EMORY
UNIVERSITY
SCHOOL OF
MEDICINE

Department of
Biomedical Informatics

BMI 500:

Introduction to Biomedical Informatics

Lecture 3. Introduction to Programming in Python

https://tinyurl.com/bmi500

7th Sept 2022

Rishikesan Kamaleswaran
Department of Biomedical Informatics, Emory University, Atlanta, GA USA
Department of Biomedical Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Objectives

1. To understand the basic structure and syntax of Python
2. To learn how to execute python code on your computer and on the computational cluster
3. To write your own python scripts
4. To allow you to progress onto more advanced training

Python Programming Language

- Python 3.0 first released in 2008, builds on a long history of earlier versions from the 1980s.
  - Python 2.x officially sunset
- Interpretable and higher-level programming language
- Object oriented language with dynamic type setting
- Automatic memory management
- Simple syntax, but limited expressibility
  - No free lunch?
- Platform agnostic

“Hello World”

Python

```
print("Hello World")
```

Java

```
class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

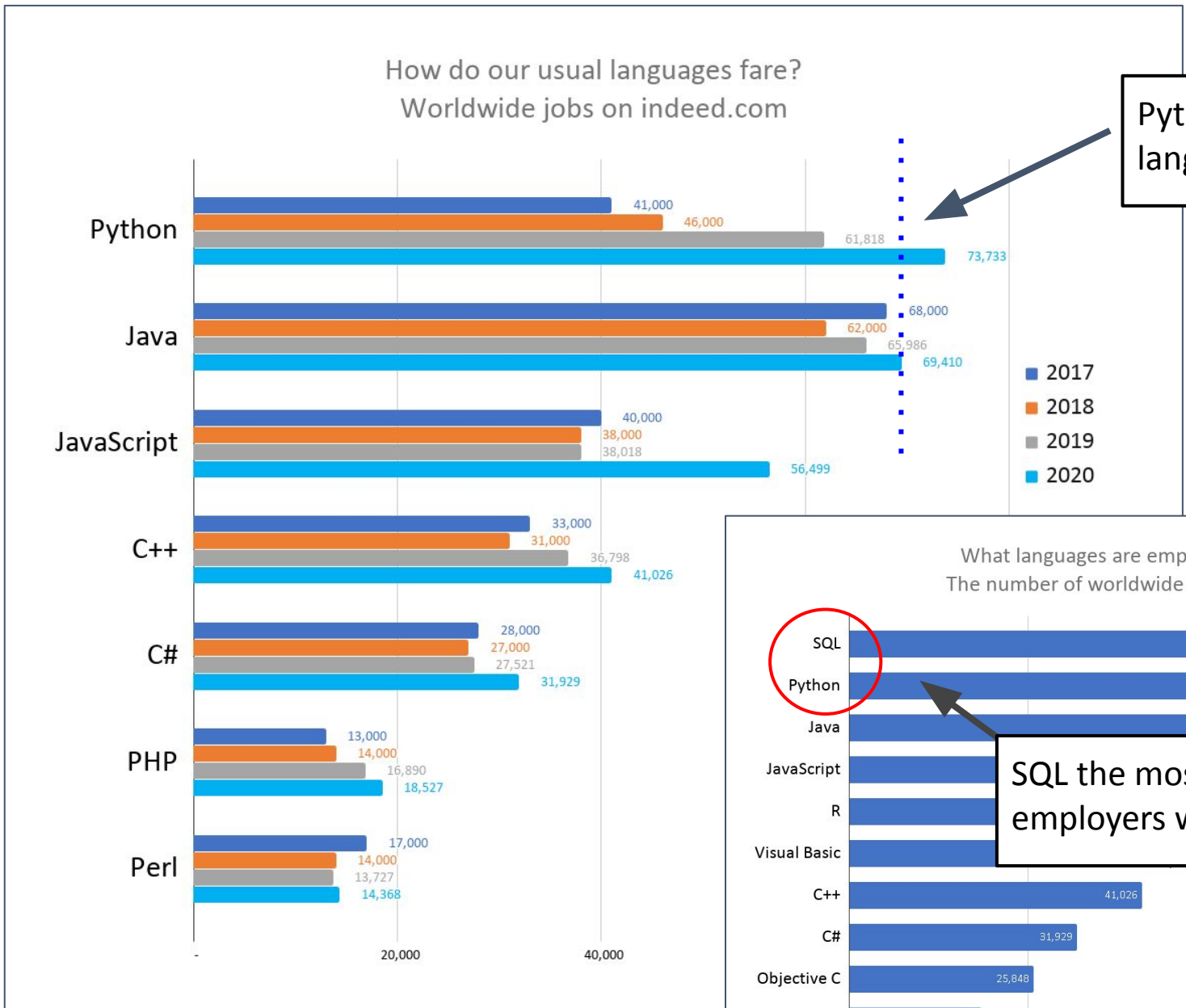
C

```
#include <stdio.h>
int main() {
   printf("Hello, World!");
   return 0;
}
```

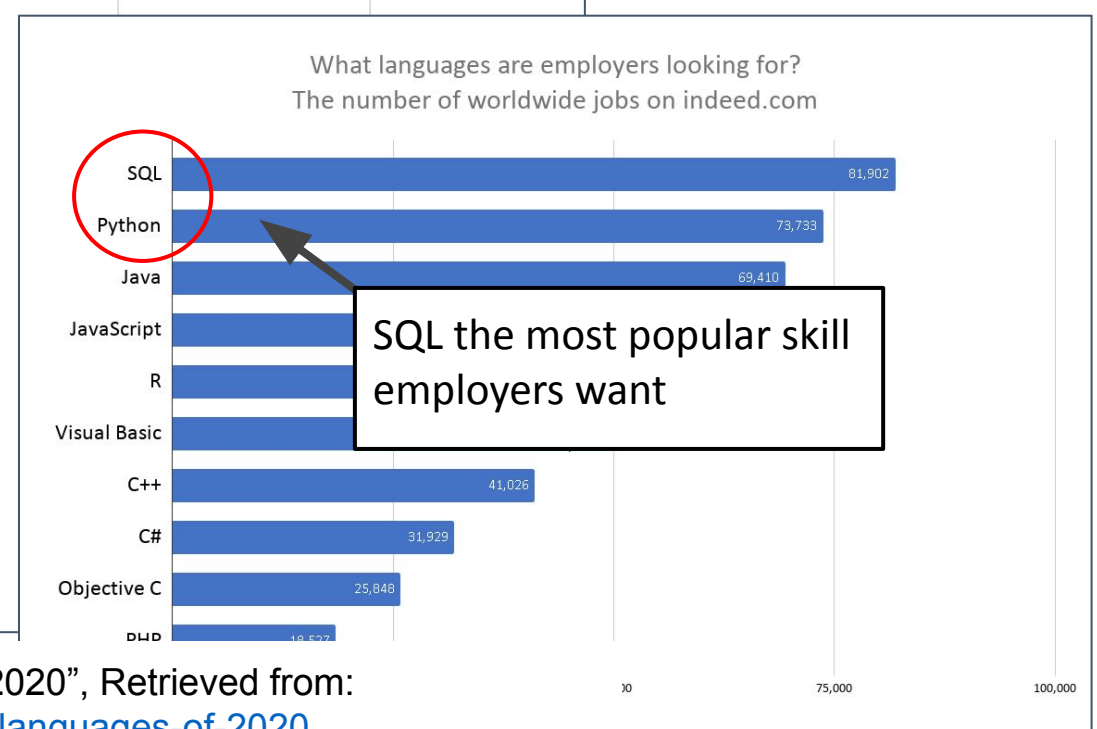
FORTRAN

```
program hello
        implicit none
        print *, "Hello World!"
      end program
```

Market asketh-- we giveth; Popularity contest 2020



Python the most popular language in 2020



SQL the most popular skill employers want

CodingDojo Inc. “Top 7 Programming Languages Of 2020”, Retrieved from: https://www.codingdojo.com/blog/top-7-programming-languages-of-2020

Read/Write IO

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

Python uniqueness - Pointers

- All Python variables are ***pointers***
- Assigning variables is as easily as a character to the left of the '='
  - var = 1111
- Unlike in C or Java, this variable is a *pointer* to the value, rather than a *container*
  - This essentially means you don't need to explicitly declare the variable
    - I.e. int var = 1111
- It also means if a subsequent pointer *x* is defined using pointer *var*; all changes will be reflected in downstream variables

Python uniqueness - Objects

- All Python variables are objects
- Every declared variable has ‘metadata’ that encodes for type

```
>>> var= 'testing'
>>> type(var)
>>> “str”
```

- Each variable also comes with free ‘methods’:
  - dtypes
  - Lists: .append(); .copy(); .pop(); .clear(); .index(); etc...

Python uniqueness - Indentation

- Typically indentation in programming languages are for documentation and readability, many languages are often minimized without any formatting, i.e. javascript:

```
!function(t,n){"object"==typeof exports&&"undefined"!=typeof module?n(exports):"function"==typeof
define&&define.amd?define(["exports"],n):n((t=t||self).d3=t.d3||{})}(this,function(t){"use strict";function n(t,n){return t<n?-1:t>n?1:t>=n?0:NaN}function
e(t){var e;return 1===t.length&&(e=t,t=function(t,r){return n(e(t),r)}),{left:function(n,e,r,i){for(null==r&&(r=0),null==i&&(i=n.length);r<i;){var
o=r+i>>>>1;t(n[o],e)<0?r=o+1:i=o}return r},right:function(n,e,r,i){for(null==r&&(r=0),null==i&&(i=n.length);r<i;){var
o=r+i>>>>1;t(n[o],e)>0?i=o:r=o+1}return r}}}var r=e(n),i=r.right,o=r.left;function a(t,n){return[t,n]}function u(t){return null===t?NaN:+t}function c(t,n){var
e,r,i=t.length,o=0,a=-1,c=0,f=0;if(null==n)for(;++a<i;)isNaN(e=u(t[a]))||(f+=(r=e-c)*(e-(c+=r/++o)));else
for(;++a<i;)isNaN(e=u(n(t[a],a,t)))||(f+=(r=e-c)*(e-(c+=r/++o)));if(o>1)return f/(o-1)}function f(t,n){var e=c(t,n);return e?Math.sqrt(e):e}function s(t,n){var
e,r,i,o=t.length,a=-1;if(null==n){for(;++a<o;)if(null!=(e=t[a])&&e>=e)for(r=i=e;++a<o;)null!=(e=t[a])&&(r>e&&(r=e),i<e&&(i=e))}else
for(;++a<o;)if(null!=(e=n(t[a],a,t))&&e>=e)for(r=i=e;++a<o;)null!=(e=n(t[a],a,t))&&(r>e&&(r=e),i<e&&(i=e));return[r,i]}var
l=Array.prototype,h=l.slice,d=l.map;function p(t){return function(){return t}}function v(t){return t}function
g(t,n,e){t=+t,n=+n,e=(i=arguments.length)<2?(n=t,t=0,1):i<3?1:+e;for(var r=-1,i=0|Math.max(0,Math.ceil((n-t)/e)),o=new Array(i);++r<i;)o[r]=t+r*e;return
o}var y=Math.sqrt(50),_=Math.sqrt(10),b=Math.sqrt(2);function m(t,n,e){var
r,i,o,a,u=-1;if(e=+e,(t=+t)===(n=+n)&&e>0)return[t];if((r=n<t)&&(i=t,t=n,n=i),0===(a=x(t,n,e))||!isFinite(a))return[];if(a>0)for(t=Math.ceil(t/a),n=Math.floor
(n/a),o=new Array(i=Math.ceil(n-t+1));++u<i;)o[u]=(t+u)*a;else for(t=Math.floor(t*a),n=Math.ceil(n*a),o=new Array(i=Math.ceil(t….
```

- Python uses indentation to indicate blocks of code
  - Other languages indicated by { … }
  - Spacing must be ***consistent*** within each block of code
  - **Variable Scoping** is limited to each block

Comments

- Comments start with a #

```
#Comments can be written in blocks
# No spacing constraints here
print("This is a test")
```

- They can be placed at the end of a line

```
print("This is a test") #Comments also inserted inline
```

- Triple quotes can be used to comment multiple lines

```
"""
This is a comment
This is a comment 2
This is a comment 3
"""
print("This is a test")
```

String processing

- "hello" + "world" >>> "helloworld" # Concatenation
- "hello"*3 >>> "hellohellohello" # Repetition
- "hello"[0] >>> "h" # indexing
- "hello"[-1] >>> "o" # (from end)
- "hello"[1:4] >>> "ell" # slicing [start:end]
- len("hello") >>> 5 # size of the string
- "hello" < "jello" >>> True # comparison, boolean condition
- "e" in "hello" >>> 1 # search; returns the index

- New line: "escapes: \n "
- Line continuation: triple quotes ’’’
- Quotes: ‘single quotes’, "raw strings"

String methods

- upper()
- lower()
- capitalize()
- count(s)
- find(s)
- rfind(s)
- index(s)
- strip(), lstrip(), rstrip()
- replace(a, b)
- expandtabs()
- split()
- join()
- center(), ljust(), rjust()

List

- A container that holds a number of other objects, in a given order
- Defined in square brackets

```
>>> t= [1, 2, 3, 4]
>>> print(t[1]) #Retrieve element at index 1
>>> 2

## Advanced functions
>>> t= []
>>> t.append(1) #Insert an element to the list
>>> t.append(2) #Insert after previous element
>>> print(len(t))
>>> 2
```

List

- More functions of list

```
>>> a = range(5)   # [0,1,2,3,4]
>>> a.append(5)   # [0,1,2,3,4,5]
>>> a.pop()   # [0,1,2,3,4]
>>> 5
>>> a.insert(0, 5.5)   # [5.5,0,1,2,3,4]
>>> a.pop(0)   # [0,1,2,3,4]
>>> 5.5
>>> a.reverse()   # [4,3,2,1,0]
>>> a.sort()   # [0,1,2,3,4]
```

List

- More functions of list

- append
- insert
- index
- count
- sort
- reverse
- remove
- pop
- extend

- Indexing e.g., L[i]
- Slicing e.g., L[1:5]
- Concatenation e.g., L + L
- Repetition e.g., L * 5
- Membership test e.g., ‘a’ in L
- Length e.g., len(L)

Nested List

- List consisting of other list(s)

```
>>> a = [“hello”, “world”]
>>> b = [20, 21]
>>> a.append(b) # ['hello', 'world', [20, 21]]
>>> a[2] # [20, 21] indexed by a list
>>> a[2][1] # 20 at index of the nested list
```

Dictionaries

- Key-value pairing, also called “associative arrays”
- Unordered set of keys, of varying data types
- Defined by { }
- Keys must be immutable
  - hash table implementation of dictionaries uses a hash value calculated from the key value to find the key

```
>>> a = { “fname”: “Alpha”,
          “lname”: “Raj”,
          “Age”: 19,
          3: “Secret message”
        }
>>> a[“fname”] # “Alpha”
>>> a[3] # “Secret Message”
```

Dictionaries

Available functions:

- keys()
- values()
- items()
- has_key(key)
- clear()
- copy()
- get(key[,x])
- setdefault(key[,x])
- update(D)
- popitem()

Tuples (immutable arrays)

- Ordered set of collections that cannot be modified once created
  - **Howeve**r** their values may change
- **Operations**: Indexing, Slicing, Concatenation, Repetition, Membership test e.g., ‘a’ in T, Length

```
>>> a = (1, [2, 3]) # Declare the tuple
>>> b = a[1] # Saves a pointer to the array
>>> b.append(4) # Appending an element
>>> print(a) # (1, [2, 3, 4])
```

Comparators/Operators

Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Operator	Description
and/&	Returns True if both statements are true
or/|	Returns True if one of the statements is true
not/~	Reverse the result, returns False if the result is true

Operations

Operator	Description	Example
=	Assignment	num **=** 7
+	Addition	num = 2 **+** 2
-	Subtraction	num = 6 **-** 4
*	Multiplication	num = 5 * 4
/	Division	num = 25 **/** 5
%	Modulo	num = 8 **%** 3
**	Exponent	num = 9 ** 2

Operations order

Order generally follows as follows:

- Brackets (inner before outer)
- Exponent
- Multiplication, division, modulo
- Addition, subtraction

- If multiple operations of the same order is defined then precedence is from left to right
- Always a good idea to bracket your operations!
  - e.g. (2 +1) * (3 / 2)

Print descriptors

Descriptor code	Type of Information to display
%s	String
%d	Integer (d = decimal / base 10)
%f	Floating point (%<width>.<precision>)

```
>>> number = 13.5255
>>> print(("%5.1f" %number) # 13.5
>>> print(("%5.2f" %number) # 13.53
```

Functions

Functions are a re-usable block of code that run when called, they can include arguments which control the output of the function

```
def month(x):
  print("The month is: " + x)

month("April")
month("May")
month("June")

>>> "The Month is April"
>>> "The Month is May"
>>> "The Month is June"
```

```
def month(x,y):
  print("Month is: " + x + ", Year " + y)

month("April","2020")
month("May","2022")
month("June","2021")

>>> "The Month is April, Year 2020"
>>> "The Month is May, Year 2022"
>>> "The Month is June, Year 2021"
```

Functions: variable scope and return values

```
In [4]: def nextMonth(x):
        # return the following month.

            # create a list of the months. this variable is not accessible in the global environment.
            months = ["January", "February", "March", "April", "May", "June", \
            "July", "August", "September", "October", "November", "December"]

            # test whether the supplied value is within the list of known Months.
            # returns a boolean.
            knownMonth = x in months

            # check and see whether the month supplied is in the list.
            # if it is not, return early with the special value None
            if knownMonth:
                print("The current month is: " + x + ".")
            else:
                print("Month " + x + "not recognized.")
                return None

            # code executed from here on will only execute if knownMonth is true -
            # otherwise it would have returned already. we could put in a second
            # conditional, but it would not be useful.

            # since the month is in the list, find which one it is.
            # you can use the "index" function built into the list class.
            currentMonthIndex = months.index(x)

            # add one to the index. use the "modulo" function so it wraps back around to zero.
            nextMonthIndex = (currentMonthIndex + 1) % len(months)

            # return the following month to the calling environment.
            return months[nextMonthIndex]

In [5]: nextMonth("April")

        The current month is: April.

Out[5]: 'May'

In [7]: nextMonth("December")

        The current month is: December.

Out[7]: 'January'

In [8]: months

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-8-fc2db21a4624> in <module>
        ----> 1 months

        NameError: name 'months' is not defined
```

defined in function
variable scope

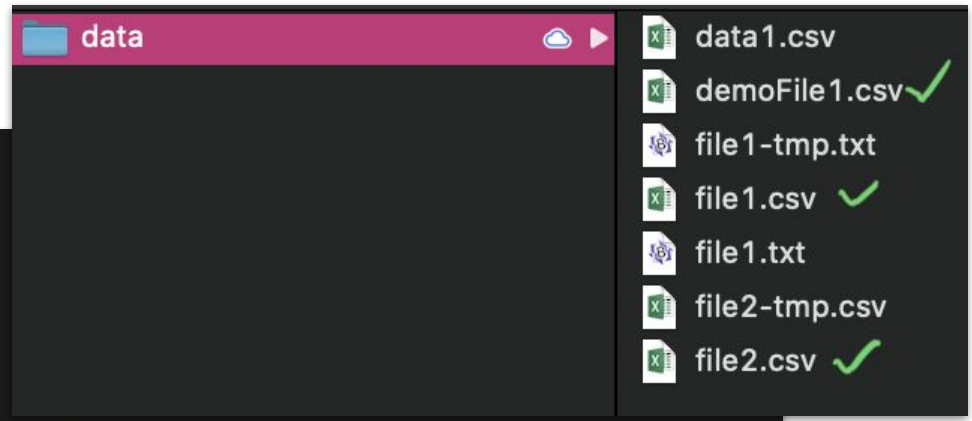
return here if
bad input

list.index() is neat
note 12 % 12 → 0,
"January"

Works ✓

not available
outside

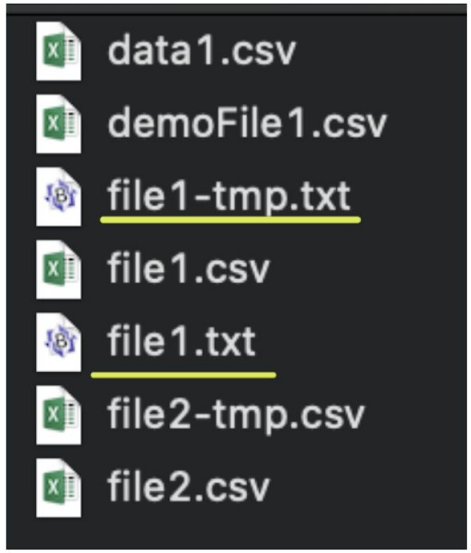
pathlib contains a lot of functionality for interactive with paths and files




```
#! python3

from pathlib import Path
import re

# let's say we want to perform an operation on everything
# called either file1.csv, file2.csv, or demoFile1.csv,
# while ignoring the .txt files and the -tmp files.

# define "data" as a Posix Path, which is easier to use than a string:
srcPath = Path("data/")
# >>> srcPath
# PosixPath('data')

# but which can be easily cast as a string:
# >>> str(srcPath)
# 'data'
```

-> a lot of this can be done similarly in bash/zsh, but if you need to aggregate data about the files themselves python data structures are much more useful.

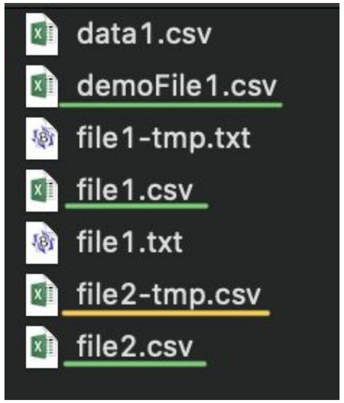
Path objects can be “globbed” to identify contents that match conditions.




```
# "glob" all of the text files from the Path object that match the glob *.txt:
[d for d in srcPath.glob("*.txt")]
# [PosixPath('data/file1.txt'), PosixPath('data/file1-tmp.txt')]   Paths

# notice that we can cast as string if those are necessary for inclusion in processing logs etc.
[str(d) for d in srcPath.glob("*.txt")]
# ['data/file1.txt', 'data/file1-tmp.txt']   'strings'
```

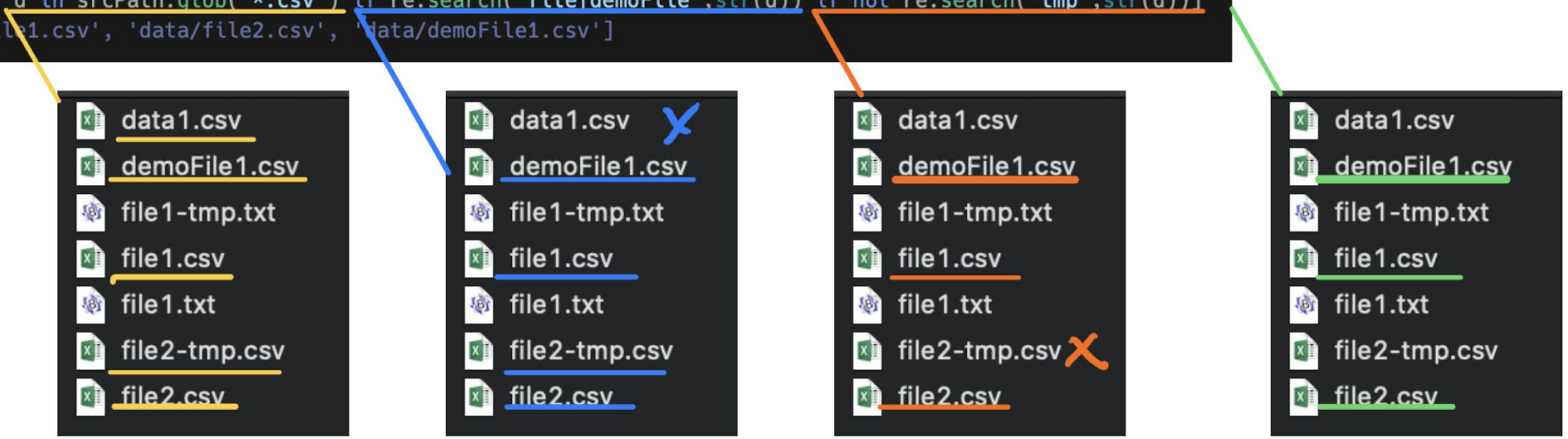
Path objects are not strings but can easily be cast with str().

re.search within the list of Path objects can precisely filter files




```
# Open up the glob to capture all *.csv files, but add a regular expression
# so that among the "*.csv" files, only files matching "file" or "demoFile" are included.
# in regexp, you "file" or "demoFile" is spelled "file|demoFile" with the "|" character
# designating "or"
[str(d) for d in srcPath.glob("*.csv") if re.search('file|demoFile',str(d))]
# ['data/file1.csv', 'data/file2.csv', 'data/file2-tmp.csv', 'data/demoFile1.csv']
```

```
# great! now the only thing left is to exclude that file labeled -tmp. to tell python to do that,
# you can chain another condition on the end of the list comprehension that says to reject anything
# previously identified that includes "tmp" in the filename.
[str(d) for d in srcPath.glob("*.csv") if re.search('file|demoFile',str(d)) if not re.search('tmp',str(d))]
# ['data/file1.csv', 'data/file2.csv', 'data/demoFile1.csv']
```



Functions: Lambda

Lambda functions are a class of anonymous functions in python, allowing you to complete simple tasks in a single line of code

```
>>> t= lambda x: x * 2
>>> t(2)
>>> 4

## Multiple args

>>> t= lambda x,y: (x+y) * 2
>>> t(2,3)
>>> 10
```

Loops: For Loop

Loops allow you to iterate over a sequence (list, tuple, dictionary, set, or string)

```
courses = ["algebra", "music", "drama", "science"]
for x in courses:
  print(x)

>>> algebra
>>> music
>>> drama
>>> science
```

Loops: While loop

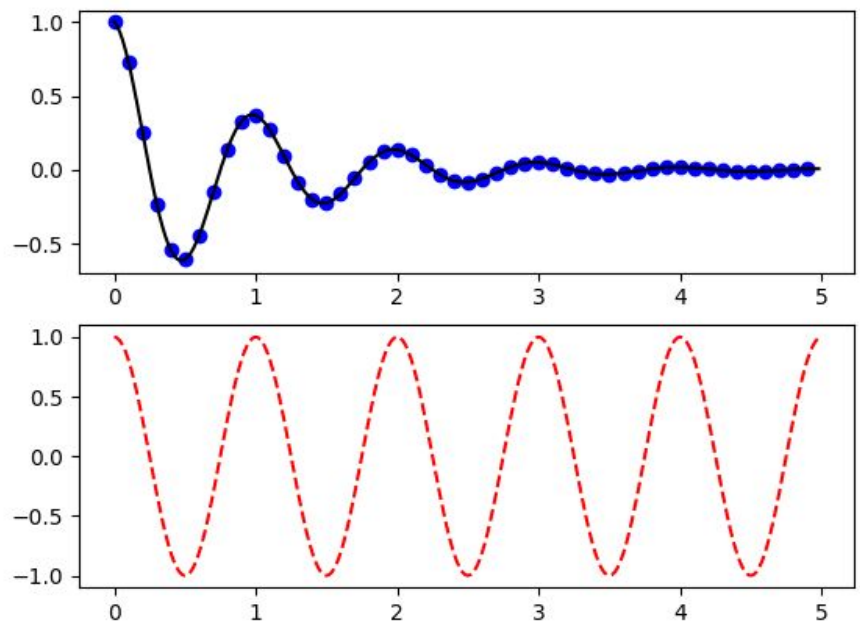
While Loops allow you to iterate over a sequence (list, tuple, dictionary, set, or string), when a condition remains true

```
def loop()
    i = 1
    while i < 6:
      print(i)
      i += 1

>>> loop() # 1 2 3 4 5
```

Visualization/Graphing

- A variety of packages exist to support visualization/graphing, the most popular is Matplotlib
- Further instructions: https://matplotlib.org/tutorials/introductory/pyplot.html

```
def f(t):
    return np.exp(-t) *
np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure()
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2),
'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
 plt.show()
```



Pickling your objects

- Python allows for serializing and de-serializing a Python object structure conveniently.
- The Python pickle uses a compact binary representation, and has advantages over text serialization methods such as JSON.
- The module ***pickletools*** contains tools for analyzing data streams generated by pickle.

The following types can be pickled:

- `None`, `True`, and `False`
- integers, floating point numbers, complex numbers
- strings, bytes, bytearrays
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module (using `def`, not `lambda`)
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose `__dict__` or the result of calling `__getstate__()` is picklable (see section Pickling Class Instances for details).

Refer: https://docs.python.org/3/library/pickle.html

Python pickle tutorial

To use pickle, start by importing it in Python.

import pickle

```
## Pickle a dictionary object

dogs_dict = { 'Ozzy': 3, 'Filou': 8, 'Luna': 5, 'Skippy': 10, 'Barco': 12, 'Balou': 9, 'Laika': 16 }

filename = 'dogs'
outfile = open(filename,'wb')

pickle.dump(dogs_dict,outfile)
outfile.close()
```

```
## Unpickle a dictionary object

infile = open(filename,'rb')
new_dict = pickle.load(infile)
infile.close()

print(new_dict)
print(new_dict==dogs_dict)
print(type(new_dict))
```

Ref: https://www.datacamp.com/community/tutorials/pickle-python-tutorial

Python package structure

Dissemination of code is important, therefore one must learn to package code for external use.

- Python has a default package structure:
  - https://packaging.python.org/tutorials/packaging-projects/
- The packaged structure allows for installing the module in external systems and utilize the code without transport errors
- Users may be able to import the entire package or subcomponents of the packages, i.e. specific modules.

```
packaging_tutorial/
├── LICENSE
├── pyproject.toml
├── README.md
├── setup.cfg
├── src/
│   └── example_package/
│       ├──   init  .py
│       └── example.py
└── tests/
```

Anaconda and virtual environments

- Python manages libraries and environments dynamically, therefore the chance of conflicts can be encountered often
- If you have multiple instances of python installed, these libraries can be downloaded to a variety of locations
- In order to ensure that your environment is protected from these conflicts, python has afforded the option to create ***virtual environments***

```
>> python3 -m venv env
>> source env/bin/activate
>> python3 hello_world.py
>> deactivate
```

Coding etiquette/Best practises

- Write readable code
- Use virtual environments
  - Avoid library conflicts
- Create code repositories and maintain good versioning
  - License/Read Me/etc.
- Create readable documentation
  - Helps when you need to refer back to code months later
- Repeatedly refer to the Python Style Guide, adhere religiously!
  - https://www.python.org/dev/peps/pep-0008/
- Fix code issues immediately!
- **Always assume the code will get audited during scientific review!**

Further instructions

- Python Homepage
  - http://www.python.org
- Python Tutorial
  - http://docs.python.org/tutorial/
- Python Documentation
  - http://www.python.org/doc
- Python Library References

  http://docs.python.org/release/2.5.2/lib/lib.html

Interactive Lab!

1. Download “Anaconda” on your computers (20 mins):

   https://www.anaconda.com/products/individual

1. Launch Jupyter Notebook
2. Complete pandas tutorial exercises (45 mins):

   https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

   https://pandas.pydata.org/pandas-docs/stable/user_guide/cookbook.html

1. Complete scikit-learn tutorial (10 mins):

   https://scikit-learn.org/stable/tutorial/index.html

1. Gentle ML pipelines (20 mins):

   https://scikit-learn.org/stable/tutorial/basic/tutorial.html

Homework

Homework quiz:
https://docs.google.com/document/d/1unkcFHgwTZ1_kIVHsve_8FWtJM1k76tkUvI69AY0bAg/edit