Python and `numpy`

David A. Clarke

Z02 – Software Development Center

26 April 2021

UNIVERSITÄT BIELEFELD
Fakultät für Physik

CRC-TR 211
Strong-interaction matter under extreme conditions

Foreword

The purpose of this talk is to

- pique the interest of someone unfamiliar with Python;
- include a few examples to get them started; and
- get them acquainted with `numpy` and `matplotlib`.

I assume you can already do some scientific programming.

This talk will be uploaded to Redmine , where you can find other excellent talks by the Software Development Center. Speaking of which, we will try to keep principles of Clean Code in mind.

If you have questions, **interrupt me!**

Outline

1. Why Python?
2. Some Python basics
3. `numpy`, and `matplotlib`
4. `scipy` advertisement
5. Wrap-up

1. Why Python?
2. Some Python basics
3. `numpy`, and `matplotlib`
4. `scipy` advertisement
5. Wrap-up

My memoir

Curiosity and innocence: DOG.BAS

```
10 PRINT "WOOF"
GOTO 10
```

The death of innocence: dog.cpp

```
#include <iostream>
using namespace std;
int main() {
  while(true) {
    cout << "woof..." << endl;
  }
  return 0;
}
```

Resurrection: dog.py

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

Why should I learn Python?

As we saw, it's **quite readable**, which is good for Clean Code.

It's **very popular**: According to `codeburst.io`, Python is the second most popular language for developers, based on surveys from StackOverflow, GitHub activity, and Google searches.

Like any other popular language it has a **large community**.

Plus knowing a popular language probably makes you more desirable, for those of you who may look for careers outside physics.

Why should I learn Python?

It's **very flexible**, especially compared to languages like R.

But most importantly, it's **well-suited for science**:

- Access to libraries useful for scientific computing.
  - `numpy`: element-wise array operations, linear algebra.
  - `scipy`: optimization, curve-fitting.
  - Fancy stuff: Google searches, Excel manipulation, machine learning.
- Object-oriented programming.
- Nice for plotting.
  - Professional looking plots using `matplotlib`.
  - More power and sophistication than e.g. `gnuplot`.

When should I use Python?

When NOT to use it:

- Python is a scripting language, so you don't compile it.
- Hence it's too slow for **computationally intensive work**.
- And it's not good for **memory intensive work**.

When to use it:

- Personally I use it for **almost everything else**.

Example. C++ is the language our group uses to generate configurations. One configuration demands for a small lattice in a simple theory $\mathcal{O}(10\,000) \times \mathcal{O}(10\,000)$ matrix multiplications. But Python is perfect to analyze the measurements.

Getting started

To try this at home, you'll need:

- 🔗Python3 (Python 2.x will not be maintained past 2020)
- 🔗numpy
- 🔗scipy

You might also find the following resources helpful:

- 🔗Python documentation
- 🔗numpy and scipy documentation
- 🔗Jupyter interactive notebooks

To run Python code: `python3 myScript.py`

You can also type just `python3` to play around in real time.

Terminal output

print is the primary command for terminal output. The code

```
print("Hello, world!")
print(99, "Luftballons")
print("Baby you're a...\t firework\n")
print("%11.5f is the loneliest number." % 1.0)
print("%11.5e can be as bad as %11.5e." % (2.0, 1.0))
```

yields

```
Hello, world!
99 Luftballons
Baby you're a...   firework

    1.00000 is the loneliest number.
2.00000e+00 can be as bad as 1.00000e+00.
```

Variables and comments

You don't need to declare types. Python figures out what they are.

```python
one   = 1                          # an integer
two   = 2.0                        # a float
three = two + one                  # cast as float
city  = "bielefeld"
thing = city + "Conspiracy"

""" Can also comment this way """
print(one, two, three, thing)
```

Note `two` is Python's **float** type, which is double precision. If you need another precision, you can use `numpy`.

```
1 2.0 3.0 bielefeldConspiracy
```

Arithmetic

```
x = 1
y = 2
z = 2.0
print(x, y, z, x/y) # x/y gives 0.5 rather than 0

x += 1
y += 2.0            # Recasts y as float
z -= 1
print(x, y, z, y/z, y**2)

y *= 2
z /= 2
print(y, z)
```

```
1 2 2.0 0.5
2 4.0 1.0 4.0 16.0
8.0 0.5
```

Lists

The basic array-like object in Python is called a list. The first element is indexed with 0. Elements don't have to be the same[1] type.

```
emptyList   = []
integerList = [1, 2, 3]
floatList   = [4.0, 5.0, 6.0]
vocabList   = ["Springfield", "embiggen"]

print( len(integerList) )
print( floatList[1] )

vocabList.append("cromulent")

print(vocabList)
```

```
3
5.0
['Springfield', 'embiggen', 'cromulent']
```

[1] You pay for this with a performance loss (in comparison to `numpy` arrays).

Dictionaries

A dictionary is an unordered collection indexed by keys. For each key there is a corresponding value.

```
emptyDictionary = {}
englishToGerman = { "I"      : "ich",
                    "you"    : ["du", "Sie", "ihr"],
                    1        : "eins",
                    2        : "zwei",
                    "police" : "Polizei" }

englishToGerman["she"] = "sie"

print( englishToGerman["I"])
print( englishToGerman["you"][0] )
print( englishToGerman[1] )
```

```
ich
du
eins
```

Conditionals

Rather than using curly brackets or keywords like **endif**, Python uses leading spaces[2] to determine the scope.

```
if 1 > 2.0:
  print("That doesn't seem right.")

if 1 == 1.0:
  print("That makes sense.")

if (0.0 < 1) and (1 < 2.0):
  print("That also makes sense.")

if (-1.0 > 1) or (-1.0 < 1):
  print("Well it has to lie somewhere...")

if not (1 < 3):
  print("Python is bad at math.")
```

```
That makes sense.
That also makes sense.
Well it has to lie somewhere...
```

[2] One can also use tabs, but I recommend spaces. This way, the leading whitespace is independent of editor preferences.

The in keyword

The **in** keyword is used to locate stuff in lists, dictionaries, files, etc.

```
planets = ["Mercury","Venus" ,"Earth" ,"Mars"   ,
           "Jupiter","Saturn","Uranus","Neptune"]

if "Earth" in planets:
  print("Earth is a planet.")

if not "Pluto" in planets:
  print("You heard about Pluto? That's messed up, right?")
```

```
Earth is a planet.
You heard about Pluto? That's messed up, right?
```

Control and scope

Note that Python uses **in** with **for** loops.

```
i = 0
while i < 3:
  print(i)
  i += 1
  j = 4

print(j)            # Note that Python remembered j

for i in range(3): # Pythonic integer loop
  print(i)
```

```
0
1
2
4
0
1
2
```

More control

Use **continue** to move on to the next iteration of a loop.

```
objects = ["A table", "Paper", "The earth", "A postcard"]

for item in objects:

  if item == "The earth":
    continue

  print(item,"is flat.")
```

```
A table is flat.
Paper is flat.
A postcard is flat.
```

The try/except construction

The keywords **try** and **except** aid with error handling. This is useful if you want the program to keep going if it encounters an error.

```
Ns = 56
Nt = 8

for confNumber in range(40,51): # Runs from 40 to 50

  fileName = "l"+str(Ns)+str(Nt)+"."+str(confNumber)

  try:
    inFile = open(fileName,'r')
  except FileNotFoundError:
    print("ERROR--No file "+fileName+".")
    continue

  inFile.close()
```

```
ERROR--No file l568.44.
```

Functions

Use functions to encapsulate general tasks. Using the keyword **args** one can pass arbitrarily many arguments.

```
def sum2(a,b):
  return a+b

def sumGeneral(*args):
  result = 0.
  for x in args:   # args is a list
    result += x
  return result

print( sum2(1,3.) )
print( sumGeneral(1,7,4.,1e-3) )
```

```
4.0
12.001
```

Using another library

Other libraries/modules[3] are accessed through the **import** keyword.

```
import numpy as np

from latqcdtools.scales_hisq import fk_PDG_2018

fK = fk_PDG_2018("MeV") * np.sqrt(2.)

print( np.sqrt(2.) )
print( "Experimental f_k is %6.2f [MeV]" % fK )
```

```
1.4142135623730951
Experimental f_k is 155.72 [MeV]
```

[3] You can also use this for personal “header” files and parameter files.

What is numpy?

`numpy` is the basic library for scientific computing, with

- multidimensional array objects;
- random sampling;
- fast array operations;
- linear algebra; and
- statistics.

Can be invoked using

```
import numpy
import numpy as np # A common way of doing it
```

Mathematical functions

numpy contains many functions that are useful for scientific calculations. Calculations using numpy are relatively[4] fast.

```
import numpy as np

pi = np.pi
print( pi )
print( np.sin(pi) )
print( np.cos(pi) )
print( np.exp(1j*pi) ) # e^i*pi
```

```
3.141592653589793
1.2246467991473532e-16
-1.0
(-1+1.2246467991473532e-16j)
```

[4] numpy arrays are localized in memory, and its functions are implemented in C and parallelized. It's not uncommon to experience an $\mathcal{O}(100)$ speedup when switching your implementation from loops to numpy.

The numpy array

You can make numpy arrays by hand or from lists.

```
import numpy as np

anArray    = np.array([1., 2., 3., 4.])
a2DArray   = np.array([[1., 2.], [3., 4.], [5., 6.]])

aList      = [1., 2., 3.]
nowAnArray = np.array(aList)

print(anArray)
print(a2DArray)
print(aList)
```

```
[1. 2. 3. 4.]
[[1. 2.]
 [3. 4.]
 [5. 6.]]
[1.0, 2.0, 3.0]
```

The `numpy` array, continued

Axes are defined for N-D arrays. A 2D array has axis 0 vertically down, and axis 1 horizontally across.

```
import numpy as np

a2DArray = np.array([[1., 2.], [3., 4.], [5., 6.]])

print( a2DArray )
print( a2DArray.ndim )
print( a2DArray.shape )
print( a2DArray.sum(axis=1) )
```

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
2
(3, 2)
[ 3.  7. 11.]
```

numpy functions on arrays

There exist many functions taking numpy arrays as arguments.

```
import numpy as np

testArray = np.array([1., 3., 2., 5., 4., 0.])

print( np.mean(testArray) )
print( np.median(testArray) )
print( np.max(testArray) )
print( np.min(testArray) )
print( np.sort(testArray) )
```

```
2.5
2.5
5.0
0.0
[0. 1. 2. 3. 4. 5.]
```

numpy functions on arrays, continued

Useful are fast, element-wise methods, which can replace loops.

```
import numpy as np

testArray = np.array([1., 3., 2., 5., 4., 0.])

print( 2 * testArray + testArray )
print( np.exp(testArray) )
print( testArray**2 )
print( np.sqrt(testArray) )
```

```
[ 3.  9.  6. 15. 12.  0.]
[ 2.71828183  20.08553692   7.3890561  148.4131591   54.59815003  1.  ]
[ 1.  9.  4. 25. 16.  0.]
[ 1.    1.73205081 1.41421356 2.23606798 2.    0.  ]
```

File IO

PROBLEM: I have a file `numbers.d` that is as follows:

```
1 2
3 4
5 6
```

I would like to

- read from `numbers.d`;
- add column 1 with column 2; and
- output to `theirSum.d`.

How is this accomplished in Python?

File IO, continued

SOLUTION: Use the **loadtxt** and **savetxt** functions

```
import numpy as np

inData  = np.loadtxt('numbers.d',unpack=True)

col0    = inData[0]
col1    = inData[1]

outData = col0 + col1

np.savetxt('theirSum.d',outData,fmt='%10.5f')
```

`theirSum.d`:

```
   3.00000
   7.00000
  11.00000
```

Linear algebra

One can perform basic linear algebra operations on arrays.

```python
import numpy as np

M    = np.array([[1., 2.], [3., 4.]])
I2x2 = np.eye(2)

print( I2x2 )
print( np.dot(M, M) )
print( M @ M )
print( np.trace(I2x2) )
```

```
[[1. 0.]
 [0. 1.]]
[[ 7. 10.]
 [15. 22.]]
[[ 7. 10.]
 [15. 22.]]
2.0
```

Linear algebra, continued

On square arrays one has more options.

```
import numpy as np

M = np.array([[1., 2.], [3., 4.]])

print( np.transpose(M) )
print( np.linalg.inv(M) )
print( np.linalg.eig(M) )
```

```
[[1. 3.]
 [2. 4.]]
[[-2.   1. ]
 [ 1.5 -0.5]]
(array([-0.37228132,  5.37228132]),
 array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```

And much more

Some other useful methods for `numpy` arrays include:

- other data types (`ulonglong`, `complex128`);
- shape manipulation (`reshape`, `moveaxis`);
- stacking and splitting arrays (`hstack`, `vsplit`);
- basic statistics (`cov`, `var`); and
- random numbers.

It may help to have a look at the quickstart tutorial to get a first idea about what other features are available.

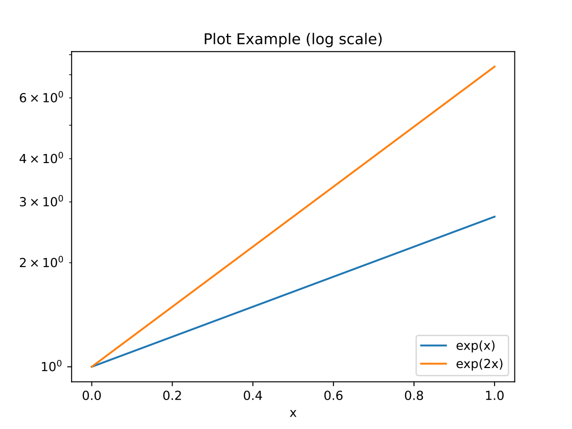
Plotting

One can start plotting right away using `matplotlib.pyplot`. In general when using `matplotlib`, one has to pay careful attention to the order of the commands.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 1, 101)
y = np.exp(x)
z = np.exp(2*x)

plt.xlabel('x')
plt.yscale('log')
plt.title('Plot Example (log scale)')

plt.plot(x,y,label='exp(x)')
plt.plot(x,z,label='exp(2x)')

plt.legend(loc='lower right')

plt.savefig('plotExample.pdf')
plt.show()
```

Plotting

`plotExample.pdf`:



[5] A possible talk for the future.

What is scipy?

`scipy` is a core library for scientific computing, with methods for

- numerical integration;
- optimization;
- interpolation;
- statistics;
- and more.

It is invoked by

```
import scipy
```

Some things you can do with it

- Bessel functions (`special.jn_zeros`)
- Integration (`integrate.quad`)
- Optimization (`optimize.minimize`)
- Linear and cubic spline interpolation (`interpolate.interp1d`)
- More involved statistics (`stats.hypergeom`)
- More involved linear algebra (`linalg.schur`)
- Sparse eigenvalue problems (ARPACK)

The scipy tutorial gives a more thorough introduction. To help simplify your notation, it may be good to use the **import/as** construct.

```
import scipy.special as sc
```

1. Why Python?
2. Some Python basics
3. `numpy`, and `matplotlib`
4. `scipy` advertisement
5. Wrap-up

What we didn't cover

With limited time/knowledge, I could only cover a few features to get you started with scientific computing using Python. Other useful features and packages include, but are not limited to:

- Classes and objects[6]
- Finding files in a Unix-like way (e.g. `glob`)
- Passing command line arguments to your script
- Logging and error handling

Before you write something in Python, you've got to ask yourself one question: "**Is there a package for this?**"

[6] A probable talk for the future.

Summary

Python is a programming language

- that's **easy** and **fun** to use;
- is extremely **readable**;
- is **flexible**; and
- has access to powerful **scientific** libraries.

`numpy` is useful for streamlined **array manipulation**, **special functions**, **high precision** calculations, and more.

`matplotlib` lets you make easily make professional plots within your own programming framework.

`scipy` is useful for **optimization**, **curve fitting**, and more.

I invite you to give Python a try if you haven't already.

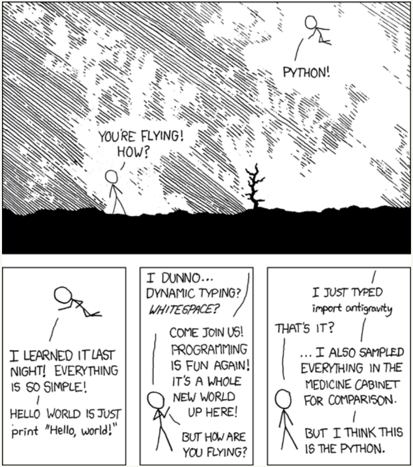


image from XKCD

Thanks for listening!