PROGRAMMING BOOTCAMP

Useful Python modules

Material Overview

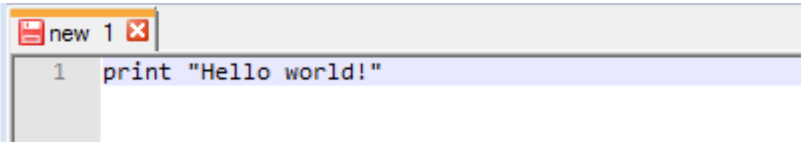
1. Review: writing scripts (instead of notebooks)
2. Useful modules
   a) `sys` – command line args, exiting scripts early
   b) `os` – doing things with file systems
   c) `glob` – getting lists of files
   d) `subprocess` – system commands from within python
   e) `time` - get the system time, create a timer
3. Odds 'n ends

1. Review of scripts

Using scripts

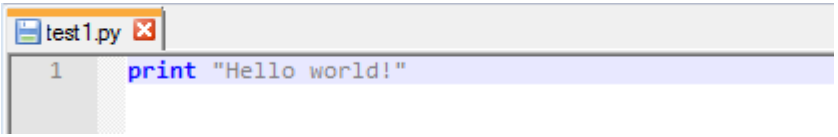
Step 1: Creating a script

- Open a plain text editor (Notepad++, TextWrangler)
- Type the following:



```
print "Hello world!"
```

- Save your file in your lab1 folder as `test1.py`
- *Note:* Depending on your text editor, you may notice some of the code has changed colors. This is called syntax highlighting:



```
print "Hello world!"
```

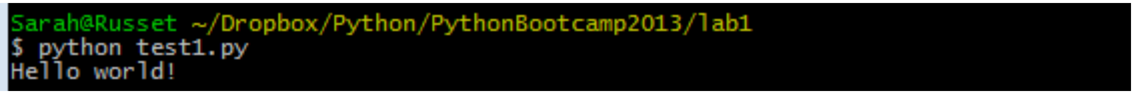
Using scripts

Step 2: Running the script

- Open your terminal and navigate to the folder where you saved your script (use `cd`, `ls`/`dir`, and `pwd`).
- Once in the correct folder, type:

```
python test1.py
```

- Python will now attempt to execute your script. If there are no errors in your code, you should see something like this:



```
Sarah@Russet ~/Dropbox/Python/PythonBootcamp2013/lab1
$ python test1.py
Hello world!
```

2a. sys

sys

Purpose: Wide variety of things... but for our purposes, it mainly provides a way of:

1. getting command line arguments
2. exiting the script early

Command line arguments

Usually when we run a python script, we type this into the terminal:

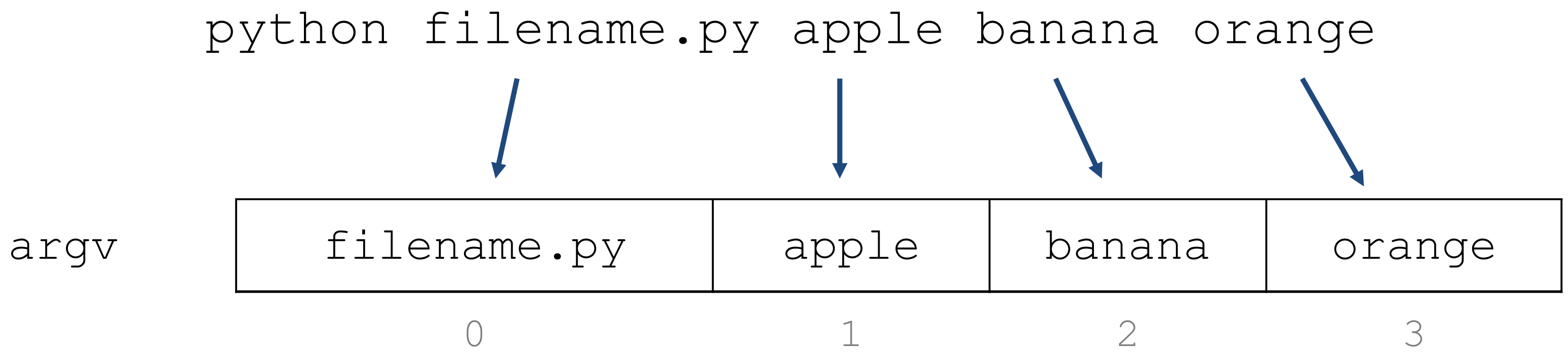
```
python filename.py
```

We can also provide additional information when we run our script ("arguments"):

```
python filename.py arg1 arg2 agr3
```

Command line arguments

You can add as many command line args as you want. All args will be automatically stored (in order) in a list called `argv`. The first item in this list will be the name of your script, followed by any arguments you included.



```
python filename.py apple banana orange
```

argv	filename.py	apple	banana	orange
	0	1	2	3

Using `argv`

Before we can use the `argv` list, we must import `sys`:

```
import sys
```

Then we can access argv by typing:

```
sys.argv[someInt]
```

For example, to get the 1st argument:

```
firstArg = sys.argv[1]
```

Example: argTest.py

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print ("Script name:", scriptName)
print ("Arg1:", arg1)
print ("Arg2:", arg2)
```

argTest.py

Result:

```
> python argTest.py apple banana
Script name: argTest.py
Arg1: apple
Arg2: banana
```

← on the command line

Example: argTest.py

argTest.py

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print ("Script name:", scriptName)
print ("Arg1:", arg1)
print ("Arg2:", arg2)
```

What if we did this? (only one arg provided)

```
> python argTest.py apple
```

Example: argTest.py

```
import sys

scriptName = sys.argv[0]
arg1 = sys.argv[1]
arg2 = sys.argv[2]
print ("Script name:", scriptName)
print ("Arg1:", arg1)
print ("Arg2:", arg2)
```

argTest.py

What if we did this? (only one arg provided)

```
> python argTest.py apple
Traceback (most recent call last):
  File "argTest.py", line 5, in <module>
    arg2 = sys.argv[2]
IndexError: list index out of range
```

Example 2: addMe.py

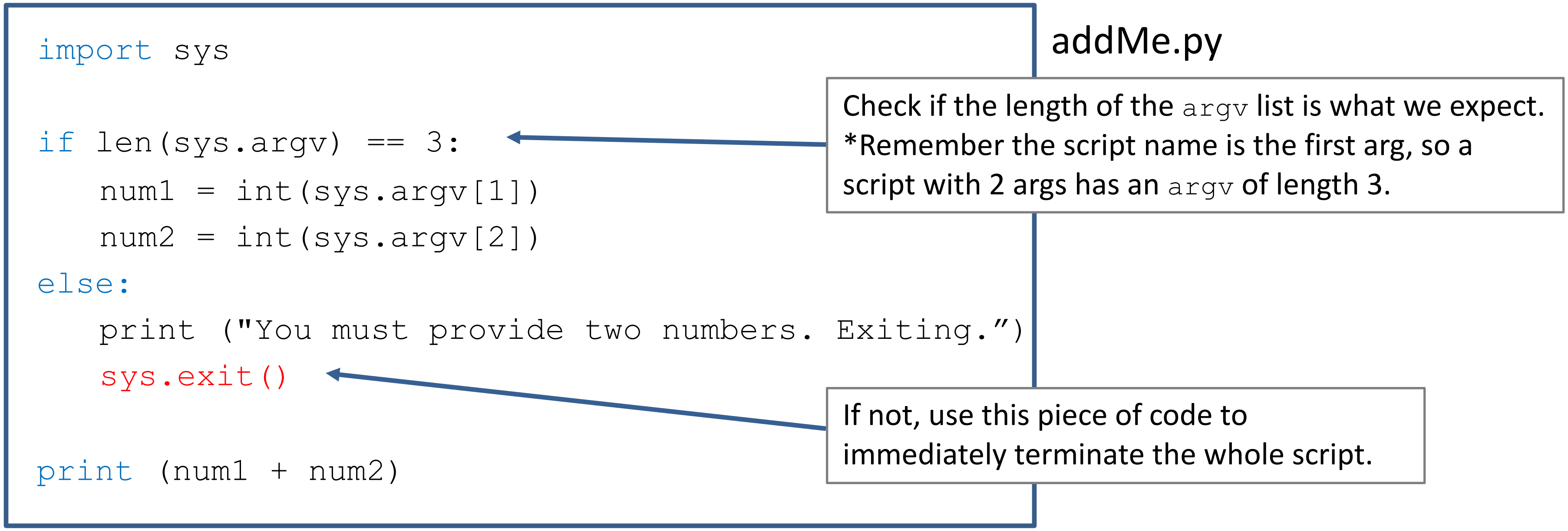
To gracefully exit when the wrong arguments are provided, you can use sys.exit():

```
import sys

if len(sys.argv) == 3:
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
else:
    print ("You must provide two numbers. Exiting.”)
    sys.exit()

print (num1 + num2)
```

addMe.py

Example 2: addMe.py

To gracefully exit when the wrong arguments are provided, you can use sys.exit():



addMe.py

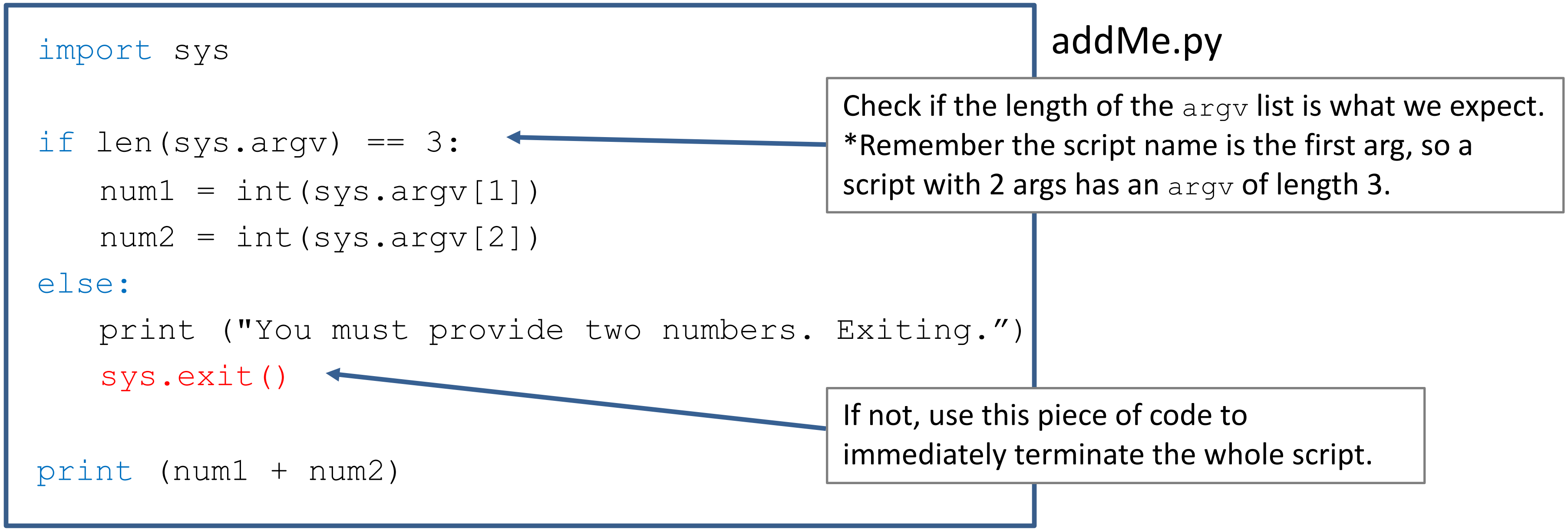
```
import sys

if len(sys.argv) == 3:
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
else:
    print ("You must provide two numbers. Exiting.”)
    sys.exit()

print (num1 + num2)
```

Check if the length of the `argv` list is what we expect.
*Remember the script name is the first arg, so a script with 2 args has an `argv` of length 3.

If not, use this piece of code to immediately terminate the whole script.

Example 2: addMe.py

To gracefully exit when the wrong arguments are provided, you can use sys.exit():



addMe.py

```
import sys

if len(sys.argv) == 3:
    num1 = int(sys.argv[1])
    num2 = int(sys.argv[2])
else:
    print ("You must provide two numbers. Exiting.”)
    sys.exit()

print (num1 + num2)
```

Check if the length of the argv list is what we expect.
*Remember the script name is the first arg, so a script with 2 args has an argv of length 3.

If not, use this piece of code to immediately terminate the whole script.

Result

```
> python addMe.py 100 50
150
```

Or:

```
> python addMe.py 302
You must provide two numbers. Exiting.
```

Other notes on command line args

- Separate args with a space
- You don't need to put quotes around strings on the command line, *UNLESS* your string contains white space
- Everything is read in as a string, so numbers must be converted with int() or float() inside the script
- Don't use commas when specifying numbers (e.g. say 10000 instead of 10,000)

Why use command line args?

- If you plan to run your script on multiple datasets, you can simply supply different filenames to the command instead of editing a hard-coded file name
- Facilitates the creation of “pipelines”, for the above reason
- If you are keeping track of what commands you run on your data (which you should!), having all the relevant info as part of the command itself (the file name, certain parameters, etc.) makes what you did more transparent and reproducible.
- The rule of thumb is: if you NEVER plan to change a variable, no matter what dataset you run your code on, it’s ok to hard code it. Otherwise, consider making it a command line arg.

2b. os

os

Purpose: Useful functions for working with file names/directory paths.

Example:

```
>>> os.path.exists("test_file.txt")
True
>>> os.mkdir("newFolder")
```

More info:

https://docs.python.org/3/library/os.html

os.path

```
>>> import os
>>> os.path.exists("test_file.txt") #checks if file/directory exists
True

>>> os.path.isfile("test_file.txt") #checks if it is a file
True

>>> os.path.isdir("test_file.txt") #checks if it is a directory
False

>>> os.path.getsize("test_file.txt") #gets size of file
18L

>>> os.path.abspath("test_file.txt") #gets absolute/full path of file

>>> fullPath = os.path.abspath("test_file.txt")
>>> os.path.basename(fullPath) #extracts file name from longer path
'test_file.txt'
>>> os.path.dirname(fullPath) #extracts path, removes file name

>>> os.mkdir("newFolder") #makes a new directory
```

A note on file paths

- So far we've mostly worked with input/output files stored in the same directory as our script
- What if we want to work with files stored somewhere else?

```
# open a file in a directory contained
# inside the current directory:
inFile = open("data/input_file.txt", 'r')

# open a file in the directory that contains
# the current directory (parent directory)
inFile = open("../input_file2.txt", 'r')

# open a file using an absolute path (i.e.
# a path that will always work, regardless of the
# current directory location)
inFile = open("/home/raju/lab7/data/input_file.txt", 'r')
inFile = open("/home/raju/lab7/input_file2.txt", 'r')
```

2c. glob

glob

Purpose: Get list of files in a folder that match a certain pattern. Good for when you need to read in a large number of files but don't have a list of all their file names.

Example:

```
glob.glob("../data/sequences/*.fasta")
```

Important to note:

The `*` here is a wildcard. So this will match any file in `../data/sequences/` that ends in `.fasta`.

More info:

http://docs.python.org/3/library/glob.html

glob

```
>>> import glob

>>> glob.glob("sequences/*") #get list of everything in "sequences" folder
['sequences/abcde.fasta', 'sequences/asdas123.fasta',
'sequences/README.txt', 'sequences/seq1.fasta', 'sequences/seq2.fasta',
'sequences/seq3.fasta', 'sequences/temp_file.tmp']

>>> glob.glob("sequences/*.fasta") #get list of all with .fasta extension
['sequences/abcde.fasta', 'sequences/asdas123.fasta',
'sequences/seq1.fasta', 'sequences/seq2.fasta', 'sequences/seq3.fasta']

>>> glob.glob("sequences/seq*.fasta") #get everything named seq*.fasta
['sequences/seq1.fasta', 'sequences/seq2.fasta', 'sequences/seq3.fasta']

>>> glob.glob("*") #get list of everything in current folder
['data', 'lab7_useful_modules.pptx', 'newFolder', 'opt_test.py',
'sequences', 'test_file.txt']
```

The * is a wildcard -- it will match anything.

2d. subprocess

subprocess

Purpose: Launch another program or a shell command from within a Python script.

Example:

```
  subprocess.Popen("python other_script.py")
```

More info:

http://docs.python.org/3/library/subprocess.html

subprocess

Basic command:

```
   job = subprocess.Popen(command)
```

Recommended version:

```
   job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

subprocess

Basic command:

```
   job = subprocess.Popen(command)
```

Recommended version:

```
   job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
```

Allows us to run shell (terminal) commands

If the command would normally output something to the terminal, this allows us to capture that output in a string variable. That way we can read through it in our code and use it, if necessary.

Allows us to capture the "standard error" stream of the command. In other words, this will allow us to check if our command succeeded or gave an error.

subprocess - an example

```
# create and run command, use variable 'job' to access results
command = "blastn -query seq1.fasta -db refseq_rna"
job = subprocess.Popen(command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.STDOUT)

# read whatever this command would have printed to the screen,
# and then actually print it (it's suppressed otherwise)
jobOutput = job.stdout.readlines()
for line in jobOutput:
    print (line,)

# check for error and ensure that the script does not continue
# until the command has finished executing.
result = job.wait()
if result != 0:
    print ("There was an error running the command.”)
```

subprocess - in a custom function

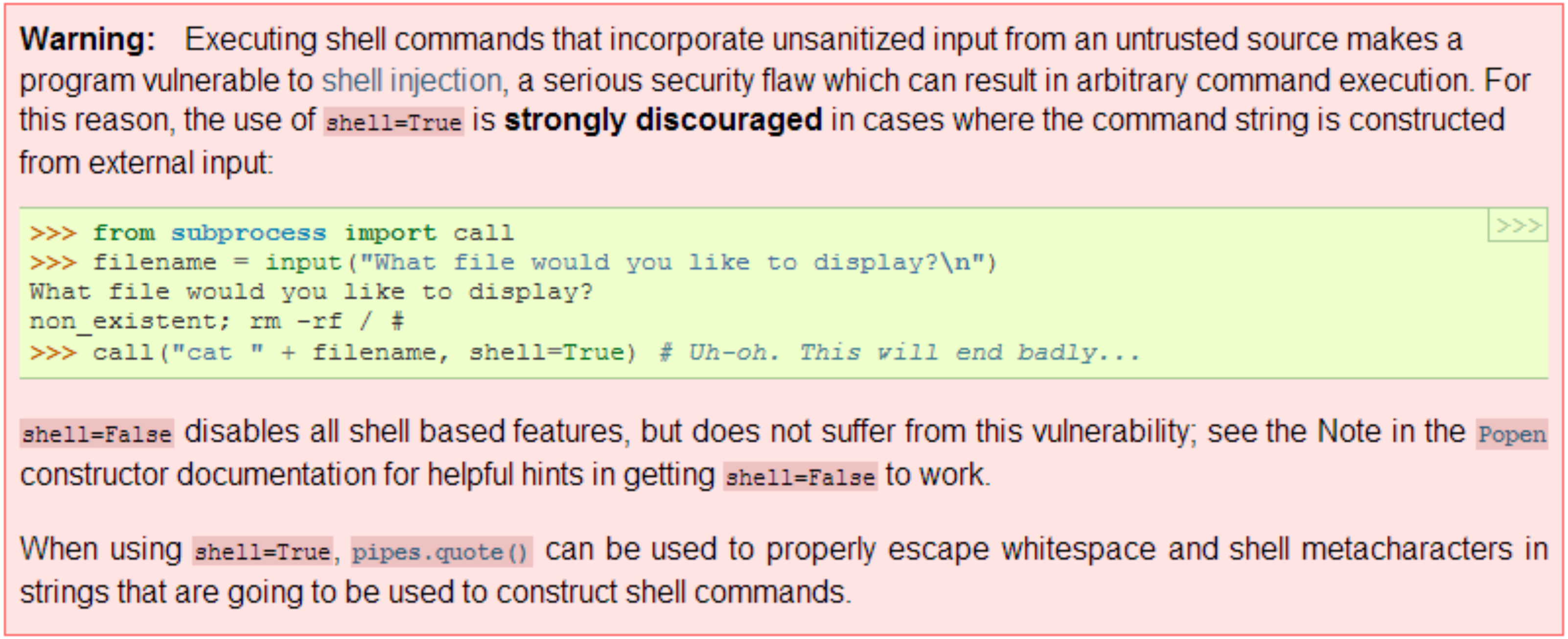
(in useful_fns.py)

```
# A function that runs the given command using the system shell.
# Returns the output of the command in a list, the result variable, and whether there was an error.
def run_command(command, verbose=False):
    import subprocess
    error = False

    if verbose == True:
        print (command)
        print ("”)

    job = subprocess.Popen(command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    jobOutput = []
    if verbose == True:
        for line in job.stdout:
            print ("    ", line,)
            jobOutput.append(line)
    else:
        jobOutput = job.stdout.readlines()
    result = job.wait()
    if result != 0:
        error = True

    return (jobOutput, result, error)
```

subprocess - in a custom function

```
# Using the custom function in another script:
import useful_fns as uf

command = "blastn -query seq1.fasta -db refseq_rna"
(output, result, error) = uf.run_command(command, verbose=True)

# check for error
if error:
    print ("Error running command:", command)
    print ("Exiting.”)
    sys.exit()

# use output, or whatever
for line in output:
    ...
```

subprocess - a warning



Warning: Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is **strongly discouraged** in cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

`shell=False` disables all shell based features, but does not suffer from this vulnerability; see the Note in the `Popen` constructor documentation for helpful hints in getting `shell=False` to work.

When using `shell=True`, `pipes.quote()` can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

If you set shell = True, this executes the command using the shell. This is good because it lets us do more things, but it's potentially dangerous because it essentially opens up a way for someone to run malicious shell commands (like in the example above, a command to delete all of your files...). Should you worry about this? Probably not, UNLESS you plan to run code **on your computer/server that accepts input from strangers over the internet**. If you're just running the code yourself, or letting other people run the code on their own computers themselves, this is a non-issue.

2e. time

time

Purpose: Get the current system time. Can be used to time your code.

Example:

```
import time

startTime = time.time()
...some code...
endTime = time.time()
elapsedTime = endTime - startTime
```

More info:

http://docs.python.org/3/library/time.html

Important to note:

`time.time()` returns a `float` that indicates the time, in seconds, since the start of the "epoch" (this is operating system-dependent) at the current moment. It won't make much sense by itself, but we can use it to make simple timers as shown here.

3. Odds 'n ends

+=

This is a shortcut for adding/concatenating onto a variable. Works for strings and numbers.

Examples:

```
count = 0
while count < 100:
    count += 1 #same as count = count + 1

name = ""
for c in ”Beyonce"
    name += c #same as name = name + c
```

Nested dictionaries

A dictionary can store almost anything... including other dictionaries! This is useful for when you want to associate several pieces of info with a given key.

Example:

```
inFile = open("genes.bed", 'r')

for line in inFile:
    (chr, start, end, geneID, score, strand) = line.split()
    geneDict[geneID] = {} #dictionary within a dictionary!
    geneDict[geneID]['chrom'] = chr
    geneDict[geneID]['startPos'] = int(start)
    geneDict[geneID]['endPos'] = int(end)
    geneDict[geneID]['strand'] = strand

inFile.close()

print (geneDict['Tcea1']['strand'] #example of accessing data)
```

genes.bed

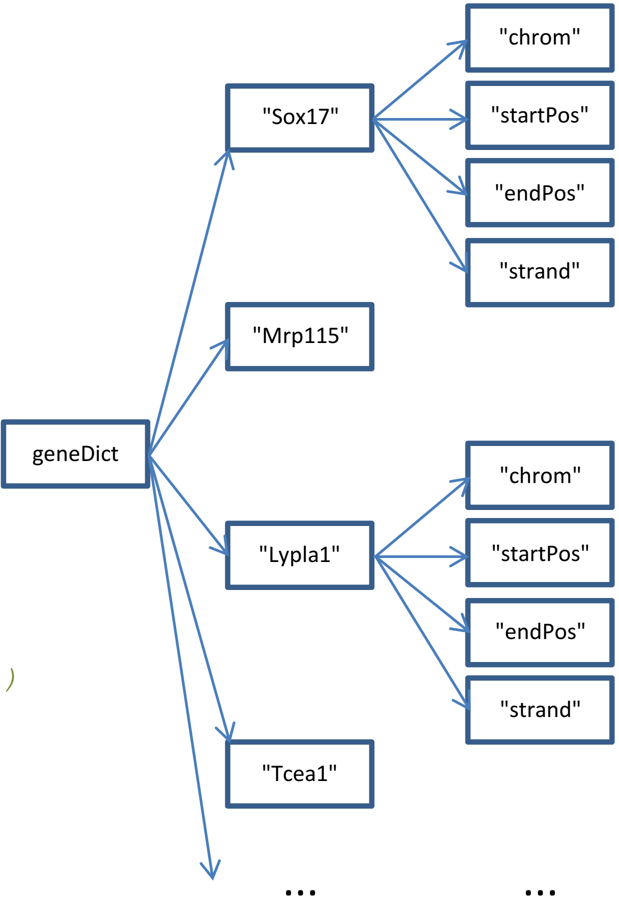
```
chr1	4492668	4493099	Sox17	0	-
chr1	4493466	4493771	Sox17	0	-
chr1	4493863	4495135	Sox17	0	-
chr1	4495942	4496290	Sox17	0	-
chr1	4776801	4777524	Mrpl15	0	-
chr1	4777648	4782567	Mrpl15	0	-
chr1	4782733	4783950	Mrpl15	0	-
chr1	4784105	4785572	Mrpl15	0	-
chr1	4807982	4808454	Lypla1	0	+
chr1	4808486	4828583	Lypla1	0	+
chr1	4828649	4830267	Lypla1	0	+
chr1	4830315	4832310	Lypla1	0	+
chr1	4832381	4837000	Lypla1	0	+
chr1	4837074	4839386	Lypla1	0	+
chr1	4839488	4840955	Lypla1	0	+
chr1	4841132	4844962	Lypla1	0	+
chr1	4857976	4858327	Tcea1	0	+
chr1	4858503	4867469	Tcea1	0	+
chr1	4867532	4878026	Tcea1	0	+
chr1	4878132	4886743	Tcea1	0	+
chr1	4886831	4889456	Tcea1	0	+
chr1	4889602	4890739	Tcea1	0	+
chr1	4890796	4891914	Tcea1	0	+
chr1	4892069	4893416	Tcea1	0	+
chr1	4893563	4894933	Tcea1	0	+
chr1	4895005	4896355	Tcea1	0	+
chr1	4910662	4912313	Rgs20	0	-
chr1	4912548	4916896	Rgs20	0	-
chr1	4916980	4923846	Rgs20	0	-
chr1	4923989	5019310	Rgs20	0	-
```

Nested dictionaries

A dictionary can store almost anything... including other dictionaries! This is useful for when you want to associate several pieces of info with a given key.

Example:

```
inFile = open("genes.bed", 'r')

for line in inFile:
    (chr, start, end, geneID, score, strand) = line.split()
    geneDict[geneID] = {} #dictionary within a dictionary!
    geneDict[geneID]['chrom'] = chr
    geneDict[geneID]['startPos'] = int(start)
    geneDict[geneID]['endPos'] = int(end)
    geneDict[geneID]['strand'] = strand

inFile.close()

print (geneDict['Tcea1']['strand'] #example of accessing data)
```



Error handling with try-except

Purpose: catch a specific error before it causes the script to terminate, and handle the error in a manner of your choosing.

Syntax:

```
try:
    ...some code here...
    ...that might create an error...
except ErrorName:
    ...code to execute if error occurs...
else:
    ...(optional) code to execute if no error...
```

You must provide the specific name of the error type (e.g. `TypeError`, `ValueError`, `IOError`, etc)

Example:

```
try:
    inFile = open(fileName, 'r')
except IOError:
    print ("Error: could not open", fileName, "--exiting.”)
    sys.exit()
for line in inFile:
...
```

You can do anything you want in the except block; you do not have to exit. However, in general it's good form to at least print some kind of message/warning.

A whole world of built-in functions

- There's tons of stuff I didn't get a chance to tell you about
- In particular, there are several functions out that that automatically do things I made you do manually (sorry! It's for the sake of learning!)
  - String functions: https://docs.python.org/3/library/stdtypes.html#string-methods
    - `string.count()`
    - `string.upper() / string.lower()`
    - `string.find()`
    - `string.join()`
  - `random.choice()`
  - many more

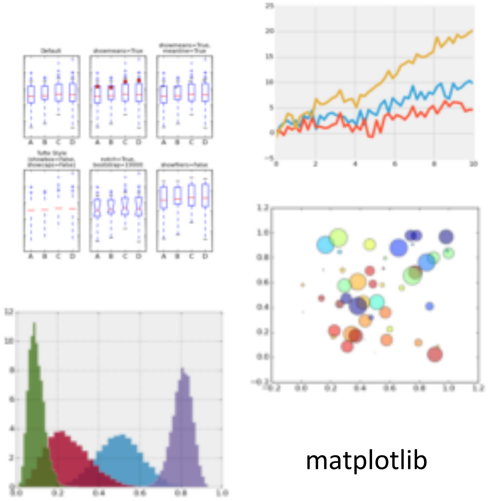
Other useful modules

Built-in:

- `multiprocessing` – functions for writing parallel code that utilizes multiple CPU cores
- `optparse/argparse` – fancier command line args
- `re` – regular expressions (advanced pattern matching)
- `collections` – advanced data structures
- `logging` – facilitates the creation of log files
- `datetime` – for accessing/manipulating date & time info

Not built-in (but comes with Anaconda)

- `SciPy` – scientific/mathematical algorithms
- `NumPy` – advanced math & linear algebra
- `matplotlib` – plotting module for python
- **`pandas` – data structures and data analysis**



matplotlib