Python Modules

Slides by Sarah Middleton and Sammy Klasfeld

(Please sign-in on the counter near the back door and take course survey on Piazza)

Penn
University of Pennsylvania

Sponsored by:

Institute for Biomedical Informatics

BGSA

GCB

PICTURE TIME!!!!

(PLEASE STAND UP AT THE FRONT OF THE CLASS ROOM WITH ME)


PROGRAMMING
BOOTCAMP

1. Before we start, a digression

Why is programming so hard?

1. It requires creativity and problem solving skills
2. You must accept that your knowledge will always be incomplete
3. You have to effectively communicate and working with others (not always in person)
4. **You need to be able to sit at your computer for an inordinate amount of time**

Ex-Google Tech Lead:
Patrick Shyu
YouTube: @TechLead

Today's TAs

Apexa

Sammy

Ben

David

Will

Jake

Parisa

Previously on Python Bootcamp

- Dictionaries and writing files

Lab5 Comments

- Explain why you believe code gives you errors
- The best way to test your code is by creating “pseudo-data”
- When writing to file, feel free to check your code by opening the new file
- Please use a pencil and paper to solve problems

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

fruits

key	value
apple	red
banana	yellow
grape	purple

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

fruits

key	value
apple	red
banana	yellow
grape	purple

key =”apple”

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

red

fruits

key	value
apple	red
banana	yellow
grape	purple

key =”apple”

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

red

fruits

key	value
apple	red
banana	yellow
grape	purple

key =~~"apple"~~ "banana"

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

red
yellow

fruits

key	value
apple	red
banana	yellow
grape	purple

key = ~~"apple"~~ "banana"

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

red
yellow

fruits

key	value
apple	red
banana	yellow
grape	purple

key =~~"apple" "banana"~~ "grape"

Problem 1

```
1 # fruit dictionary
2 fruits = {"apple":"red",
3 "banana":"yellow",
4 "grape":"purple"}
5
6 for key in fruits:
7     print (fruits[key])
```

red
yellow
purple

fruits

key	value
apple	red
banana	yellow
grape	purple

key =~~"apple" "banana"~~ "grape"

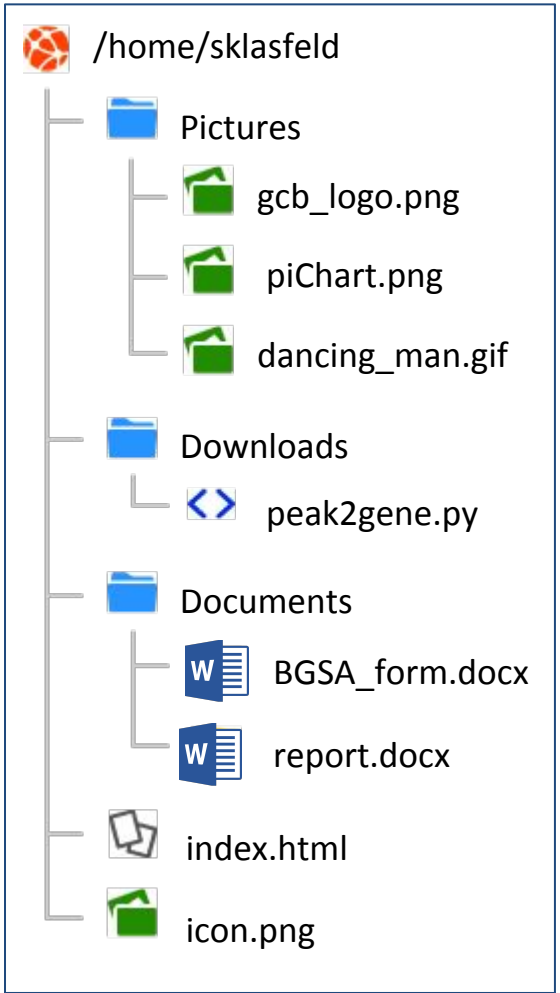
What we need for today’s class

- Your favorite text editor
- A linux command line

Today’s schedule

1. Review: file paths
2. Review: writing scripts (instead of notebooks)
3. Writing your own modules
4. Useful modules for writing scripts
    a. argparse - command line arguments
    b. sys - exiting scripts early
    c. os - doing things with file systems

1. File Paths



File Paths

- Files paths are **a string of characters that represent an address of where files are located on your system**
- Used to represent the directory/file relationship, the delimiting character is most commonly the slash (“/”)
- For example, the file path to “gcb_logo.png” (which is located based on the figure to the left) is: `/home/sklasfeld/Pictures/gcb_logo.png`

File paths in Python

- 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/sammy/lab6/data/input_file.txt", 'r')
inFile = open("/home/sammy/lab6/input_file2.txt", 'r')
```

2. Scripts

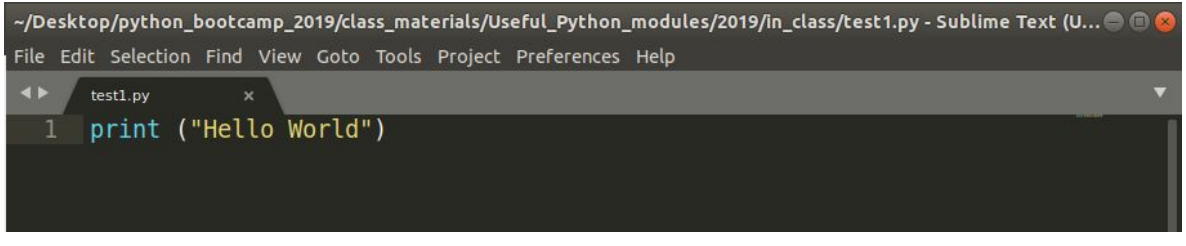
Basic Terminal Commands

ls	see what is in current working directory
pwd	see current working directory path
cd	change directory
man	see manual of terminal command
head <file>	see top of file
tail <file>	see bottom of file
wc -l <file>	see how many lines are in a file

Using scripts

Step 1: Creating a script

- Open a plain text editor (Notepad++, TextWrangler, Sublime)
- Type the following:
  `print ("Hello World")`
- Save your file in your lab6 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:



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:

Command line:

```
sklasfeld@sklasfeld-XPS-12-9310 :~/Desktop/python_bootcamp/lab7 $ python
test1.py
Hello World
```

3. Creating your own modules

What is a module again?

- A module is a Python object with named attributes that you can bind and reference

What is a module again?

- A module is a Python object with named attributes that you can bind and reference
- Usage example:

```
import random


random.random()
random.randint(0,10)
random.gauss(5, 2)
```

What is a module again?

- A module is a Python object with named attributes that you can bind and reference
- Usage example:

```
import random


random.random()
random.randint(0,10)
random.gauss(5, 2)
```

You always need to import the module!

Namespaces

- Previously we have just import modules with “import {module_name}” and then we ran functions from that module using the module’s namespace “{module_name}.{mod_function}”

Namespaces

- Previously we have just import modules with “import {module_name}” and then we ran functions from that module using the module’s namespace “{module_name}.{mod_function}”
- You can give the **module’s namespace a nickname** using the “as” operator.

```
import numpy as np
np.ones(5)
```

Namespaces

- Previously we have just import modules with “import {module_name}” and then we ran functions from that module using the module’s namespace “{module_name}.{mod_function}”
- You can give the **module’s namespace a nickname** using the “as” operator.

```
import numpy as np
np.ones(5)
```

- You can import **specific functions into your namespace**

```
from random import randint
randint(0,10)
```

Namespaces

- Previously we have just import modules with “import {module_name}” and then we ran functions from that module using the module’s namespace “{module_name}.{mod_function}”
- You can give the **module’s namespace a nickname** using the “as” operator.

```
import numpy as np
np.ones(5)
```

- You can import **specific functions into your namespace**

```
from random import randint
randint(0,10)
```

- You can also import **all the functions into your namespace**

```
from math import *
sqrt(6)
```

Why create modules?

If you have a set of functions you want to use in various different scripts (e.g. a function to read in a fasta file), you can **save these functions in a separate file and then *import* them into other scripts**.

Keep your functions in a separate file

useful_fns.py (custom module script):

```
# Count (potentially overlapping) instances of a
subsequence in a string
def count_occurrences(seq, subseq):
      seq = seq.upper()
      subseq = subseq.upper()
      count = 0
      index = 0
      done = False
      while not done:
            index = seq.find(subseq, index)
            if (index == -1):
                  done = True
            else:
                  count += 1
                  index += 1
      return count
```

Keep your functions in a separate file

useful_fns.py (custom module script):

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
      seq = seq.upper()
      subseq = subseq.upper()
      count = 0
      index = 0
      done = False
      while not done:
            index = seq.find(subseq, index)
            if (index == -1):
                  done = True
            else:
                  count += 1
                  index += 1
      return count
```

Test.py (main code):

```
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print ("The subseq occurs", result, "times")
```

Keep your functions in a separate file

useful_fns.py (custom module script):

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
      seq = seq.upper()
      subseq = subseq.upper()
      count = 0
      index = 0
      done = False
      while not done:
            index = seq.find(subseq, index)
            if (index == -1):
                  done = True
            else:
                  count += 1
                  index += 1
      return count
```

Test.py (main code):

```
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print ("The subseq o
```

we save the file of functions as useful_fns.py, but then import it using just the file name (no .py). Then we can access the functions in this file by saying `useful_fns.functionName()`

Keep your functions in a separate file

useful_fns.py (custom module script):

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
      seq = seq.upper()
      subseq = subseq.upper()
      count = 0
      index = 0
      done = False
      while not done:
            index = seq.find(subseq, index)
            if (index == -1):
                  done = True
            else:
                  count += 1
                  index += 1
      return count
```

Test.py (main code):

```
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print ("The subseq occurs", result, "times")
```

Result:

```
> python test.py
Full sequence: CGCACGCACGCGC
Subseq to search for: CGC
The subseq occurs 4 times
```

4. Useful modules for writing scripts

4a. argparse

argparse

Purpose: Command line parsing.

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"):

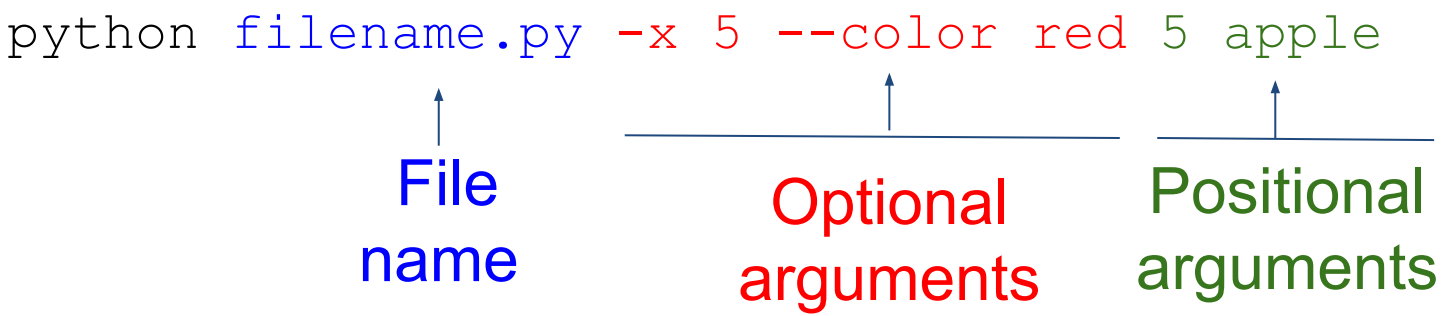
```
python filename.py -o arg1 arg2 agr3
```

More info:

https://docs.python.org/3/howto/argparse.html

Command line arguments

Your script can take as many command line arguments as you want. `argparse.ArgumentParser()` handles these arguments by separating them into positional and optional arguments.



Using `argparse`

- Before using command line args, we must import `argparse`:

```
import argparse
```

Using `argparse`

- Before using command line args, we must import `argparse`:

```
import argparse
```

- Then we can initialize our Argument parser by typing:

```
parser=argparse.ArgumentParser()
```

Using `argparse`

- Before using command line args, we must import `argparse`:

```
import argparse
```

- Then we can initialize our Argument parser by typing:

```
parser=argparse.ArgumentParser()
```

- To set up arguments you use call the `add_argument` function with the initialized ArgumentParser

```
parser.add_argument("square", type=int,
    help="display a square of a given number")
```

Using `argparse`

- Before using command line args, we must import `argparse`:

```
import argparse
```

- Then we can initialize our Argument parser by typing:

```
parser=argparse.ArgumentParser()
```

- To set up arguments you use call the `add_argument` function with the initialized ArgumentParser

```
parser.add_argument("square", type=int,
    help="display a square of a given number")
```

- To interpret the arguments we parse the arguments to an object called “args”

```
args= parser.parse_args()
```

Adding Arguments

- Arguments are by default, **positional arguments** (the order you put them on the command line matters)

```
parser.add_argument("var1", type=int, help="an integer")
```

Adding Arguments

- Arguments are by default, **positional arguments** (the order you put them on the command line matters)

```
parser.add_argument("var1", type=int, help="an integer")
```

- Adding **--** to an argument name makes it **optional** by default

```
parser.add_argument("--var2", help="any string")
```

Adding Arguments

- Arguments are by default, **positional arguments** (the order you put them on the command line matters)

```
parser.add_argument("var1", type=int, help="an integer")
```

- Adding **--** to an argument name makes it **optional** by default

```
parser.add_argument("--var2", help="any string")
```

- You can also add short versions of the **optional arguments**, but the long version contains the name where the argument is stored

```
parser.add_argument("-v3","--var3",type=str, help="any string")
```

Example: argTest.py

argTest.py

```
import argparse

parser=argparse.ArgumentParser(description="test code")
parser.add_argument("var1", type=str, help="any string")
parser.add_argument("var2", type=str, help="any string")
parser.add_argument("-v3","--var3", type=str,
    help="any string")
args=parser.parse_args()

print(args.var1)
if args.var3:
    print(args.var3)
```

Script Description

Positional Argument

Positional Argument

Optional Argument

Command Line Help

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -h
```

Command Line Help

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", typ
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -h
usage: argTest.py [-h] [-v3 VAR3] var1 var2

test code

positional arguments:
  var1                  any string
  var2                  any string

optional arguments:
  -h, --help            show this help message and exit
  -v3 VAR3, --var3 VAR3
                        any string
```

Example command using `argparse`

argTest.py

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish cat dog
```

Example command using `argparse`

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish cat dog
```

Script name

var3 var1 var2

What will this code print?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish cat dog
```

Script name

var3 var1 var2

What will this code print?

argTest.py

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish cat dog
cat
fish
```

What if we **only** listed **positional** arguments?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py cat dog
```

What if we **only** listed **positional** arguments?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py cat dog
cat
```

What if we **only** listed **one positional** argument?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py cat
```

What if we **only** listed **one positional** argument?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py cat
usage: argTest.py [-h] [-v3 VAR3] var1 var2
argTest.py: error: the following arguments
are required: var2
```

What if we **reordered** dog and cat?

argTest.py

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish dog cat
```

What if we **reordered** dog and cat?

argTest.p
y

```
import argparse

parser=argparse.ArgumentParser(description="test code")

parser.add_argument("var1", type=str, help="any string")

parser.add_argument("var2", type=str, help="any string")

parser.add_argument("-v3","--var3", type=str,
    help="any string")

args=parser.parse_args()


print(args.var1)

if args.var3:

    print(args.var3)
```

Command line:

```
> python argTest.py -v3 fish dog cat
dog
fish
```

Other notes on command line args

- Separate positional args with a space
- The order of positional args matter, but the order of optional args do not. You can put them before or after the positional args.
- You don't need to put quotes around strings on the command line, *UNLESS* your input string contains white space(s)
- Everything is read in as a string, so numbers must be set by the **type** parameter in parser.add_argument()
- Don't use commas when specifying numbers (e.g. say 10000 instead of 10,000)

The add_argument() method

`ArgumentParser.`**`add_argument`**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- name or flags - Either a name or a list of option strings, e.g. `foo` or `-f, --foo`.
- action - The basic type of action to be taken when this argument is encountered at the command line.
- nargs - The number of command-line arguments that should be consumed.
- const - A constant value required by some action and nargs selections.
- default - The value produced if the argument is absent from the command line.
- type - The type to which the command-line argument should be converted.
- choices - A container of the allowable values for the argument.
- required - Whether or not the command-line option may be omitted (optionals only).
- help - A brief description of what the argument does.
- metavar - A name for the argument in usage messages.
- dest - The name of the attribute to be added to the object returned by `parse_args()`.

https://docs.python.org/3/library/argparse.html#name-or-flags

When to 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.

4b. sys

sys

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

1. Appending to your python path
2. Printing to standard output and standard error
2. Exiting the script early

Appending to your python path

- What is a python path?
  - A list of directories python does through to search for modules and files
- When would we append items to this path?
  - When we have a custom python module that is not in the same directory as the script that we plan to use

Append to Python Path with sys.path.append()

../script/useful_fns.py (custom module script):

```
# Count (potentially overlapping) instances of a subsequence in a string
def count_occurrences(seq, subseq):
      seq = seq.upper()
      subseq = subseq.upper()
      count = 0
      index = 0
      done = False
      while not done:
            index = seq.find(subseq, index)
            if (index == -1):
                  done = True
            else:
                  count += 1
                  index += 1
      return count
```

Test.py (main code):

```
import sys
sys.path.append("../scripts")
import useful_fns

seq = raw_input("Full sequence: ")
subseq = raw_input("Subseq to search for: ")
result = useful_fns.count_occurrences(seq, subseq)
print ("The subseq occurs", result, "times")
```

Tells python to look for custom module in this directory

Result:

```
> python test.py
Full sequence: CGCACGCACGCGC
Subseq to search for: CGC
The subseq occurs 4 times
```

stdout and stderr

Every command can send its output to one of two places:

- stdout - output messages (this is where the `print` statement is sent)
- stderr - error messages

Both are printed to your console unless directed elsewhere

Example: stdPrints.py

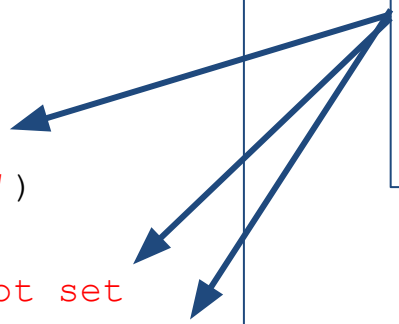
stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("-s", "--set_this",

    action="store_true", help="please set")

args=parser.parse_args()


if args.set_this:

    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Example: stdPrints.py

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("-s", "--set_this",

    action="store_true", help="please set")

args=parser.parse_args()


if args.set_this:

    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```



These functions do not automatically print a new line at the end of the string so we must use “\n” to make a new line.

What happens when we run this command?

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("-s", "--set_this",

    action="store_true", help="please set")

args=parser.parse_args()


if args.set_this:

    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Command line:

```
> python stdPrints.py -s
```

What happens when we run this command?

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()
parser.add_argument("-s", "--set_this",
    action="store_true", help="please set")
args=parser.parse_args()

if args.set_this:
    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Command line:

```
> python stdPrints.py -s
You set it!
Done!
```

***Unless standard output and/or standard error are redirected. They print to the screen.**

What happens when we run this command?

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("-s", "--set_this",

    action="store_true", help="please set")

args=parser.parse_args()


if args.set_this:

    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Command line:

```
> python stdPrints.py
You did not set it.
Done!
```

What happens when we run this command?

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("-s", "--set_this",

    action="store_true", help="please set")

args=parser.parse_args()


if args.set_this:

    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Command line:

```
> python stdPrints.py -s >
stdout.txt 2> stderr.txt
```

What happens when we run this command?

stdPrints.py

```
import argparse, sys

parser=argparse.ArgumentParser()
parser.add_argument("-s", "--set_this",
    action="store_true", help="please set")
args=parser.parse_args()

if args.set_this:
    sys.stdout.write("You set it!\n")
else:
        sys.stderr.write("You did not set
   it.\n")
sys.stderr.write("Done!\n")
```

Command line:

```
> python stdPrints.py -s >
stdout.txt 2> stderr.txt
```

we redirect standard output and standard error

stdout.txt:

```
You set it!
```

stderr.txt:

```
Done!
```

Example: addPosNums.py

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

addPosNums.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("var1", type=int,
help="positive integer")

parser.add_argument("var2", type=int,
help="positive integer")

args=parser.parse_args()


if args.var1 < 0 or args.var2 < 0:

    sys.exit("var1 and var2 must be
positive")
else:
    print (args.var1 + args.var2)
```

Check that both var1 and var2 are greater than 0

If not, use this piece of code to immediately terminate the whole script. Prints string to standard error.

What happens when we run this command?

addPosNums.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("var1", type=int,
help="positive integer")

parser.add_argument("var2", type=int,
help="positive integer")

args=parser.parse_args()


if args.var1 < 0 or args.var2 < 0:

    sys.exit("var1 and var2 must be
positive")
else:
    print (args.var1 + args.var2)
```

Command line:

```
> python addPosNums.py 1 2
```

What happens when we run this command?

addPosNums.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("var1", type=int,
help="positive integer")

parser.add_argument("var2", type=int,
help="positive integer")

args=parser.parse_args()


if args.var1 < 0 or args.var2 < 0:

    sys.exit("var1 and var2 must be
positive")
else:
    print (args.var1 + args.var2)
```

Command line:

```
> python addPosNums.py 1 2
3
```

What happens when we run this command?

addPosNums.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("var1", type=int,
help="positive integer")

parser.add_argument("var2", type=int,
help="positive integer")

args=parser.parse_args()


if args.var1 < 0 or args.var2 < 0:

    sys.exit("var1 and var2 must be
positive")
else:
    print (args.var1 + args.var2)
```

Command line:

```
> python addPosNums.py 1 -2
```

What happens when we run this command?

addPosNums.py

```
import argparse, sys

parser=argparse.ArgumentParser()

parser.add_argument("var1", type=int,
help="positive integer")

parser.add_argument("var2", type=int,
help="positive integer")

args=parser.parse_args()


if args.var1 < 0 or args.var2 < 0:

    sys.exit("var1 and var2 must be
positive")
else:
    print (args.var1 + args.var2)
```

Command line:

```
> python addPosNums.py 1 -2
var1 and var2 must be positive
```

4c. 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:

http://docs.python.org/2/library/os.path.html

http://docs.python.org/2/library/os.html#module-os

os.path functions

```
>>> 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.mkdir("new_folder") #creates new directory
```

os.path functions

```
>>> import os
>>> os.path.abspath("test_file.txt") #gets absolute/full path of file
'C:/Users/Sammy/Dropbox/Python/PythonBootcamp2019/lab6/test_file.txt'

>>> 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
'C:/Users/Sammy/Dropbox/Python/PythonBootcamp2019/lab6'

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

Other useful modules

Built-in:

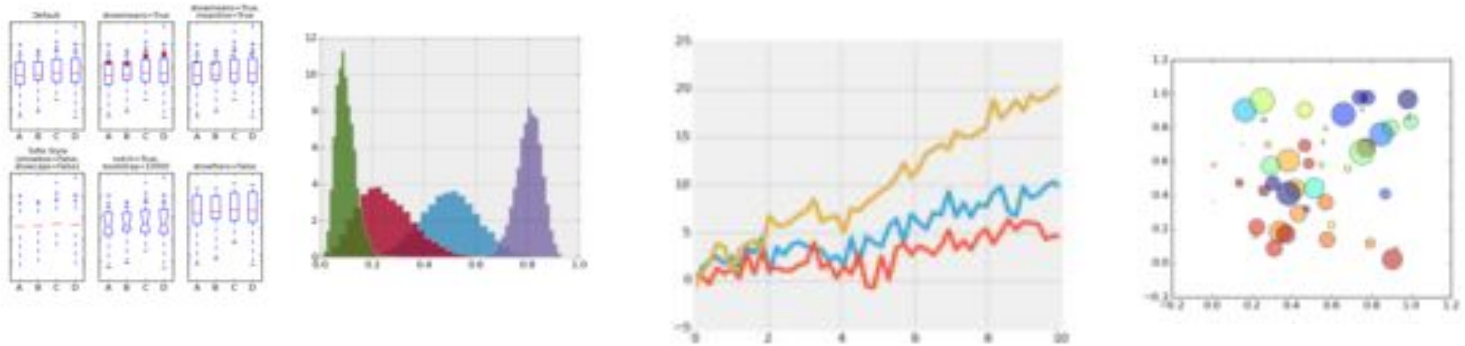
- `glob*` – getting lists of files
- `subprocess*` – system commands from within python
- `time*` - get the system time, create a timer
- `multiprocessing` – functions for writing parallel code that utilizes multiple CPU cores
- `optparse/argparse` – fancier command line args
- `re` – regular expressions (advanced pattern matching)
- `collections` – advanced data structures
- `logging` – facilitates the creation of log files
- `datetime` – for accessing/manipulating date & time info

* I have slides for these libraries at the end of the lecture that I won't go over today

Other useful modules

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

Next Class: Pandas Crash Course

- **You will need to have pandas and numpy installed!!!!!**
- To test if you already have these libraries installed type the following in your **jupyter notebook**. If you do not get errors, then both are already installed:

```
import pandas
import numpy
```

- If you installed jupyter notebooking using conda, you can install these libraries by typing the following into your **linux command line**:

```
conda install pandas
conda install numpy
```

conda install pandas

```
import pandas
import numpy
```

Slides on python libraries: re, glob, subprocess, and time

re

re

Purpose: To identify regular expressions (patterns) in strings

Example:

```
re.search(r"^>", ">seq1")
```

More info:

http://docs.python.org/2/library/re.html

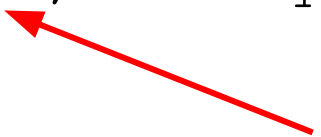
re.search(*pattern*, *string*)

Scans through *string* looking for the first location where the regular expression pattern produce a match

- Returns None if no match is found
- Otherwise, returns a match object which can be used in if/else statements

re.search(*pattern*, *string*)
example

```
import re

if re.search(r"^>", ">seq1"):
   print("Match")
else:
   print("no Match")
```



This code represents a pattern. We put an r in front of the string to make sure it is a raw string. The `^` character checks to see if the pattern is at the beginning of the line. (Note: this function may be used for fasta files)

Regular expression basics

.	any character except newline
a	the character a
ab	the string ab
a|b	a or b
a*	0 or more a’s
\	escapes a special character

https://www.debuggex.com/cheatsheet/regex/python

Regular expression quantifiers

*	0 or more
+	1 or more
?	0 or 1

https://www.debuggex.com/cheatsheet/regex/python

Regular expression character classes

[ad-f]	a character of: a, d, e, f
[^ab-d]	a character of: a, b, c, d
[0-5]	any number 0-5
\s	a white space
\w	a word character

https://www.debuggex.com/cheatsheet/regex/python

Regular expression assertions

^	start of string
$	end of string

https://www.debuggex.com/cheatsheet/regex/python

Practice with Regular Expressions

```
import re

if re.search(r"GA", "AGG"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"GA", "AGG"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
   no Match
```

Practice with Regular Expressions

```
import re

if re.search(r"T|C", "AGG"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"T|C", "AGG"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
   no Match
```

Practice with Regular Expressions

```
import re

if re.search(r"A|C", "AGG"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"A|C", "AGG"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
   Match
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]", "hello world"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]", "hello world"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
Match
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]{1}", "hello world"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]{1}", "hello world"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
Match
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]{2}", "hello world"):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re

if re.search(r"[\s]{2}", "hello world"):
   print("Match")
else:
   print("no Match")
```

Returns:

```
no Match
```

Practice with Regular Expressions

```
import re
dna="ATCGCGGATCCA"
if re.search(r"GG[AT]CC", dna):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re
dna="ATCGCGGATCCA"
if re.search(r"GG[AT]CC", dna):
   print("Match")
else:
   print("no Match")
```

Returns:

```
no Match
```

Practice with Regular Expressions

```
import re
dna="ATCGCGGACCA"
if re.search(r"GG[AT]CC", dna):
   print("Match")
else:
   print("no Match")
```

Practice with Regular Expressions

```
import re
dna="ATCGCGGACCA"
if re.search(r"GG[AT]CC", dna):
   print("Match")
else:
   print("no Match")
```

Returns:

```
Match
```

Help with regular expression

- The syntax for regular expressions can get complicated even for advanced programmers
- There are tools online to test your regular expressions
  - https://regex101.com/

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")
```

More info:
http://docs.python.org/2/library/glob.html

Important to note:

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

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.

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/2/library/subprocess.html

http://stackoverflow.com/questions/89228/calling-an-external-command-in-python

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 - 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."
```

(in useful_fns.py)

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.

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
```

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.

More info:

http://docs.python.org/2/library/time.html

Extra Python Life Hacks

+=

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 "Wilfred"
    name += c #same as name = name + c
```

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/2/library/stdtypes.html#string-methods
    - `string.count()`
    - `string.upper()` / `string.lower()`
    - `string.find()`
    - `string.join()`
  - `random.choice()`
  - many more