Introductory Python Tutorial with Resources

Physics 512 Computational Physics

Prof. Jonathan Sievers
TAs: Marcus Mayfield (marcus.merryfield@mail.mcgill.ca)
Sabrina Berger (sabrina.berger@mail.mcgill.ca)

Tutorial Overview

- Basic Unix & Github commands
- Brief review of
  - Data Types
  - Assignments
  - Functions
  - Conditionals
  - Loops
  - Useful Numpy features
  - Matplotlib 1D and 2D plotting
- Practice with a few sample coding tasks

Certain slides are courtesy of Orion Lyau and Nicholas Rui.

Data Types in Python

Type	Example
`int`	`10`
`float`	`3.14`
`str`	`"Cat", 'any text'`
`bool`	`True, False` (note first capital letter)

More thorough introduction to data types: https://realpython.com/python-data-types/

Data Types in Python

Type	Example
`tuple`	`(1, 3, 4)` `(note immutable)`
`list`	`['Cat', 'Dog', 2]`
`dict`	`{'key':'value', 'coding':'fun'}`
`set`	`{'Apple', 'Banana', 'Orange'}`

More thorough introduction to data types: https://realpython.com/python-data-types/

Assignment

Assignment takes the **expression** on the right hand side of the **=** and **binds** it to the **name** on the left hand side of the **=**. The **=** is the assignment **operator**.

```
pi = 3.15159
tau = 2 * pi
```

This is the simplest means of **abstraction**. Here we can use names to refer to more complicated things.

Type Conversion

Example: `str` to `int`

```
my_int = 1
my_string = str(my_int)
```

Example: `int` to `str`

```
new_string = '8'
new_int = int(new_string)
```

Anatomy of a Function

Functions are defined in the following format:

```
def <name>(<parameters>):
    return <expression>
```

It is important to indent each line underneath the **`def`** statement

Anatomy of a Function

Example:

```
def hypotenuse(a, b):
    return (a ** 2 + b ** 2) ** 0.5
```

Name

Parameters

Expression to return

Usage:

```
hypotenuse(3, 4)
```

Arguments

Scope

When you're in the body (indented region) of a function, the variables you create only exist inside that function (i.e. within the **scope** of the function).

```
def times_five(x):
    result = x * 5
    return result

def times_ten(x):
    result = x * 10
    return result
```

The `result` in `times_five` and `times_ten` are independent of each other, and each `result` is not accessible outside their respective functions.

Comparison Operators

Comparisons evaluate to `True` or `False` and include the following operators:

- `==` equal to
- `!=` not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to
- `<=` less than or equal to

Examples:

```
1 == 1  →  True
1 != 1  →  False
5 > 3   →  True
```

Operator

Operand

Boolean Operators

Boolean operators (sometimes also called **logical operators**) evaluate to `True` or `False` and include the following operators:

```
and
or
not
```

Examples:

```
True and False  →  False
True or False   →  True
not False       →  True
```

Conditional Statements

Example:

```
if <expression>:
    <stuff>
elif <expression>:
    <stuff>
else:
    <stuff>
```

Stuff inside conditional statements must be indented.

Starts at the top, and the only the **`<stuff>`** under the first true expression is run.

Must start with **`if`** clause, followed by zero or more **`elif`** clauses, and zero or one **`else`** clause.

If an **`else`** clause is present, the stuff there will only run if all **`if`** and **`elif`** expressions evaluate to false.

`while` loop

A **`while`** loop repeats for as long as a given expression is true.

Example:

```
while <expression>:
    <stuff>
```

1. Evaluate **`<expression>`**, and if **`<expression>`** is true, go to step 2
2. Run **`<stuff>`**, then go back to step 1

You need to make sure that **`<expression>`** will eventually evaluate to false, otherwise the loop will repeat indefinitely. If this happens, it is said to be an **infinite loop**.

for loop

A **for** loop repeats as many times as there are entries in the thing you're iterating over.

Example:

```
for <name> in <iterable>:
    <stuff>
```

At each iteration, you will be able to access the current element of **<iterable>** through the loop variable **<name>**.

Things you might often iterate over include lists, tuples, or ranges of numbers. The latter can be represented by **range(n)** where **n** is the end of the range.

`break` and `continue`

In a `for` or `while` loop, the `break` and `continue` statements let you either terminate a loop (`break`) or continue to the next loop iteration (`continue`).

You can also use the `return` statement anywhere within a function (including within a loop) to immediately exit the function and return a value, without needing to execute anything further.

`import numpy as np`

Numpy is the core scientific computing package in Python!

- Numpy arrays (`np.array`) are very similar to Python lists, but with extra features!

Consider: `a = np.array([1,2,3])`

- `a+1 # Prints `array([2,3,4])``
- `a*2 # Prints `array([2,4,6])``
- `a.shape # Prints (3,)`
- `a[0] # Prints `1``

NumPy has many great functionalities to explore! See the documentation

Ex: Array creation

There are a number of ways to generate numpy arrays:

- `a = np.array([[1,2],[3,4]]) # From a (nested) list`
- `b = np.zeros((2,2)) # Generates a 2x2 array of 0s`
- `c = np.ones((2,2)) # Generates a 2x2 array of 1s`
- `d = np.full((2,2),5) # Generates a 2x2 array of 5s`
- `e = np.eye(2) # Generates a 2x2 identity matrix`

... And more! You can also do element-wise math between arrays:

- ``a+c # Prints `array([[2., 3.], [4., 5.]])` ``
- ``a*b # Prints `array([[0., 0.], [0., 0.]])` ``

... etc!

1D/2D Linear/Logarithmically Spaced Arrays

Often we want to generate a series of equally spaced numbers in numpy to feed into a function f(x) or for other uses. This is simple enough:

```
• x = np.linspace(0,1,100) # 100 elements spaced btw 0 & 1
• # NOTE: np.logspace also exists, and does what you think!
• y = f(x) # where f is your desired function
```

For functions of two variables, f(x,y), `np.meshgrid` is also super useful!

```
• x = np.linspace(0,1,3); y = np.linspace(0, 2, 2)
• X, Y = np.meshgrid(x,y)
• Z = f(X, Y) # returns function evaluated on the grid
```

Matrix Operations

NumPy can easily handle pretty much any matrix operation

- `a = np.array([[1,2],[3,4]]); b = np.array([[5,6],[7,8]])`
- `a @ b # this is the matrix multiplication of a & b`
- `# NOTE: recall a * b is element-wise multiplication!`
- `a_inv = np.linalg.inv(a) # matrix inversion`
- `e_val, e_vec = np.linalg.eig(a) # eigenvalues/vectors`

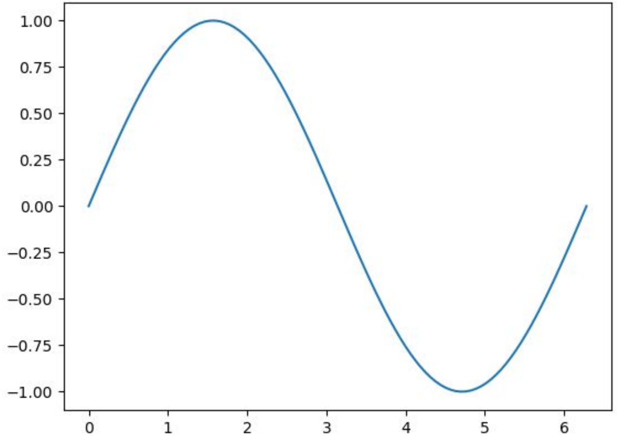
`np.linalg` has a lot of interesting functionalities, and if you want to learn more you can explore what the library has to offer [here]

import matplotlib.pyplot as plt

Matplotlib is the package pretty much everyone uses for plotting data in python. In particular, `matplotlib.pyplot` is the function you'll be using most of the time, and pretty much everyone import it as `plt`!

- `x = np.linspace(0,2*np.pi,100); y = np.sin(x)`
- `plt.plot(x,y); plt.show()`

This will very easily produce a simple plot:

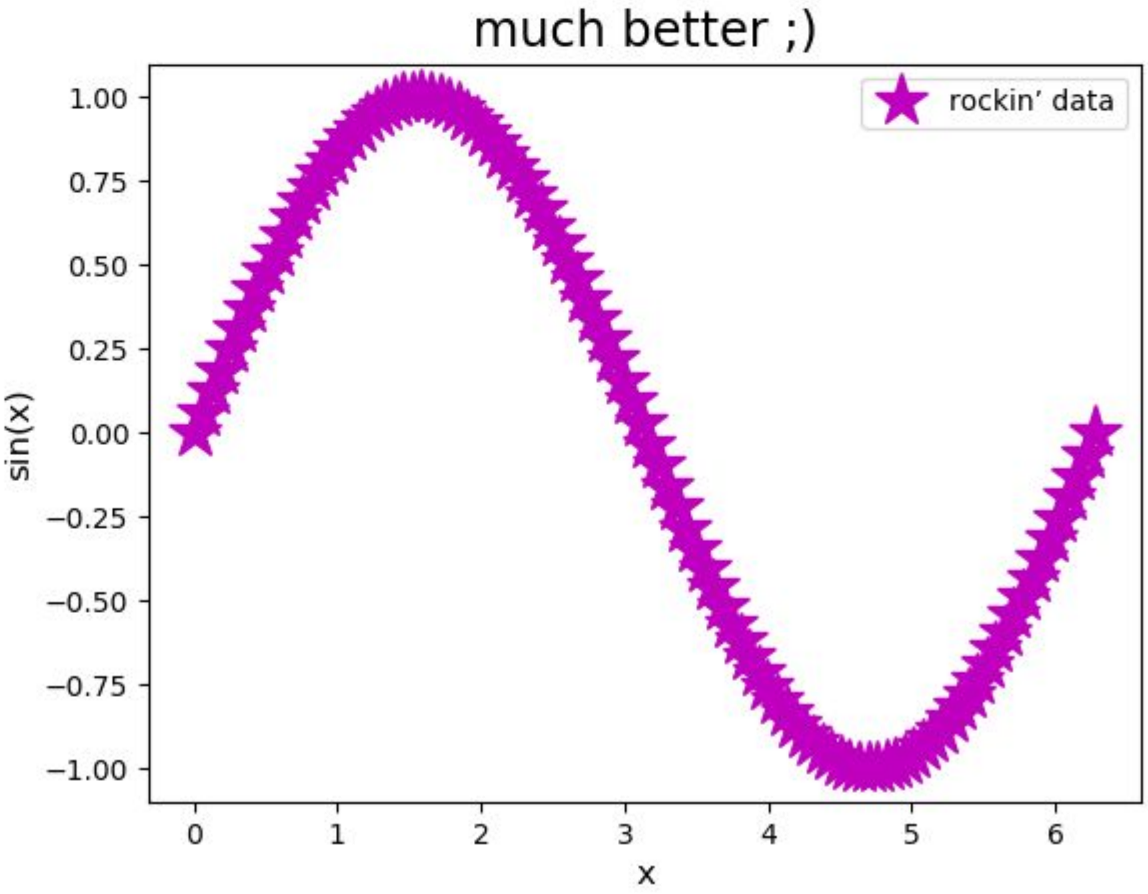


AESTHETIC

“But Marcus!”, you object, “your axes have no labels, and the plot is so bland!”
Well, this is all entirely correct, so let’s spice up our plot a bit

- `fmt = ‘m*’; size=20; label=”rockin’ data”`
- `plt.plot(x,y,fmt=fmt,markersize=size,label=label)`
- `plt.legend(); plt.xlabel(“x”,fontsize=”large”)`
- `plt.ylabel(“sin(x)”, fontsize=”large”)`
- `plt.title(“much better ;)”, fontsize=”xx-large”)`

Which gives us...



Obviously your day-to-day plots don't have to be as extravagant as... *this*, but it's worthwhile to play around with the formatting options in `plt.plot` to find the best way to represent your data!

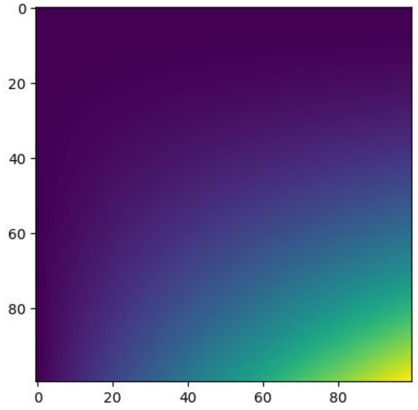
2D Plots

Now say we want to plot a function of two variables! Matplotlib actually has a number of ways to do this, each with its advantages and disadvantages.

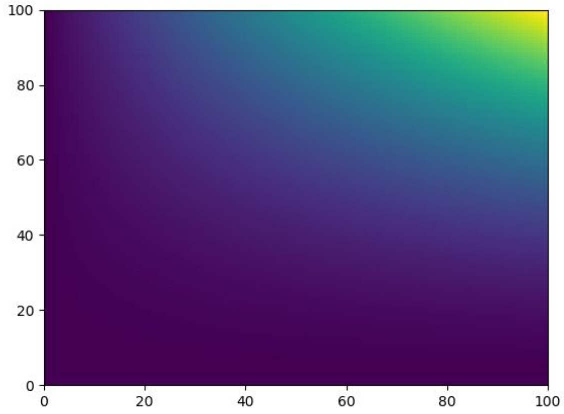
- `x = np.linspace(0,1,50); X, Y = np.meshgrid(x,x)`
- `Z = X * Y**2 # some arbitrary 2D function`
- `plt.imshow(Z);plt.show()`
- `plt.pcolormesh(Z);plt.show()`
- `plt.contourf(Z);plt.show()`

NOTE: There are many colour maps to choose from and I guarantee at some point in your life you'll spend way too much time experimenting to find juuuust the right one! But more seriously, a good choice of colour map is important.

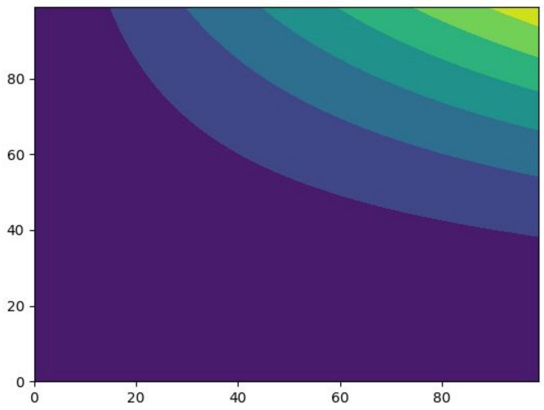
imshow



pcolormesh



contourf



Each of these functions has their advantages and disadvantages, and I recommend doing a bit of reading on each to figure out which is best for your use case.
If you're curious, I find myself using `imshow` the most of all of them, because it's (in my opinion) the simplest.
Also, the colour bars are missing, but a simple `plt.colorbar()` call before showing/saving the figure will remedy this!

That’s all!

Any questions before some exercises?

Resources

Very new to Python? Try reading the first few chapters of this book for a thorough introduction to Python: Python for Astronomers

NumPy Basics

From Python to NumPy (more advanced)

The Python Data Science Handbook

Some practice tutorials

Some examples of animations with matplotlib

Computer Memory (as discussed in first lecture)