

# Introduction to Python

## Part 2

v0.9

Research Computing Services  
Information Services & Technology



# Tutorial Outline – Part 2

- If / else
- Classes
- Loops
- Tuples and dictionaries
- Modules
- Some useful modules
- Script setup
- Development notes

# If / Else

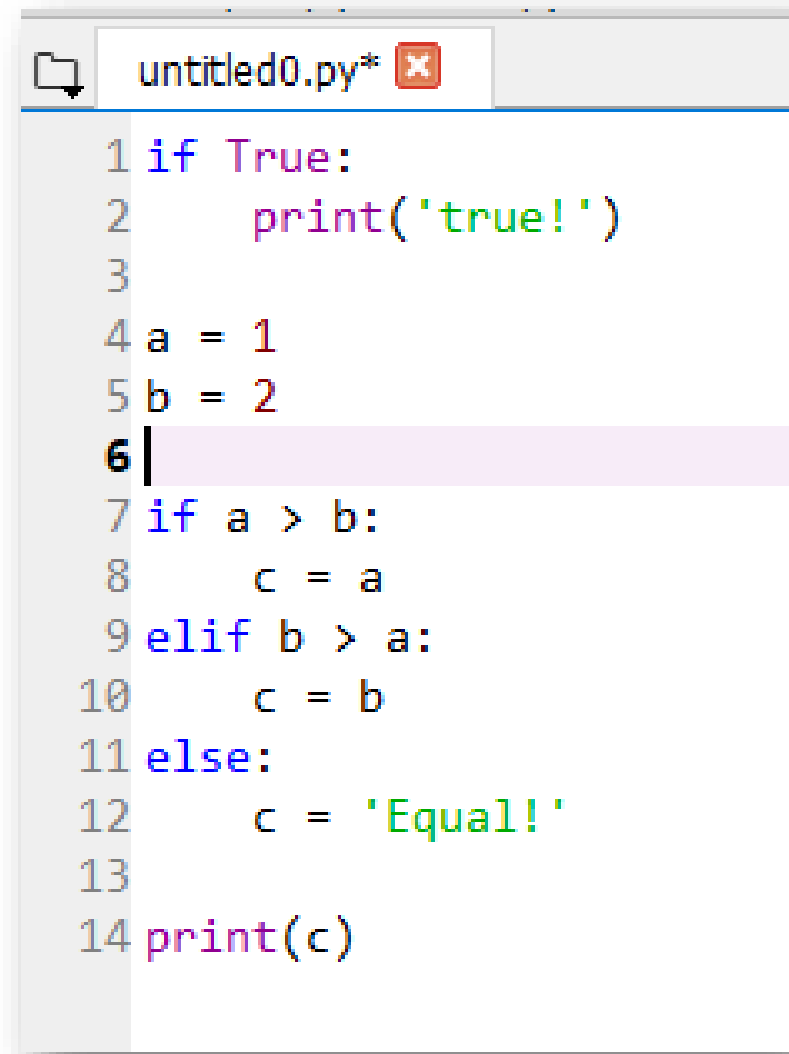
- *If*, *elif*, and *else* statements are used to implement conditional program behavior

- Syntax:

```
if Boolean_value:  
    ...some code  
elif Boolean_value:  
    ...some other code  
else:  
    ...more code
```

- *elif* and *else* are not required – use them to chain together multiple conditional statements or provide a default case.

- Try out something like this in the Spyder editor.
- Do you get any error messages in the console?
- Try using an *elif* or *else* statement by itself without a preceding *if*. What error message comes up?



The screenshot shows a Spyder Python IDE window titled 'untitled0.py\*'. The code in the editor is as follows:


```
1 if True:
2     print('true!')
3
4 a = 1
5 b = 2
6
7 if a > b:
8     c = a
9 elif b > a:
10    c = b
11 else:
12    c = 'Equal!'
13
14 print(c)
```

Line 6 is highlighted with a light purple background, indicating a syntax error. The error is a missing colon at the end of the line.

# If / Else code blocks

- Python knows a code block has ended when the indentation is removed.
- Code blocks can be nested inside others therefore *if-elif-else* statements can be freely nested within others.
  - Or used in functions...

```
a = 1
b = 2
if a <= b:
    c = a
    print('a <= b')
    if c == 1:
        print('c is 1')
print('out of the if statement')
```



# Project Euler Problem 1

- Let's code this!

## Multiples of 3 or 5

### Problem 1



If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

```
def euler1_(N):  
    ''' Sum of all natural numbers  
    from 1 to N-1 that are multiples of  
    3 or 5. '''  
    # hint: use a for loop and the range()  
    # function.  
    return ...your answer...  
  
# Prints 23  
print(euler1(10))  
# Prints 233168  
print(euler1(1000))
```

# Python Classes

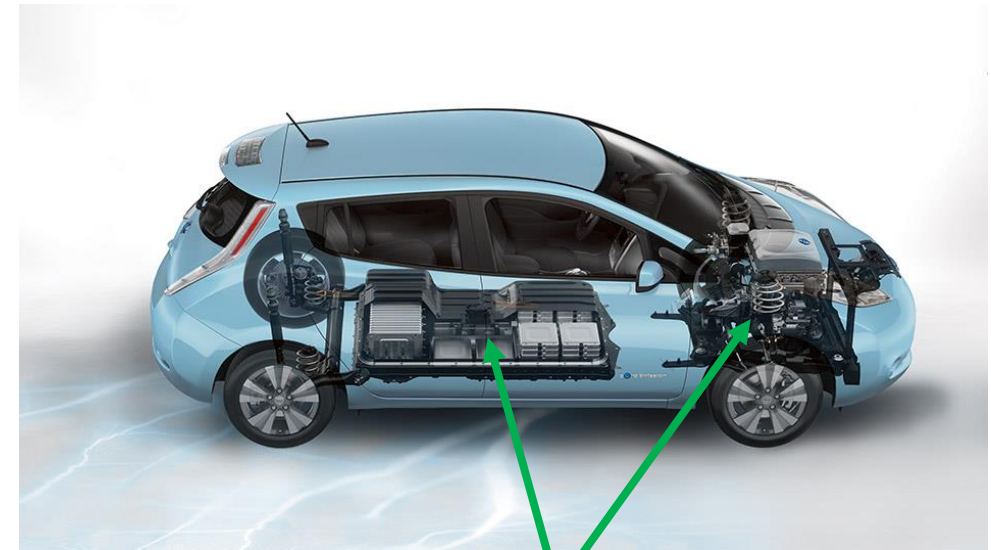
- OOP: Object Oriented Programming
- In OOP a *class* is a data structure that combines data with functions that operate on that data.
- An *object* is a variable whose type is a *class*
  - Also called an *instance* of a class
- Classes provide a lot of power to help organize a program and can improve your ability to re-use your own code.

# Object-oriented programming

- Classes can contain data and methods (internal functions).
- Methods can call other code inside the class to implement complex behavior.
- This is a highly effective way of modeling real world problems inside of a computer program.

“Class Car”

public interface



internal data and methods



# Object-oriented programming

- Python is a fully object oriented programming (OOP) language.
- Some familiarity with OOP is needed to understand Python data structures and libraries.
- You can write your own Python classes to define custom data types.

Boston	
Population	685094
Area (km <sup>2</sup> )	232.1
# of colleges	35

Track via separate variables

```
boston_pop = 685094  
boston_sq_km = 232.1  
boston_num_colleges = 35
```

A class lets you bundle these into one variable

# Writing Your Own Classes

- Define your own Python classes to:
  - Bundle together logically related pieces of data
  - Write functions that work on specific types of data
  - Improve code re-use
  - Organize your code to more closely resemble the problem it is solving.

```
class City:
    ''' A class to hold info about a city '''
    def __init__(self, name, area, pop, num_colleges):
        self.name = name
        self.area = area
        self.pop = pop
        self.num_colleges = num_colleges

    def is_best_city(self):
        return self.name == 'Boston'

boston = City('Boston', 685094, 232.1, 35)
new_york = City('New York City', 8804190, 1223.59, 120)

print(new_york.is_best_city()) # prints False
```

# Syntax for using Python classes

Create an object, which is a variable whose type is a Python class.

Created by a call to the class or returned from a function.

Call a method for this object:

`object_name.method_name(args...)`

```
# Open a file. This returns a file object.
file = open('some_file.txt')

# Read all the lines from the text file.
# Return them as a list.
lines = file.readlines()

# Get the filename
file.name # --> some_file.txt
```

Access internal data for this object:

`object_name.data_name`

# Classes bundle data and functions

- In Python, calculate the area of some shapes after defining some functions.

```
radius = 14.0
width_square = 14.0
a1 = area_circle(radius)           # ok
a2 = area_square(width_square)     # ok
a3 = area_circle(width_square)     # !! OOPS
```

- If we defined Circle and Rectangle classes with their own *area()* methods...it is not possible to miscalculate.

- Group data with matching functions into classes.

```
class Circle
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14.159 * self.radius**2

class Square:
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width**2

c1 = Circle(radius)
r1 = Square(width_square)

a1 = c1.area()
a2 = r1.area()
```

# When to use your own class

- A class works best when you've done some planning and design work before starting your program.
- This is a topic that is best tackled after you're comfortable with solving programming problems with Python.
- Some tutorials on using Python classes:


W3Schools: [https://www.w3schools.com/python/python\\_classes.asp](https://www.w3schools.com/python/python_classes.asp)

Python tutorial: <https://docs.python.org/3.6/tutorial/classes.html>

# Strings Are a Class In Python

- Python defines a string class – **all strings in Python are objects.**
- This means strings have:
  - Their own internal (hidden) memory management to handle storage of the characters.
  - A variety of methods (functions) that operate on the stored string once you have a string object.
- You can't access string functions without a string – in Python the string provides its own functions.
  - C: strcat, strcmp, strlen functions
  - Matlab: strlen, isletter, etc
  - R: nchar, toupper, etc

# String functions

- In the Python console, create a string variable called *mystr*
- type: *dir(mystr)*
- Try out some functions: 
- Need help? Try:  
*help(mystr.title)*

```
mystr = 'Hello!'

mystr.upper()

mystr.title()

mystr.isdecimal()

help(mystr.isdecimal)
```



# The len() function

- The len() function is not a string specific function.
- It'll return the length of any Python object that contains **any** countable thing.

```
len(mystr) → 6
```

- In the case of strings it is the number of characters in the string.

# String operators

- Try using the + and += operators with strings in the Python console.
- + concatenates strings.
- += appends strings.
  - These are defined in the string class as functions that operate on strings.
- Index strings using square brackets, starting at 0.

```
a="Hello BU!"  
print(a[4])
```

# String operators

- Changing elements of a string by an index is **not allowed**:

```
In [79]: a='Hello BU!'

In [80]: a[4] = '0'
Traceback (most recent call last):

  File "<ipython-input-80-7c5733c2cb67>", line 1, in <module>
    a[4] = '0'

TypeError: 'str' object does not support item assignment
```

- Python strings are **immutable**, i.e. they can't be changed.

# Old School String Substitutions

- Python provides an easy way to stick variable values into strings called *substitutions*

- Syntax for one variable:

`'string with a %s' % variable`

%s means sub in value

variable name comes after a %

Variables are listed in the substitution order inside ()

- For more than one:

`'x: %s y: %s z: %s' % (xval,yval,zval)`

- Printing:

`print('x: %s, y: %s, z:%s' % (xval,yval,2.0))`

# Recommended: f-string Substitutions

- f-strings are a more contemporary way to format strings.
- Use a lowercase *f* before the first quote.
- Put the names of variables, or function calls, in {} pairs inside the strings.

```
name = 'Boston'  
school = f'{name} University'
```



```
result = f'{mathcalc(1,2,3)}'
```

# While Loops

- While loops have a condition and a code block.
  - the indentation indicates what's in the while loop.
  - The loop runs until the condition is false.
- The *break* keyword will stop a while loop running.
- In the Spyder edit enter in some loops like these. Save and run them one at a time. What happens with the 1<sup>st</sup> loop?

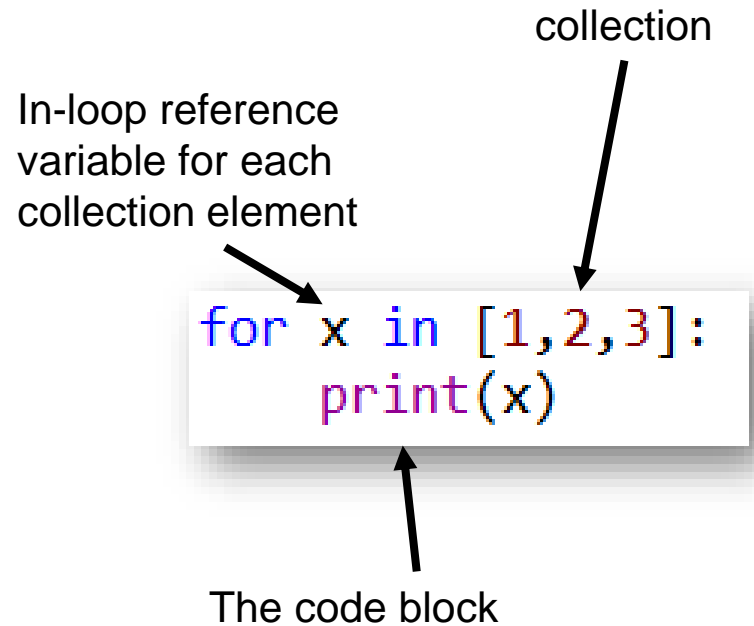
```
while True:
    print("looping!")

a=10
while a > 0:
    print(a)
    a -= 1

my_list=['a','b','c','d','e']
i=0
while i < len(my_list):
    print( my_list[i] )
    i += 1
    if i==3:
        break
```

# For loops (again)

- *for* loops in general loop through a collection of things.
- The *for* loop syntax has a collection and a code block.
  - Each element in the collection is accessed in order by a reference variable
  - Each element can be used in the code block.
- The *break* keyword can be used in *for* loops too.



# Processing lists element-by-element

- A for loop is a convenient way to process every element in a list.
- There are several ways:
  - Loop over the list elements
  - Loop over a list of index values and access the list by index
  - Do both at the same time
  - Use a shorthand syntax called a *list comprehension*
- Open the file *looping\_lists.py*



# Lists With Loops

- Open the file *read\_a\_file.py*
- This is an example of reading a file into a list. The file is shown to the right, *numbers.txt*
- We want to read the lines in the file into a list of strings (1 string for each line), then extract separate lists of the odd and even numbers.

numbers.txt

```
38, 83, 37, 21, 98  
50, 53, 55, 37, 97  
39, 7, 81, 87, 82  
18, 83, 66, 82, 47  
56, 64, 9, 39, 83  
...etc...
```

- *read\_a\_file\_low\_mem.py* is a modification that uses less memory by processing the file line-by-line.

# Tuples

- Tuples are lists whose elements can't be changed.
  - Like strings they are immutable
- Indexing (including slice notation) is the same as with lists.

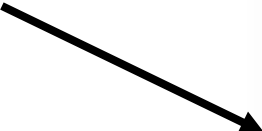
```
# a tuple
a = 10,20,30
# a tuple with optional parentheses
b = (10,20,30)
# a list
c = [10,20,30]
# ...turned into a tuple
d = tuple(c)

# and a tuple turned into a list
e = list(d)
```

# Return multiple values from a function

- Tuples are used to return multiple values from a function.
- Python syntax can automatically unpack a tuple return value.

```
def min_max(x):  
    ''' Return the maximum and minimum  
        values of x '''  
    minval = min(x)  
    maxval = max(x)  
    # a tuple return...  
    return minval,maxval  
  
a = [10,4,-2,32.1,11]  
  
val = min_max(a)  
min_a = val[0]  
max_a = val[1]  
  
# Or, easier...  
min_a, max_a = min_max(a)
```



# Dictionaries

- Dictionaries are another basic Python data type that are tremendously useful.

- Create a dictionary with a pair of curly braces:

```
x = {}
```

- Dictionaries store *values* and are indexed with *keys*

- Create a dictionary with some initial values:

```
x = {'a_key':55, 100:'a_value', 4.1:[5,6,7]}
```

# Dictionaries

- Values can be any Python thing
- Keys can be primitive types (numbers), strings, tuples, and some custom data types
  - Basically, any data type that is **immutable**
- Lists and dictionaries cannot be keys but they can be stored as values.
- Index dictionaries via keys:

```
x['a_key'] → 55  
x[100] → 'a_value'
```

# Try Out Dictionaries

- Create a dictionary in the Python console or Spyder editor.
- Add some values to it just by using a new key as an index. Can you overwrite a value?
- Try `x.keys()` and `x.values()`
- Try: `del x[valid_key]` → deletes a key/value pair from the dictionary.

```
x = {}  
x[3] = -3.3  
x[10.2] = []  
  
print(x)
```

# Modules

- Python modules, aka *libraries* or *packages*, add functionality to the core Python language.
- The [Python Standard Library](#) provides a very wide assortment of functions and data structures.
  - Check out their [Brief Tour](#) for a quick intro.
- Distributions like Anaconda provides dozens or hundreds more
- You can write your own libraries or install your own.

# PyPI

- The [Python Package Index](#) is a central repository for Python software.
  - Mostly but not always written in Python.
- A tool, *pip*, can be used to install packages from it [into your Python setup](#).
  - Anaconda provides a similar tool called *conda*
- Number of projects (as of January 2023): **430,524**
- You should always do your due diligence when using software from a place like PyPI. Make sure it does what you think it's doing!



# Python Modules on the SCC

- Python modules should not be confused with the SCC *module* command.
- For the SCC there are [instructions](#) on how to install Python software for your account or project.
- Many SCC modules provide Python packages as well.
  - Example: tensorflow, pycuda, others.
- Need help on the SCC? Send us an email: [help@scc.bu.edu](mailto:help@scc.bu.edu)

# Importing Libraries

- The *import* command is used to load a library.
- The name of the library is prepended to function names and data structures in the module.
  - The preserves the library *namespace*
- This allows different libraries to have the same function names – when loaded the library name keeps them separate.

```
import math  
  
z=math.sin(0.1)  
  
print(z)  
  
dir(math)  
  
help(math.ceil)
```

Try these out!

# Fun with *import*

- The *import* command can strip away the module name:

```
from math import *
```

- Or it can import select functions:

```
from math import cos  
from math import cos, sqrt
```

- Or rename on the import:

```
from math import sin as pySin
```

# Easter Eggs

```
# Try to load curly braces for Python  
from __future__ import braces
```

```
# Proof that Python programmers have more fun  
import antigravity
```

# Fun with *import*

- The *import* command can also load your **own** Python files.
- The Python file to the right can be used in another Python script:

```
# Don't use the .py ending
import myfuncs
x = [1,2,3,4]
y = myfuncs.get_odds(x)
```

myfuncs.py

```
def get_odds(lst):
    ''' Gets the odd numbers in a list.

    lst: incoming list of integers
    return: list of odd integers '''
    odds = []
    for elem in lst:
        # Odd if there's a remainder when
        # dividing by 2.
        if elem % 2 != 0:
            odds.append(elem)
    return odds
```

- Splitting your code into multiple files helps with development and organization.

# Import details

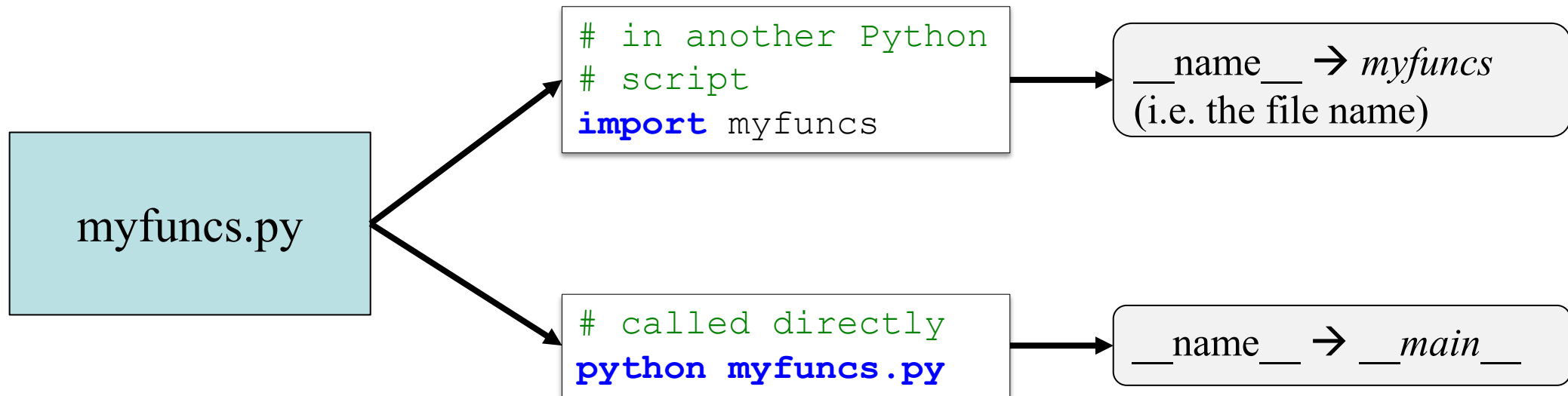
- Python reads and executes a file when the file is:
  - opened directly: `python somefile.py`
  - imported: `import somefile`
- Lines that create variables, call functions, etc. are **all executed**.
- Here these lines will run when it's imported into another script!

myfuncs.py

```
def get_odds(lst):  
    ''' Gets the odd numbers in a list.  
  
    lst: incoming list of integers  
    return: list of odd integers '''  
    odds = []  
    for elem in lst:  
        # Odd if there's a remainder when  
        # dividing by 2.  
        if elem % 2 != 0:  
            odds.append(elem)  
    return odds  
  
x = [1,2,3,4]  
y = get_odds(x)  
print(y)
```

# The `__name__` attribute

- Python stores object information in hidden fields called *attributes*
- Every file has one called `__name__` whose value depends on how the file is used.



# The `__name__` attribute

- `__name__` can be used to make a Python script usable as a standalone program **and** as imported code.
- Now:
  - `python myfuncs.py` → `__name__` has the value of `'__main__'` and the code in the *if* statement is executed.
  - `import myfuncs` → `__name__` is `'myfuncs'` and the *if* statement does not run.

myfuncs.py

```
def get_odds(lst):  
    ''' Gets the odd numbers in a list.  
  
    lst: incoming list of integers  
    return: list of odd integers '''  
    odds = []  
    for elem in lst:  
        # Odd if there's a remainder when  
        # dividing by 2.  
        if elem % 2 != 0:  
            odds.append(elem)  
    return odds  
  
if __name__ == '__main__':  
    x = [1, 2, 3, 4]  
    y = get_odds(x)  
    print(y)
```



# Very Useful Modules

- [numpy](#) is a Python library that provides efficient multidimensional numeric data structures
- [matplotlib](#) is a popular plotting library
  - Remarkably similar to Matlab plotting commands!
- [scipy](#) provides a wide variety of numerical algorithms:
  - Integrations, curve fitting, machine learning, optimization, root finding, etc.
  - Built on top of numpy
- [pandas](#) is used for data analysis using DataFrame structures
  - Very similar to what you find in R.

# numpy

- numpy provides data structures written in compiled C code
- Many of its operations are executed in compiled C or Fortran code, not Python.
- Check out *numpy\_basics.py*

# numpy datatypes

- Unlike Python lists, which are generic containers, numpy arrays are *typed* and hold a single type of data.
- If you don't specify a type, numpy will assign one automatically.
- A [wide variety of numerical types](#) are available.
- Proper assignment of data types can sometimes have a significant effect on memory usage and performance.

```
import numpy as np
x = np.array([1, 2])
# Prints "int64"
print(x.dtype)

x = np.array([1.0, 2.0])
# Prints "float64"
print(x.dtype)

x = np.array([1, 2], dtype=np.uint8)
# Prints "uint8"
print(x.dtype)
```

# Numpy operators

- Numpy arrays will do element-wise arithmetic:  $+$   $-$   $*$   $**$
- Matrix (or vector/matrix, etc.) multiplication needs the `.dot()` function.
- Numpy has its own `sin()`, `cos()`, `log()`, etc. functions that will operate element-by-element on its arrays.

```
import numpy as np
x = np.array([1, 2])

x = x + 1
print(x)

y=x / 2.5

print(y.dtype)
print(y)

print(y * x)
print('Dot product: %s' % y.dot(x))
```

Try these out!

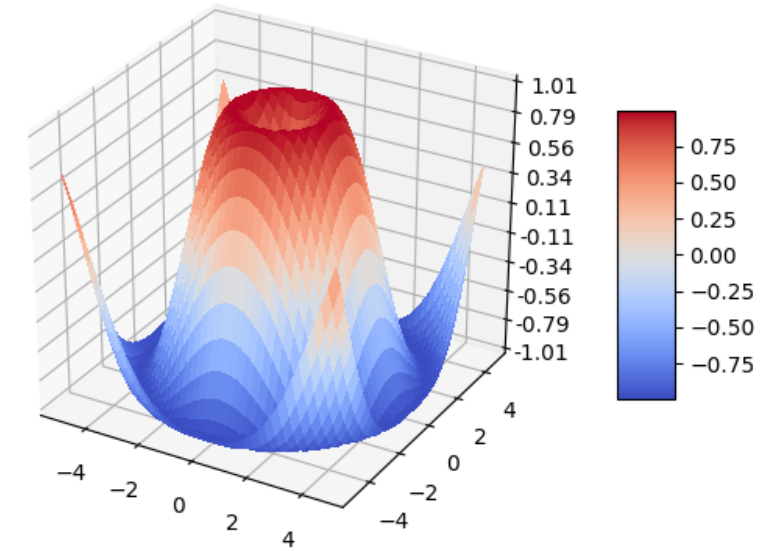
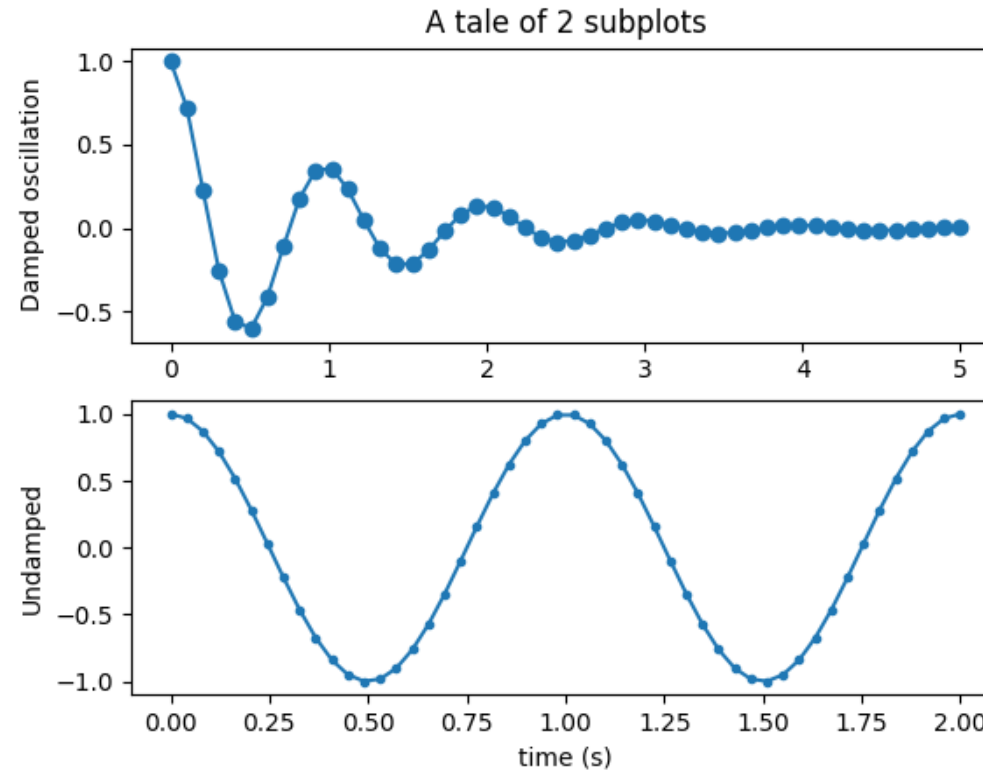
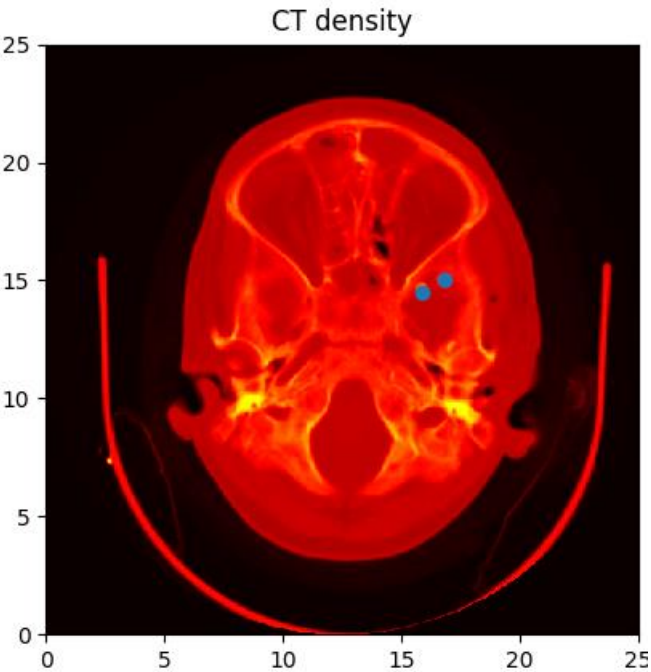
# Plotting with matplotlib

- Matplotlib is the most popular Python plotting library
  - [Seaborn](#) is another.
- Based on Matlab plotting.
- Plots can be made from lists, tuples, numpy arrays, etc.

```
import matplotlib.pyplot as plt
plt.plot([5,6,7,8])
plt.show()

import numpy as np
plt.plot(np.arange(5)+3, np.arange(5) / 10.1)
plt.show()
```

Try these out!



- Some [sample images](https://matplotlib.org) from matplotlib.org
- A vast array of plot types in 2D and 3D are available in this library.

# A numpy and matplotlib example

- *numpy\_matplotlib\_fft.py* is a short example on using numpy and matplotlib together.
- Open *numpy\_matplotlib\_fft.py*
- This sample extracts signals from a noisy background.

# Writing Quality Pythonic Code

- Cultivating good coding habits pays off in many ways:
  - Easier and faster to write
  - Easier and faster to edit, change, and update your code
  - Other people can understand your work
- Python lends itself to readable code
  - It's quite hard to write **completely** obfuscated code in Python.
    - Exploit language features where it makes sense
  - Contrast that with [this sample](#) of obfuscated [C code](#).
- Here we'll go over some suggestions on how to setup a Python script, make it readable, reusable, and testable.



# Compare some Python scripts

- Open up three files and let's look at them.
- A file that does...something...
  - *bad\_code.py*
- Same code, re-organized:
  - *good\_code.py*
- Same code, debugged, with testing code:
  - *good\_code\_testing.py*

# Command line arguments

- Try to avoid hard-coding file paths, problem size ranges, etc. into your program.
- They can be specified at the command line.
- Look at the [argparse module](#), part of the Python Standard Library.

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N                an integer for the accumulator

optional arguments:
  -h, --help      show this help message and exit
  --sum           sum the integers (default: find the max)
```

# Function, class, and variable naming

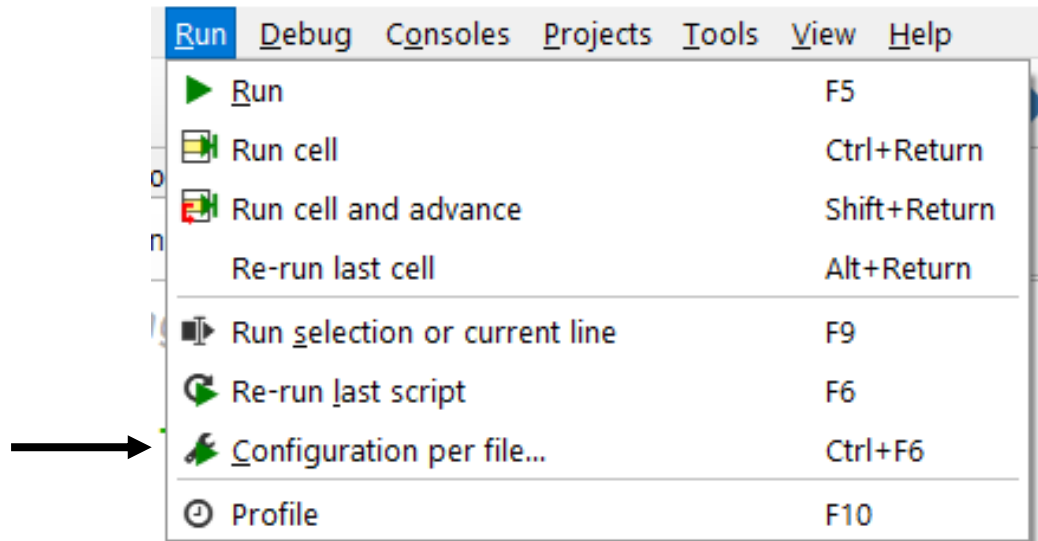
- There's no word or character limit for names.
- It's ok to use descriptive names for things.
- An IDE (like Spyder) will help you fill in longer names so there's no extra typing anyway.
- Give your functions and variables names that reflect their meaning.
  - Once a program is finished it's easy to forget what does what where

# An example development process

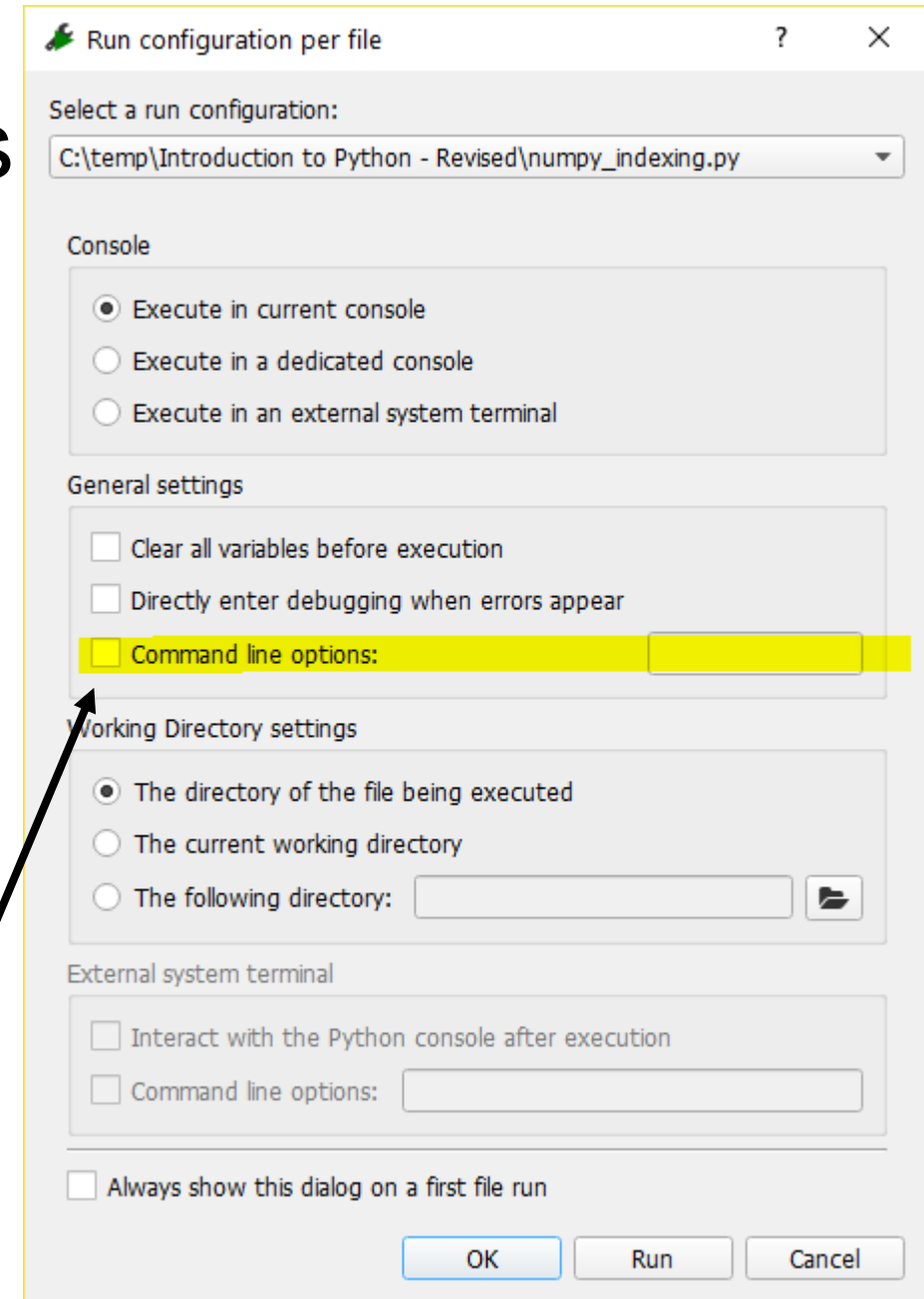
- Work to develop your program.
  - Do some flowcharts, work out algorithms, and so on.
  - Write some Python to try out a few ideas.
  - Get organized.
- Write a “1st draft” version that gets most of what’s needed done.
- Move hard-coded values into the `if __name__ == '__main__':` section of your code.
- Once the code is testing well add command line arguments and remove hard-coded values
- Finally (e.g. to run as an SCC batch job) test run from the command line.

# Spyder command line arguments

- Click on the Run menu and choose *Configuration per file*



- Enter command line arguments



# Python from the command line on the SCC

- To run Python from the command line:

```
[bgregor@scc1 bg]$ module load python3/3.8.10  
[bgregor@scc1 bg]$ python my_script.py -N 30 -L 25 -o outfile.txt
```

- After a Python module is loaded just type *python* followed by the script name followed by script arguments.

# Where to get help...

- The official [Python Tutorial](#)
- [Automate the Boring Stuff with Python](#)
  - Focuses more on doing useful things with Python, not focused on scientific computing
- [Full Speed Python](#) tutorial
- Contact Research Computing: [help@scc.bu.edu](mailto:help@scc.bu.edu)

# End-of-course Evaluation Form

- Please visit this page and fill in the evaluation form for this course.
- Your feedback is highly valuable to the RCS team for the improvement and development of tutorials.

[http://scv.bu.edu/survey/tutorial\\_evaluation.html](http://scv.bu.edu/survey/tutorial_evaluation.html)