Introduction to Beginner-Level Python

Leonardo Bernasconi
Center for Research Computing
University of Pittsburgh

2 February 2023

PittResearch
Center for Research Computing

CRC Foundational Python Track

Part 1: Introduction to Beginner-Level Python (2/22/2023)

Part 2: Introduction to Intermediate-Level Python (2/16/2023)

Part 3: Introduction to Data Manipulation and Visualization (3/2/2023)

Industry-sponsored AI/ML Workshops

More details to come in February.

https://crc.pitt.edu/training/crc-workshops-spring-2023

PittResearch

Center for Research Computing

Purpose of this Workshop

- Learn hot to use Python for automating simple repetitive tasks
- Basic ideas on how to create and run programs in Python
- Understand how to structure a code to make it reusable and readable
- Learn how to install packages to extend Python’s capabilities

Purpose of this Workshop

- Learn hot to use Python for automating simple repetitive tasks
- Basic ideas on how to create and run programs in Python
- Understand how to structure a code to make it reusable and readable
- Learn how to install packages to extend Python's capabilities

About me

PhD in Physical and Theoretical Chemistry (Oxford, UK, 2001)
Postdoc in Theoretical Chemistry (Cambridge, UK, 2001-2004)
Postdoc in Theoretical Chemistry (Amsterdam, The Netherlands, 2004-2008)
Principal Scientist (STFC Rutherford-Appleton Lab, UK, 2008-218)
Research Assistant Professor in Chemistry and Consultant at CRC (2018-)

$$\hat{H}(t)\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N;t) = i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N;t)$$

PittResearch

Center for Research Computing

Overview

1. Introduction: What is Python
2. How to run Python
3. Python syntax
4. Examples
5. Virtual environments
6. Introduction to NumPy/Matplotlib

1
Introduction

Center for Research Computing

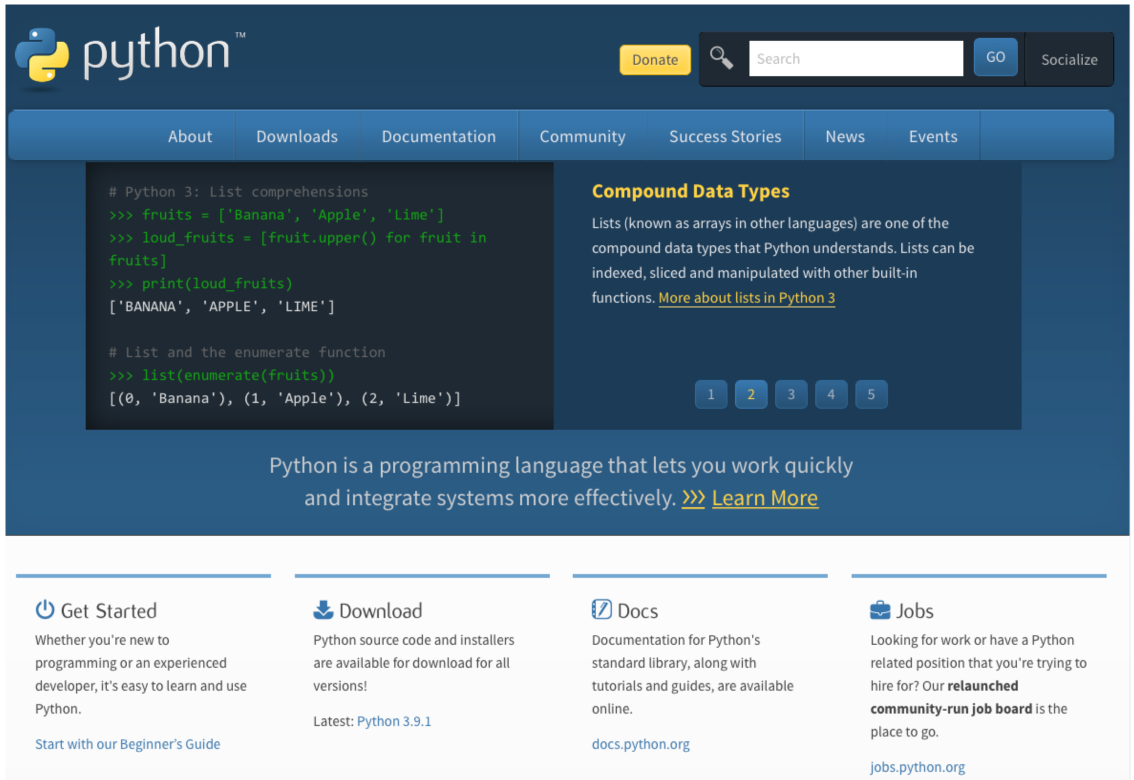
python™

- A general-purpose scripting and programming language
- It is a high-level language: it looks more like English than machine language
- It is an interpreted language: the interpreter converts it line-by-line into ML

- The structure of Python helps programmers write clear and readable code
- It can be useful for small scripts as well as for large software projects

- A relatively young language: first release by Guido van Rossum in 1991, followed by Python 2 (2000) and Python 3 (2008)
- Widely used in industry and academia
- One of the main strength of Python is the existence of a huge standard library: over 287,000 packages for science, machine learning, data analytics, *etc*.

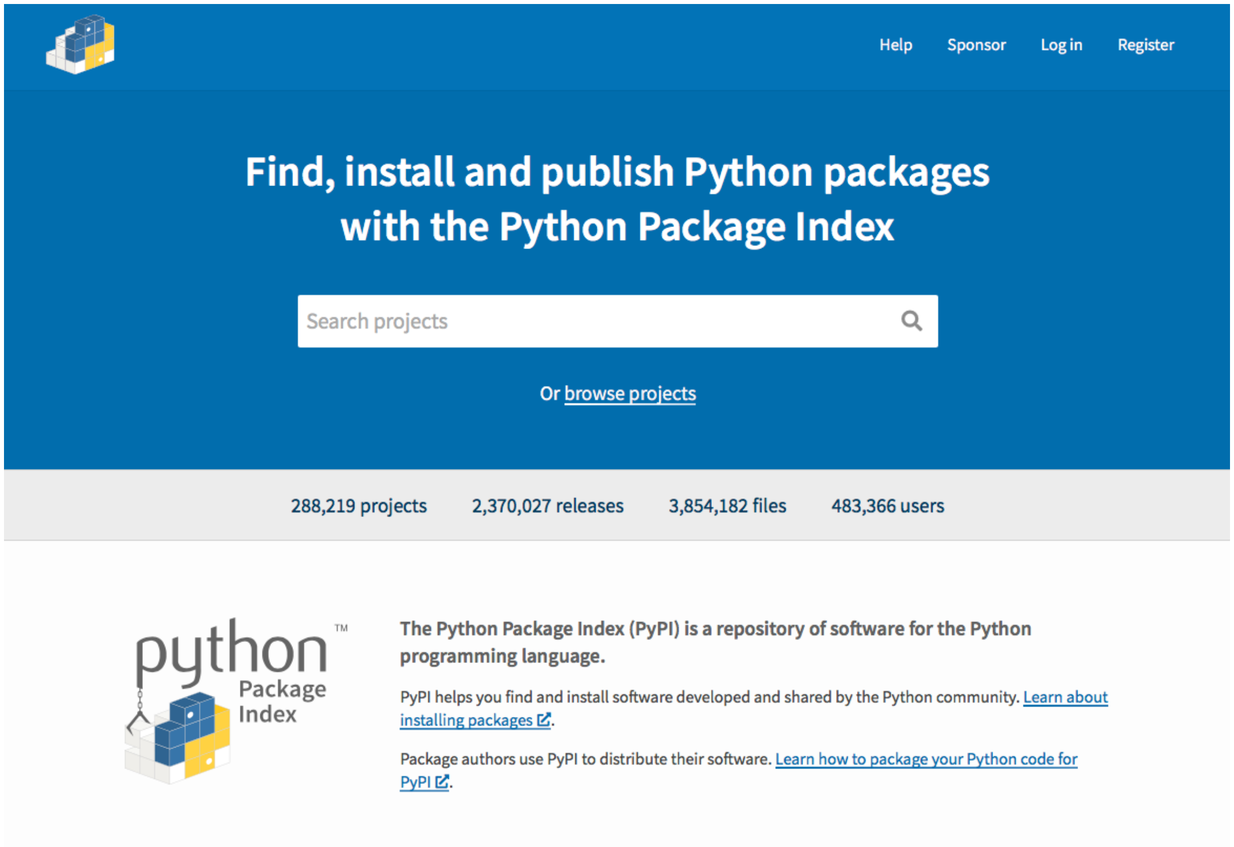
- Python is free and open source
- It is maintained and distributed by the Python Software Foundation
- It is available on most OSs



https://www.python.org

PittResearch
Center for Research Computing

- Python packages are distributed by their developers
- They are typically very easy to install



https://pypi.org

PittResearch
Center for Research Computing

Main strength of Python

The ability to write clear and well-structured code, with no need to worry about low level operations (*e.g.*, memory management)

Main disadvantage

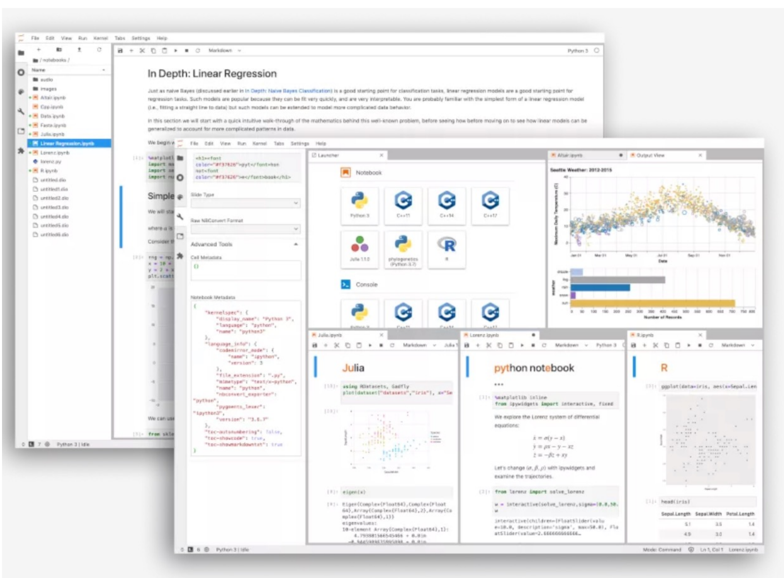
Python code is slow compared to compiled languages (https://julialang.org/benchmarks/)

Often the best solution is to write computationally intensive parts of a code in a compiled language and use Python wrappers to orchestrate these low-level, but very efficient, parts of the code.

2
How to run Python

How to run Python

1) Through an interactive session
2) Executing a script/program
3) Using Jupyter notebooks (https://jupyter.org)
4) Using Google Colab (https://colab.research.google.com)
5) Using an integrated development environment (IDE), *e.g.,* PyCharm (https://www.jetbrains.com/pycharm/)



University of Pittsburgh | Center For Research Computing

Jupyter notebooks on the CRC cluster through Jupyter Hub and Open Ondemand

https://crc.pitt.edu/Access-CRC-Web-Portals

PittResearch

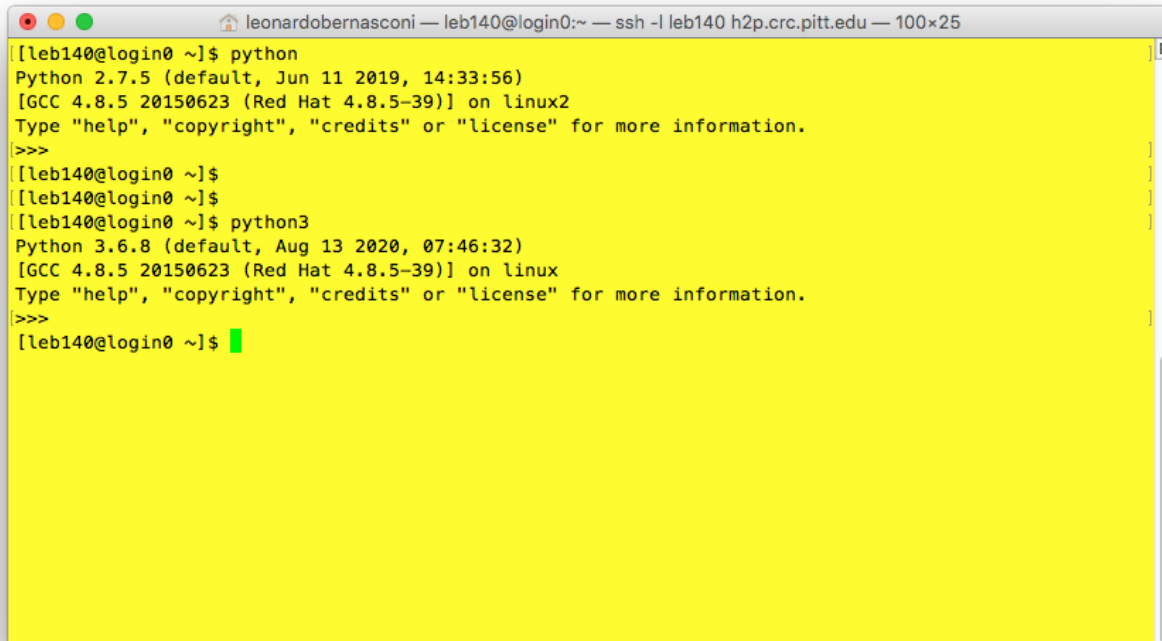
Center for Research Computing

Interactive sessions

1) Start Python: python (for Python2) or python3
2) Type commands line by line
3) Exit using: Ctrl + D

or:

exit()



PittResearch

Center for Research Computing

3
Python syntax

PittResearch

Center for Research Computing

Data types

Numbers

`12, 299792458, 0.001, 3+5j`

Python as a calculator

Variable assignment (*e.g.,* `c = 299792458`)

Operators

`+, -, *, /, %, //, **`

==, !=, <, >, >=, <=

Logical variables (True and False)

The `math` module:

```
import math
dir(math)
```

Built-in modules: `help('modules')`

PittResearch

Center for Research Computing

Data types

Lists

`l = [1, 2, 13.3, “today”, 6+5j]`

List index (always integer; can be negative)
Length of a list: `len(l)`

Sublists: note **slicing** is from an index to a given element position

List manipulation:
`insert(pos, element), append(), remove(), pop(), extend()`
`list1 + list2`

Membership operators: `in / not in`

Nested lists

Data types

Strings

```
string1 = “today”
string2 = ‘tomorrow’
string3 = ‘”yesterday”’
```

String indices
Substrings, slicing

Concatenation: `string1 + string2 + string3`
Repetition: `string1 * 3`

Membership operators: `in / not in`

Data types

Tuples

Similar to strings, but their elements are **immutable**

```
t1 = (1, 2, 3)
```

Tuple indices

Substrings, slicing

Nested tuples and their indices
Membership operators: `in / not in`

Data types

Dictionaries

```
d1 = {}
d1[1] = 1; d1[2] = 4; etc.
```

Keys: `d1.keys()`
Values: `d1.values()`
Clear: `d1.clear()`

Nested dictionaries

Data types

Files

Read from file and write to file

Read from file:

```
input_file = open('input.file', 'r')
input_file.read()
input_file.close()
```

Write to file:

```
output_file = open('output.file', 'w')
output_file.write()
output_file.close()
```

We can read/write a file as a single string or as a sequence of lines

Control statements and loops

Conditional

```
if condition1:
    (execute some instructions)
elif condition2:
    (execute some other instructions)
elif condition3:
    (execute some other instructions)
else:
    (execute some other instructions)
```

Indentation (four blank spaces) is very important in Python!!!

Switch to running scripts.

`/ihome/sam/leb140/IntroToPython/example1.py`

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 90×26

```
a = 15

if a == 10:
    print(a)
    print("... It is Monday")
elif a == 15:
    print(a)
    print("... It is Tuesday")
else:
    print("... I do not know what day it is")
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"example1.py" 10L, 174C
```

PittResearch

Center for Research Computing

Control statements and loops

for loop

`for` variable `in` sequence`:`
    (execute some instructions)

The function `range():`

```
range(n)
range(start, stop)
range(start, stop, step)
```

Nested loops

Loops with `if/else` blocks:

`for` variable `in` sequence`:`
    `if` Condition:
        (execute some instructions)
    `else:`
        (execute some other instructions)

Loops and conditionals: example2.py

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 90×26

```
#mylist = [1, 2, 3]
#
#for element in mylist:
#    print(element)
#
# I am going to ignore the lines above

#for i in range(0, 50, 2):
#    print(i)

mylist = [1, 2, 3, 4, 5, 6]

for element in mylist:
    if element % 2 == 0:
        print(element)
        print("Even number")
    else:
        print(element)
        print("Odd number")
~
~
~
~
~
~
"example2.py" 19L, 340C
```

Control statements and loops

Reading files line-by-line

```
open_file = open("some_file", "r")

for line in open_file:
        (execute some instructions on the line)

open_file.close()
```

Example: read a file with multiple values per line and store the values in lists

The `strip()` and `split()` methods

example3.py

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 90×26

```
myfile = open("file.txt", "r")

for line in myfile:
    mylist = line.strip("\n").split(",")
    print(int(mylist[0]) + int(mylist[1]))
    #print(line.strip("\n"))
    print(mylist)

myfile.close()
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"example3.py" 9L, 199C
```

Control statements and loops

while loop

```
while condition:
      (execute some instructions)
```

Nested loops

Loops with else blocks:

```
while condition:
      (execute some instructions)
else:
      (execute some other instructions)
```

example4.py

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 90×26

```
i = 0

while i <= 10:
    if i <= 5:
        print(i)
    else:
        print("i is larger than 5")
    i += 1
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"example4.py" 8L, 111C
```

PittResearch

Center for Research Computing

Control statements and loops

The `break` statement

It is used to terminate a for/while loop when a given condition is met

```
for variable in sequence:
    (execute some instructions)
    if condition:
        break                              <- Will exit the loop
    (execute some other instructions)
```

The `continue` statement

It is used to skip instructions within a for/while loop

```
for variable in sequence:
    (execute some instructions)
    if condition:
        continue
    (execute some other instructions)      <- Will be skipped, but will not exit the loop
```

Control statements and loops

The `pass` statement

It tells the Python interpreter to *do nothing*. It works as a placeholder.

```
for variable in sequence:
    (execute some instructions)
    if condition:
        pass
    else:
        (do something else)
```

Functions

Functions are blocks of code that carry out specific tasks. They are useful if a given set of operations must be repeated more than once in a code.

They give the code **re-usability**, *i.e.,* the ability to use a given set of instructions at different stages of the computation without having to modify the code.

They help with code **readability**, especially if they are well documented. All the instructions required by a given task are grouped together.

They also avoid **redundancy**, helping with code maintainability and greatly improving extendibility.

Functions (and their equivalents in other programming languages) are essential ingredients in good programming practice.

Functions

```
def function_name(function_arguments):
    (do something)
    return
```

(`return` is optional)

Default arguments can be used to avoid errors when calling a function

```
def function_name(arg1, arg2=something):
 (do something)
    return
```

Functions always appear before the main code.

User defined functions and *built-in* functions

See function1.py

PittResearch
Center for Research Computing

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 90×26

```
# A function that takes two numbers as input, squares the first number and adds
# the second number and returns the result.

def myfunction(a_number, another_number):

    """This function does what I wrote above."""

    return a_number * a_number + another_number


def anotherfunction(a_number):

    """This function computed the square of a_number."""

    return a_number * a_number

# Main code
for a in range(10):
    b = a + 4
    print(myfunction(a, b))
    print(anotherfunction(a))
~
~
~
~
"function1.py" 21L, 498C
```

Invoking external commands in Python

List files using ls command:

```
from subprocess import call
call('ls')
```

Return date using the Unix 'date' command:

```
import subprocess
time = subprocess.check_output('date')
print("It is", time)
```

PEP8: Style Guide for Python code

Guidelines that improve the readability and consistency of Python code

https://peps.python.org/pep-0008/

Python syntax checkers can be installed, which parse Python code and report any PEP8 violations, *e.g.,* pip8 and pycodestyle.

They can be installed in a virtual environment (see below) using

```
python3 –m pip install pep8
```

or

```
python3 –m pip install pycodestyle
```

PittResearch

Center for Research Computing

4
Examples

Exercise 1

Write a function that returns all *prime numbers* up to a given maximum.

A prime number is an integer greater than 1 that cannot be written as the product of any lower natural number: 2 is prime, 3 is prime, 4=2*2 is not prime, *etc.*

Questions:

1) What should the input parameter(s) of the function be?
2) How do we use loops to find out if a given number is the product of two lower numbers?
3) What should the function return?

leonardobernasconi — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 113×31

```
def primes(maxnumber):

    """This function returns a list of prime numbers within the range (2, maxnumber).

    Input:
          maxnumber = maximum number in the range to consider;
    Output:
          A list of prime numbers up to maxnumber."""

    # Define the list of prime numbers
    prime_numbers = []

    # Loop over all integers from 2 to maxnumber
    for i in range(2, maxnumber+1):

        # I assume that i is prime
        i_is_prime = True

        # Loop over integers lower than i
        for j in range(2, i):

            if i%j == 0:
               i_is_prime = False
               break

        if i_is_prime:
            prime_numbers.append(i)

    return(prime_numbers)
```

1,12 Top

prime_numbers.py

PittResearch

Center for Research Computing

Exercise 2

Write a code (containing at least one function) that computes the difference between a series of numbers read from two different files (number from file1 minus number from file 2) and saves these differences to an output file file3.

Note: each of the two input files contains one number per line, but the two files need not have the same number of lines. We will only compute differences for numbers that can be read from both files.

Questions:

1) How many files do we need to open at a given time?
2) How do we deal with the fact that the number of lines in the two input files can be different?

leonardobernasconi — leb140@login0:~/Python — ssh -Y -l leb140 h2p.crc.pitt.edu — 122×58

```
def subtract(a, b):

    """This function computes an element-by-element difference between the two lists
       a and b and returns is as a list c."""

    # Initialize return list c (an empty list)
    c = []

    # Find the number of elements for which the difference can be computed:
    # We use the intrinsic function min
    maxel = min(len(a), len(b))

    # Index for elements of a and b
    index = 0

    # Loop on the elements of a
    while index < maxel:
        c.append(a[index]-b[index])
        index += 1

    return c

# Main program

# Read lines of file1 and store them in list aread
finput = open("file1", 'r')
aread = finput.readlines()
finput.close()

# Read lines of file2 and store them in list bread
finput = open("file2", 'r')
bread = finput.readlines()
finput.close()

# Convert aread into a list of integers (a)
a = []
for item in aread:
    a.append(int(item))
# Convert aread into a list of integers (a)
b = []
for item in bread:
    b.append(int(item))

# Compute the element-by-element difference between a and b
aminb = subtract(a, b)

# Convert aminb into a list of strings (aminbs)
aminbs = []
for item in aminb:
    aminbs.append(str(item) + "\n") # We need to add "\n" to indicate new lines

# Print aminb to a file file3
fout = open("file3", 'w')
fout.writelines(aminbs)

~
```

56,0-1 All

Possible solution to Exercise 2.

Can we improve this code?

PittResearch

Center for Research Computing

```
def subtract(a, b):

    """This function computes an element-by-element difference between the two lists
       a and b and returns is as a list c."""

    # Initialize return list c (an empty list)
    c = []

    # Find the number of elements for which the difference can be computed:
    # We use the intrinsic function min
    maxel = min(len(a), len(b))

    # Index for elements of a and b
    index = 0

    # Loop on the elements of a
    while index < maxel:
        c.append(a[index]-b[index])
        index += 1

    return c

# Main program

# Read lines of file1 and store them in list aread
finput = open("file1", 'r')
aread = finput.readlines()
finput.close()

# Read lines of file2 and store them in list bread
finput = open("file2", 'r')
bread = finput.readlines()
finput.close()

# Convert aread into a list of integers (a)
a = []
for item in aread:
    a.append(int(item))
# Convert aread into a list of integers (a)
b = []
for item in bread:
    b.append(int(item))

# Compute the element-by-element difference between a and b
aminb = subtract(a, b)

# Convert aminb into a list of strings (aminbs)
aminbs = []
for item in aminb:
    aminbs.append(str(item) + "\n") # We need to add "\n" to indicate new lines

# Print aminb to a file file3
fout = open("file3", 'w')
fout.writelines(aminbs)

~
                                                                  56,0-1        All
```

Unnecessary code duplication

Possible solution to Exercise 2.

Can we improve this code?

Exception handling

leonardobernasconi — leb140@login0:~/Python — ssh -Y -l leb140 h2p.crc.pitt.edu — 120×52

```
def subtract(a, b):

    """This function computes an element-by-element difference between the two lists
       a and b and returns is as a list c."""

    # Initialize return list c (an empty list)
    c = []

    # Find the number of elements for which the difference can be computed:
    # We use the intrinsic function min
    maxel = min(len(a), len(b))

    # Index for elements of a and b
    index = 0

    # Loop on the elements of a
    while index < maxel:
        c.append(a[index]-b[index])
        index += 1

    return c

~
~
~
s0.py                                                                          1,1           All
```

```
def subtract(a, b):

    """This function computes an element-by-element difference between the two lists
       a and b and returns is as a list c."""

    # Initialize return list c (an empty list)
    c = []

    # Index for elements of a and b
    index = 0

    # Loop on the elements of a
    for elementa in a:

        # Exception handling
        try:
            c.append(elementa-b[index])
            index += 1
        except:
            break

    return c

~
s1.py                                                                          1,1           All
```

Exception handling

5
Virtual environments

Virtual environments

A virtual environment is a complete Python installation which is isolated from the system Python and from other virtual environments.

The Python interpreter, scripts, libraries and packages installed in the virtual environment are independent and may differ from the system Python.

Virtual environments are useful for maintaining specific sets of packages or different versions of the same package.

They are very useful when we work on HPC systems, like the CRC cluster, which do not allow users to modify the system Python. With virtual environments we have complete control on package installation, uninstallation, *etc*.

Official man page: https://docs.python.org/3/library/venv.html

Virtual environments

The command `venv` is used to **create** a new virtual environment:

```
python3 -m venv myenv
```

This will create a directory `myenv` containing the new Python installation.

We now need to **activate** the environment:

```
source myenv/bin/activate
```

We can ”exit” the virtual environment and return to the system Python using:

```
deactivate
```

(For Windows, see https://docs.python.org/3/library/venv.html or https://realpython.com/python-virtual-environments-a-primer/.)

Virtual environments: install Python packages

After activating a virtual environment, we will be using the specific version of Python built in the environment.

To install new packages, use:

```
python3 -m pip install <package_name>
```

If a given virtual environment is no longer needed, we can delete it simply by removing its directory:

```
rm -rf myenv/
```

Example: install numpy in a virtual environment myenv

Create and activate the virtual environment:

```
python3 -m venv myenv
source myenv/bin/activate
```

Install numpy:

```
python3 -m pip install numpy
```

Now launch the python interpreter:

```
python3
```

and check if the new package has been installed:

```
import numpy
```

To list all installed packages: `python3 -m pip list`

PittResearch

Center for Research Computing

Virtual environments: Anaconda (https://anaconda.org)

Create a conda environment:

```
conda create -n yourenvname python=x.x anaconda
```

Activate the virtual environment:

```
source activate yourenvname
```

Install packages:

```
conda install -n yourenvname [package]
```

Deactivate the environment:

```
source deactivate
```

https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/20/conda/

Using virtual environments with CRC JupyterHub

As an example, we will create a virtual environment called *myenv* to be used with Jypyter Hub in notebooks.

In a terminal (either on h2p or on Jupyter Hub) use the following commands:

```
module purge
module load python/3.7.0
python3 -m venv myenv
source myenv/bin/activate
python3 -m pip install ipykernel
python3 -m ipykernel install --user --name=myenv
```

On Jupyter Hub open a new notebook and select *myenv* from the notebook kernels available. To check that the version of Python running is the one from the virtual environment, and not the system Python, use:

[ ]:
```
import sys
print(sys.executable)
```

which should return something like

```
[...]/.virtualenvs/myenv/bin/python
```

https://crc.pitt.edu/user-support/installed-software/python

PittResearch
Center for Research Computing

Python on the CRC cluster

H2P access: https://crc.pitt.edu/user-support/cluster-access

To see the versions of python installed: `module spider python`

To use a specific version of Python: `module load python/3.7.0`

leo — leb140@login0:~/IntroToPython — ssh -Y -l leb140 h2p.crc.pitt.edu — 111×42

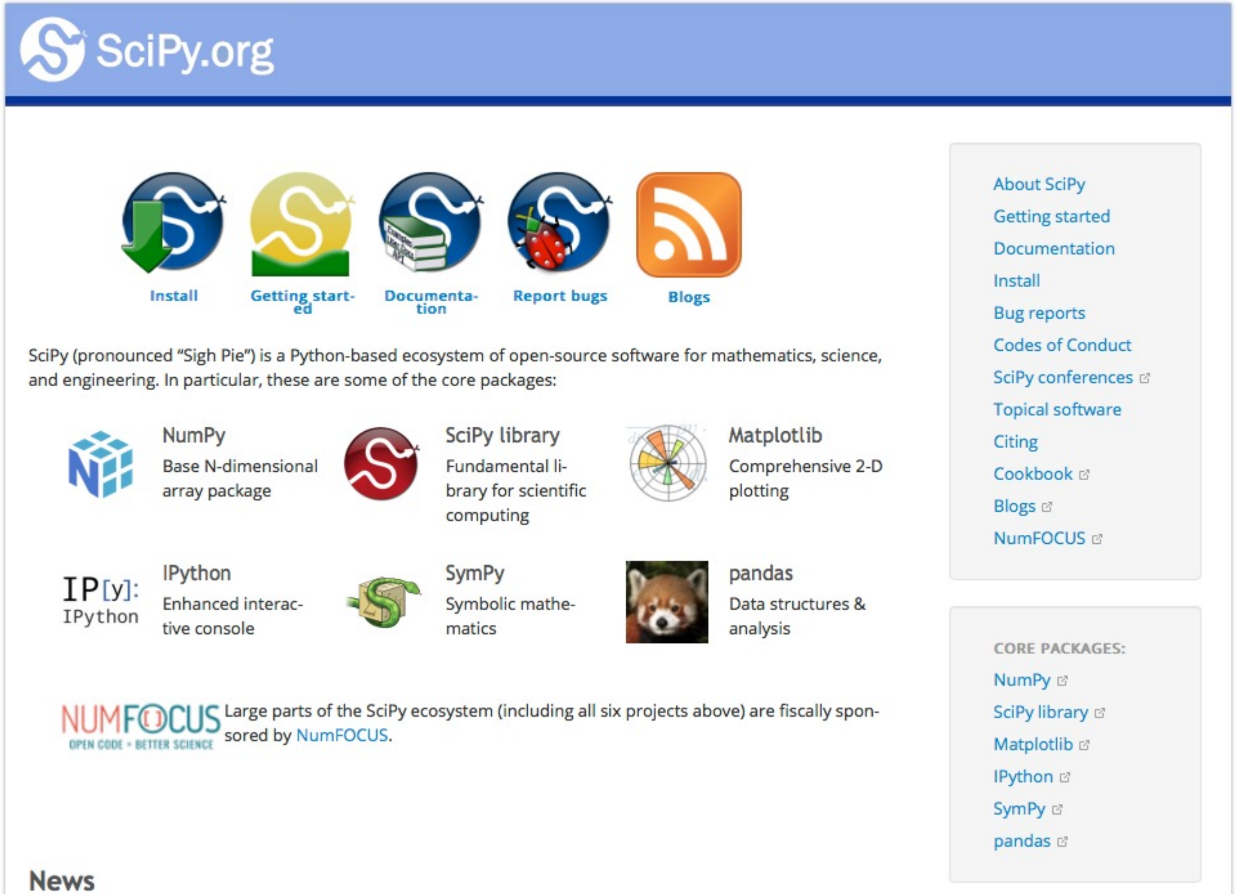
```
-------------------------------------------------------------------------------------
  python:
-------------------------------------------------------------------------------------
    Description:
      Anaconda is the leading open data science platform powered by Python.

     Versions:
        python/anaconda2.7-4.2.0_westpa
        python/anaconda2.7-4.2.0
        python/anaconda2.7-4.4.0_genomics
        python/anaconda2.7-5.2.0_westpa
        python/anaconda2.7-5.2.0
        python/anaconda2.7-2018.12_westpa
        python/anaconda3.5-4.2.0-dev
        python/anaconda3.5-4.2.0
        python/anaconda3.6-5.2.0_deeplabcut
        python/anaconda3.6-5.2.0_leap
        python/anaconda3.6-5.2.0
        python/anaconda3.7-5.3.1_genomics
        python/anaconda3.7-2018.12_westpa
        python/anaconda3.7-2019.03_astro_bagpipes-0.8.2
        python/anaconda3.7-2019.03_astro_bagpipes-0.8.8
        python/anaconda3.7-2019.03_astro
        python/anaconda3.7-2019.03_deformetrica
        python/anaconda3.7-2019.03
        python/anaconda3.8-2020.11
        python/anaconda3.9-2021.11
        python/bioconda-2.7-5.2.0
        python/bioconda-3.6-5.2.0
        python/bioconda-3.7-2019.03
        python/intel-3.5
        python/intel-3.6_2018.3.039
        python/intel-3.6_2019.2.066
        python/intel-3.6
        python/ondemand-jupyter-python3.8
        python/3.7.0-dev
        python/3.7.0-fastx
        python/3.7.0

     Other possible modules matches:
lines 1-41
```

PittResearch

Center for Research Computing

6
NumPy/Matplotlib

Center for Research Computing



https://scipy.org

A few words on NumPy

NumPy is a Python library used for working with arrays. It also functions for working in domain of linear algebra, Fourier transform and matrices.

You can see what NumPy makes available using the dir() function

```
import numpy as np
dir(numpy)
```

NumPy provides an array object that is up to 50x faster than traditional Python lists.

```
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)

arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Arrays can have 1, 2, 3 or more dimensions.

PittResearch
Center for Research Computing

Arrays

Accessing array elements:

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0, 1])
```

Negative indices can be used as in standard Python lists. Slicing also works like in lists:

```
print(arr[1, 1:4])
```

Copy and **view** arrays:

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 0
print(arr); print(x)

arr = np.array([1, 2, 3, 4, 5])
y = arr.view()
y[0] = 0
print(arr); print(y)
```

PittResearch
Center for Research Computing

Shape, reshape and iteration

Shape of an array:

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

Answer: `(2, 4)`

Reshape an array:

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Iterating through array elements:

```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

PittResearch
Center for Research Computing

Join, split and search arrays

Join arrays:

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
```

Split arrays:

```
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)
```

Search arrays:

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)

Answer: (array([3, 5, 6]),)
```

Sort and filter arrays

Sort arrays:

```
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

Answer: `[0 1 2 3]`

It can be used with higher-dimensional arrays and with arrays of strings or booleans.

Filter arrays: use a *boolean index* list to select values from an array:

```
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

Answer: `[41 43]`

PittResearch
Center for Research Computing

Universal functions (ufunc)

In addition to built-in functions, user-defined functions can be defined, which perform faster than standard Python functions on lists and operate on NumPy arrays.

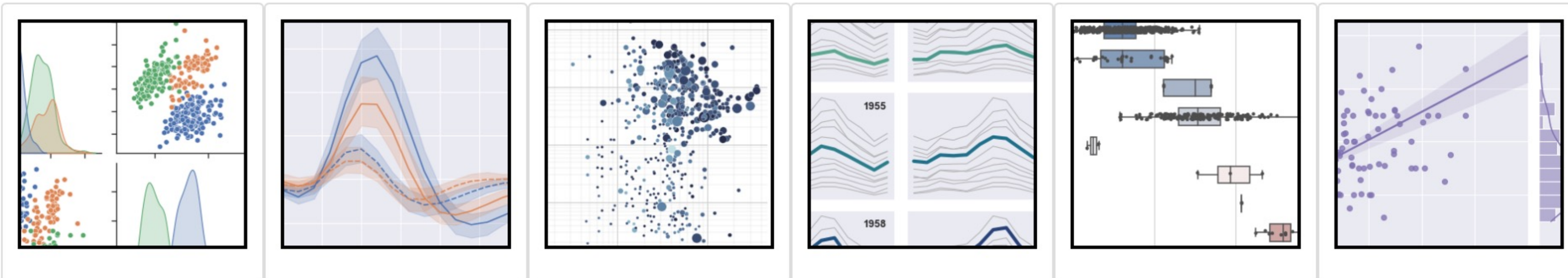
Example:

```
import numpy as np

def myadd(x, y):
  return x+y

myadd = np.frompyfunc(myadd, 2, 1)

print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))
```

`frompyfunc` adds the new function myadd to the NumPy ufunc library. ufunc uses vectorization, which is a faster way to operate on elements of arrays.

More info:
https://www.w3schools.com/python/numpy/numpy_ufunc.asp

Plotting data



We will need to install two additional packages in our virtual environment:

```
python3 -m pip install matplotlib

python3 -m pip install seaborn
```

More info:
https://matplotlib.org
https://seaborn.pydata.org

PittResearch
Center for Research Computing

Example: visualizing a normal distribution

The normal (or Gaussian) distribution represents the distribution of many events around a maximum. In NumPy, we can build this distribution using the `random` module:

```
from numpy import random
```

The method `random.normal` creates the distribution:

```
random.normal(loc, scale, size)
```

`loc`: center of the distribution (mean)
`scale`: width of the distribution (standard deviation)
`size`: shape of the NumPy array containing the distribution

Example: visualizing a normal distribution

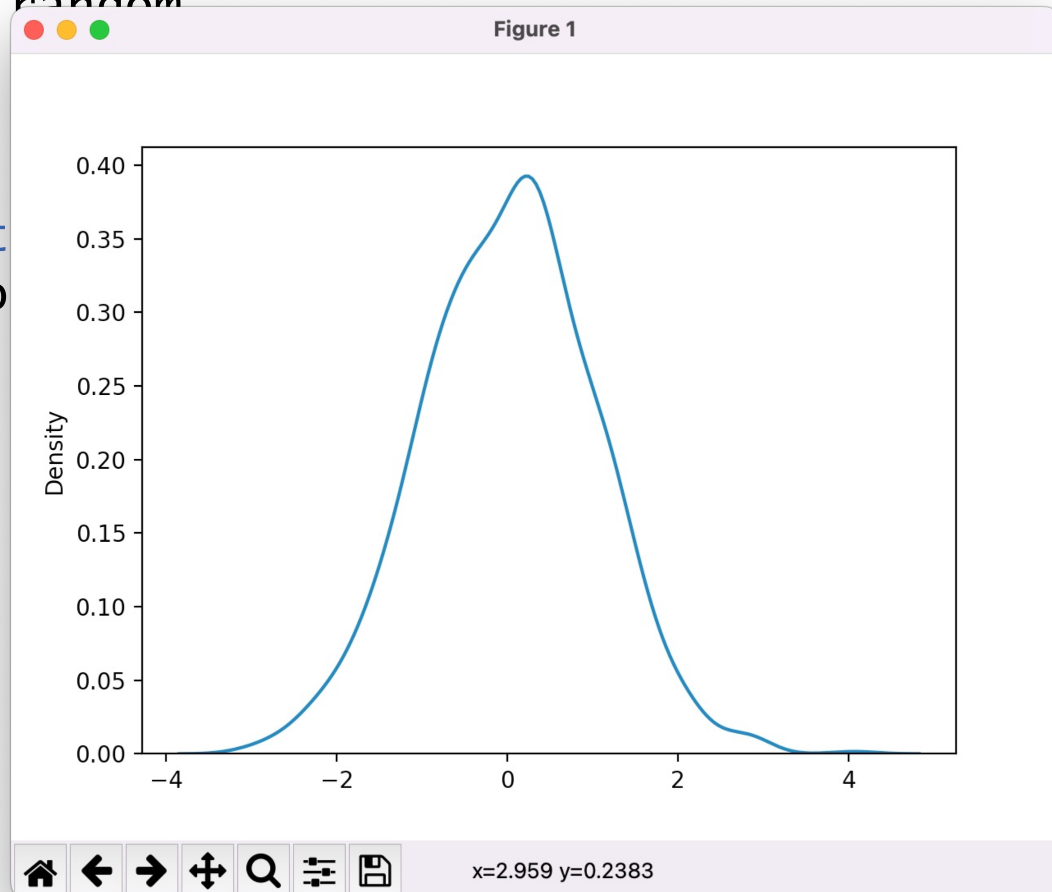
```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns

# Create distribution
sample = random.normal(loc=0.0, scale=1.0, size=1000)

# Plot graph
sns.distplot(sample, hist=False)
plt.show()

# We can also save the plot to a file
plt.savefig(“plot.png”)
```

More info:
https://matplotlib.org
https://seaborn.pydata.org

PittResearch
Center for Research Computing

Example: visualizing a normal distribution

```
from numpy import random
import matplotlib.
import seaborn as

# Create distribut
sample = random.no

# Plot graph
sns.distplot(sampl
plt.show()
```



More info:
https://matplotlib.org
https://seaborn.pydata.org

PittResearch
Center for Research Computing

Example: visualizing a normal distribution

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns

# Make the example reproducible
np.random.seed(0)

# Create distribution
sample = random.normal(loc=0.0, scale=1.0, size=1000)

# Plot graph
sns.distplot(sample, hist=False)
plt.show()
```

More info:
https://matplotlib.org
https://seaborn.pydata.org

PittResearch
Center for Research Computing

Summary

- Python is a powerful all-purpose programming and scripting language
- It has a huge standard library of packages
- It is easy and fun to learn
- It can be used to write wrappers for low-level code
- (It has object-oriented capabilities)

Where to go from here:
- Develop your own software project
- Test Jupyter and Colab notebooks
- Play with virtual environments; test Python packages

Questions and suggestions: leb140@pitt.edu

CRC web site: https://crc.pitt.edu

PittResearch

Center for Research Computing