys # we first have to import the sys module

def command_line_arguments(arg1):
    """Print argument passed from the command line"""
    print("Command line argument: " + arg1)

if __name__ == '__main__' :
    argument = sys.argv[1] # sys.argv[1] is the second argument o
n the command line
    command_line_arguments(argument)
```

We can pass multiple arguments from the command line and each will be stored in the `sys.arvg` list.

Copy the following code into a file (`print_args.py`) and execute it from the command line:

```
python print_args.py a b c
```

```
def command_line_arguments(arg1, arg2, arg3):
    """
    Print three arguments passed from the command line
    """
    print("Command line arguments:", arg1, arg2, arg3)

if __name__ == '__main__' :
    import sys
    argument1, argument2, argument3 = sys.argv[1:] # a list slice
    command_line_arguments(argument1, argument2, argument3)
```

What happens if the arguments are integers?

A more versatile and recommended approach is to use **`argparse`** .

Copy the following code into a text file (`concat_seqs.py`) and execute it from the command line:

```
python concat_seqs.py -a CCCC -b GGGG
```

```
def arg_parse():
    """
    Parse command line arguments
    """
    import argparse
    # First we need to create an instance of ArgumentParser which
we will call parser:
    parser = argparse.ArgumentParser()
    # The add_argument() method is called for each argument:
    # We provide two version of each argument:
    # -a is the shortand, --sequence1 is the longhand
    # We can specify a help message describing the argument with
help="message"
    # To require an argument, we use required=True
    parser.add_argument('-a', '--sequence1', required=True, help=
"first sequence")
    parser.add_argument('-b', '--sequence2', required=True, help=
"second sequence")
    # The parse_args() method parses the arguments
    args = parser.parse_args()
    print('args:', args)
    # Here, we'll return the arguments as a tuple
    return args.sequence1, args.sequence2

def cat():
    """
    Concatenate two sequences
    """
    # Assign the values returned from arg_parse to variables
    seq1, seq2 = arg_parse()
    return seq1+seq2

if __name__ == '__main__':
    print(cat())
```

Exercise 12a

Modify `concat_seqs.py` to concatenate three sequences contained with three sequence files. You can use the bash code below to create the sequence file.

In [1]:
```
%%bash
printf 'CCCCCC' >file1.txt
printf 'AAAAAA' >file2.txt
printf 'TTTTTT' >file3.txt
```

Subprocesses

It is often necessary to run other programs within a Python script. The recommended way of doing so is with `subprocess`.

We first import `subprocess`:

In [3]:
```
import subprocess
```

To run command line code within a python script, use the `subprocess.run()` function. Let's use `pwd` to get our current working directory path:

In [4]:
```
subprocess.run('pwd')
```

Out[4]:
```
CompletedProcess(args='pwd', returncode=0)
```

Of course, that's not particularly useful. We have to first store the standard output (stdout) and then we can retrieve it. The following is a general approach for working with `subprocess`:

In [10]:
```
cess = subprocess.run('pwd', stdout=subprocess.PIPE, stderr=subprocess.PIP
ut(process.stdout)
cess.stderr
```

```
b'/Users/montgomery/Documents/DSCI511/notebooks\n'
```

Out[10]:
```
b''
```

If you want to run `subprocess` with a combination of commands and arguments, they have to be passed as a list with each element as something that would normally be separated by a space. Let's make a directory called `my_directory` using the unix command `mkdir`:

In [13]:
```
subprocess.run(['mkdir', 'my_directory'])
```

Out[13]:
```
CompletedProcess(args=['mkdir', 'my_directory'], returncode=1)
```

We can use the unix command `ls` to see if the directory exists:

In [15]:
```
process = subprocess.run('ls', stdout=subprocess.PIPE, stderr=subprocess.
print(process.stdout.decode('utf-8'))
```

```
command_line_arguments.ipynb
data.csv
dictionaries.ipynb
file1.txt
file2.txt
file3.txt
foo.pdf
fpkm.pdf
lists.ipynb
ls_stdout.txt
miRNAs.fa
my_directory
regular_expressions.ipynb
test.pdf
tuples.ipynb
```

The `decode()` method is used to convert the output to unicode so that `\n` shows up as a new line.

More simply we can get the output of ls using `check_output`:

In [17]:
```
print(subprocess.check_output('ls').decode('utf-8'))
```

```
command_line_arguments.ipynb
data.csv
dictionaries.ipynb
file1.txt
file2.txt
file3.txt
foo.pdf
fpkm.pdf
lists.ipynb
ls_stdout.txt
miRNAs.fa
my_directory
regular_expressions.ipynb
test.pdf
tuples.ipynb
```

In [ ]:
```
subprocess.run('ls', stdout=open('ls_stdout.txt', 'w'), stderr=subprocess
```

stdout, stdin, and stderr are the standard streams between a computer and its environment (see Wiki (https://en.wikipedia.org/wiki/Standard_streams) for more details).

Exercise 12b

Use `subprocess` to retrieve your working directory using the unix command `pwd`.

In [18]:
```
subprocess.check_output('pwd')
```

Out[18]:
```
b'/Users/montgomery/Documents/DSCI511/notebooks\n'
```

Let's write a script (bowtie_wrapper.py) for filtering and mapping high-throughput sequencing data.

We'll use `subprocess` to call trimmomatic to remove adapters and quality filter data in a fastq file and then bowtie2 to map the data to a reference genome.

For trimmomatic, we'll use the following settings:

```
trimmomatic SE 'input_file' 'output_file' ILLUMINACLIP:/home/stud
ent/TruSeq-smallRNA.fa:2:30:10 MINLEN:16 AVGQUAL:30

Where input_file = /home/student/small_RNAs.fastq
```

And for bowtie alignment, we'll use:

```
bowtie2 -x 'path_to_bowtie_index/prefix' -U 'fastq_file_name' -S
'output.sam'

Where path_to_bowtie_index/prefix = /home/student/cel_bowtie/cel
```

Next, we'll log on to a server using `ssh` to test our program.

From the command line, make a directory on the server called `yourname_dir`:

`cd` into the directory and get the directory path using `pwd`.

Open a new terminal window and change into the directory containing your script.

Copy your Python script into your directory on the server using `scp`:

```
scp path_to_script_on_your_computer username@servername:/path_to_
yourname_dir
```

Run you python script on the server.

Useful Python packages

There are many useful Python libraries for working with genomics data. We'll briefly touch on a couple of these if time allows.

NumPy

NumPy is the most popular scientific computing package for Python. It comes standard with Anaconda Python distributions (alternatively, it can be installed using `pip install numpy`). Its highlights include:

```
An N-dimensional array object
Additional array functionality
Tools for integrating C/C++ and Fortran code into your python pro
grams
Linear algebra capabilities
```

In [ ]:
```
import numpy as np
```

Let's revisit the homeworks assignment where you were asked to transpose an array, using numpy to do the task instead:

In [ ]:
```
a = np.array([(1,2,3), (4,5,6)])
a
```

In [ ]:
```
b = np.transpose(a)
b
```

For a Numpy tutorial, see SciPy (https://docs.scipy.org/doc/numpy/user/quickstart.html).

matplotlib

Is the most popular library for plotting 2D data. It comes standard with Anaconda Python distributions (alternatively, it can be installed using `pip install matplotlib`). matplotlib.pyplot is a collection of functions that provide MATLAB like functionility to Python.

In [ ]:
```
import matplotlib.pyplot as plt
```

Let's plot the integers 1-4 against their values multiplied by 2:

In [ ]:
```
plt.plot([1, 2, 3, 4], [2, 4, 6, 8]) # ro = red circles
plt.ylabel('2xNumber')
plt.xlabel('Number')
plt.show()
```

The default plot format is a solid blue line (code: 'b-'). Let's change our plot so that it has red circles for data points:

In [ ]:
```
plt.plot([1, 2, 3, 4], [2, 4, 6, 8], 'ro') # ro = red circles
plt.ylabel('2xNumber')
plt.xlabel('Number')
plt.show()
```

Generally when working with matplotlib, you will use numpy arrays:

In [ ]:
```
a = np.array([1,2,3,4])
plt.plot(a, a**2)
plt.ylabel('Square')
plt.xlabel('Number')
plt.show()
```

Let's do a bar plot:

In [ ]:
```
labels = ['wt', 'mut1', 'mut2']
values = [100, 25, 50]
index = np.arange(len(labels))
plt.bar(index, values)
plt.xticks(index, labels)
plt.show()
```

This doesn't even begin to scratch the surface of numpy functionality. For more resources, see matplotlib (https://matplotlib.org/tutorials/index.html).

pandas

pandas is a package for data manipulation that uses R DataFrame objects thereby providing R functionality to Python. pandas, numpy, and matplotlib are often used in combination. For more details see pandas (https://pandas.pydata.org/pandas-docs/stable/tutorials.html).

In [ ]:
```
import pandas as pd
```

Create a file (`data.csv`) with genes and correspoding fpkm values:

In [ ]:
```
%%bash
printf 'gene_id,fpkm\n' >data.csv
printf 'gene1,173.2\n' >>data.csv
printf 'gene2,0\n' >>data.csv
printf 'gene3,11.6\n' >>data.csv
printf 'gene4,19.2\n' >>data.csv
printf 'gene5,190.2\n' >>data.csv
printf 'gene6,50.6\n' >>data.csv
printf 'gene7,74.1\n' >>data.csv
printf 'gene8,102.3\n' >>data.csv
printf 'gene9,92.3\n' >>data.csv
printf 'gene10,120.7\n' >>data.csv
cat data.csv
```

Use pandas to create a dataframe from the csv file:

In [ ]:
```
genes = pd.read_csv('data.csv')
```

Plot the data using `Dataframe.plot` , a pandas interface to matplotlib:

In [ ]:
```
genes.plot(x='gene_id', y ='fpkm', kind='bar')
plt.savefig('fpkm.pdf')
```

For more details on DataFrame.plots, see [pandas (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html)](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html).