



UNIVERSITY  
OF ALBERTA

# Introduction to Programming and Data Analysis

Dark Matter Week – Tuesday

William Woodley



# Programming Languages

- There are a number of languages and tools commonly used in physics:

Fortran

Python

Bash

C

MATLAB

LaTeX

C++

Mathematica

IDL

C#

Maple

Verilog

Java

R

Visual Studio

# Programming Languages

- There are a number of languages and tools commonly used in physics:

Fortran

**Python**

Bash

C

MATLAB

LaTeX

C++

Mathematica

IDL

C#

Maple

Verilog

Java

R

Visual Studio

- We're going to be focussing on Python this week.

# Why Python?

- There are a number of reasons to prioritise Python over other languages:
  1. It's free.
  2. It can be used on any platform.
  3. It has a lot of scientific libraries with very good documentation, and more are constantly being added.
  4. It has a huge scientific community, and its use in physics is growing. It has become the default programming language for the physics community.
  5. It has good support and an active development community.
  6. It is a high-level language, and is much easier to learn for beginners than other popular languages, like C++.

# The Zen of Python

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one -- and preferably only one -- obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

# Using Python in a Jupyter Notebook

- You can write Python code in text files and run them from the terminal (or write Python in the terminal directly).
- A more modern way of writing Python code is in a Jupyter notebook.

<https://cybera.syzygy.ca>



# Python Basics

# Arithmetic Operators

- The basic arithmetic operators in Python are:

<u>Operation:</u>	<u>Operator:</u>	<u>Example:</u>
Addition	+	"Hi" + " there"
Subtraction	-	5 - 3, -1
Multiplication	*	2*"Hello"
Division	/	15/3
Exponentiation	**	10**2
Modulus	%	25%2
Floor Division	//	26.5//2

- Use brackets to organise complicated expressions.

```
(5**(10 + 3))/(((75 - 6)*(32**2)) + (1/5))
```



# Data Types

- There are a number of different data types in Python.
- Types are assigned dynamically, so you do not have to specify types.
- To get a variable's type, use `type(var)`.

<u>Type:</u>	<u>Name:</u>	<u>Example:</u>
<b>Numeric</b>	int, float, complex	4, 36.8, 1.2e6, 6+8j
<b>Text</b>	str	"Hello, World!"
<b>Sequence</b>	list, tuple, range	[1, 2, 3], ("a", "b", "c")
<b>Mapping</b>	dict	{"a": 1, "b": 2, "c": 3}
<b>Sets</b>	set, frozenset	{"hello", 2+5.3j, (0)}
<b>Boolean</b>	bool	True, False
<b>Binary</b>	bytes, bytearray, memoryview	b"Hello"
<b>None</b>	NoneType	None

# Assigning Values

- The **=** sign is the assignment operator in Python. This assigns values to variable names.
- You can assignment multiple variables at once.
- Assignment happens after the evaluation of the expression.
- There are some shorthand notations for certain operations.

```
var = 3
```

# Assigning Values

- The **=** sign is the assignment operator in Python. This assigns values to variable names.
- You can assignment multiple variables at once.
- Assignment happens after the evaluation of the expression.
- There are some shorthand notations for certain operations.

```
var1, var2 = 3, 4
```

# Assigning Values

- The `=` sign is the assignment operator in Python. This assigns values to variable names.
- You can assignment multiple variables at once.
- Assignment happens after the evaluation of the expression.
- There are some shorthand notations for certain operations.

```
var1 = var1*3
```

# Assigning Values

- The `=` sign is the assignment operator in Python. This assigns values to variable names.
- You can assignment multiple variables at once.
- Assignment happens after the evaluation of the expression.
- There are some shorthand notations for certain operations.

```
var1 *= 3
```

# Variable Names

- You can use letters, numbers, and underscores for variable names.
- Variable names are case-sensitive.
- Don't start variable names with numbers.
- It's good to be detailed with your variable names. Long names aren't a bad thing.

## Valid Variable Names:

n	E_sq
N	F2
variable_name	_
momentum	ex_var_8

## Invalid Variable Names:

2pi	>ab
for	value!
#velocity	lambda
alpha\$beta	4th_term

# Keywords

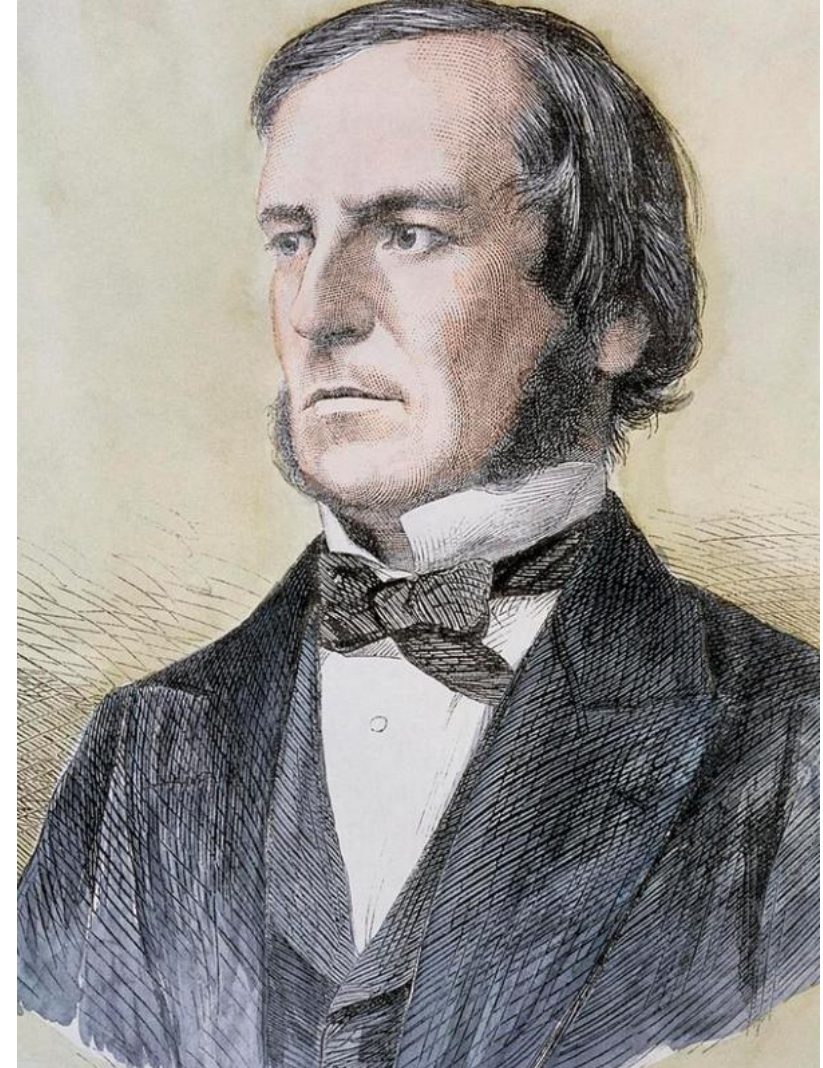
- There are a number of reserved key words in Python. These are words Python uses to understand your program, so you should not and cannot overwrite them.

<code>False</code>	<code>await</code>	<code>else</code>	<code>import</code>	<code>pass</code>
<code>None</code>	<code>break</code>	<code>except</code>	<code>in</code>	<code>raise</code>
<code>True</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>and</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>as</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>
<code>assert</code>	<code>del</code>	<code>global</code>	<code>not</code>	<code>with</code>
<code>async</code>	<code>elif</code>	<code>if</code>	<code>or</code>	<code>yield</code>

# Boolean Algebra

- Boolean algebra is binary system of logic. It was created by Charles Boole in 1847.
- Variables can have two values: TRUE, FALSE.
- There are three main operators: NOT, AND, OR.
- We understand logical operations through truth tables.

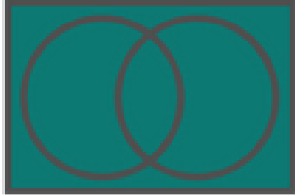
A	B	NOT A	A AND B	A OR B
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F



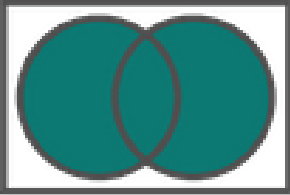
Charles Boole (1815 – 1864)



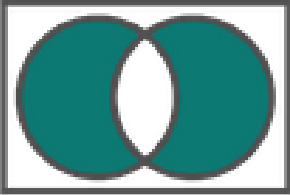
# Boolean Algebra



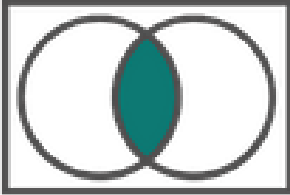
Tautology



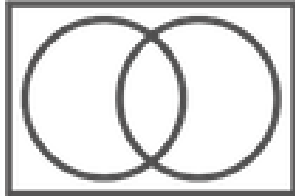
A OR B



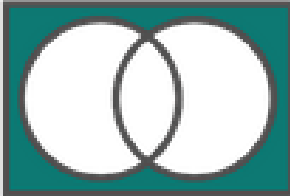
A XOR B



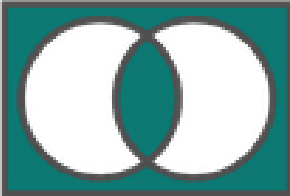
A AND B



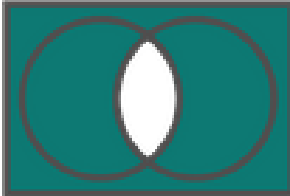
Contradiction



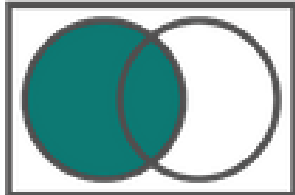
A NOR B



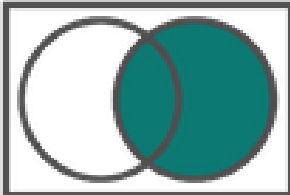
A XNOR B



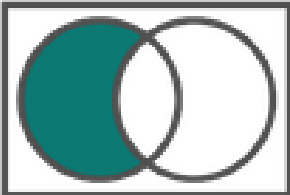
A NAND B



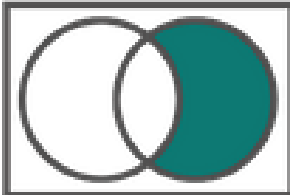
A



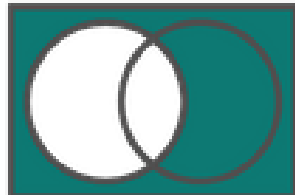
B



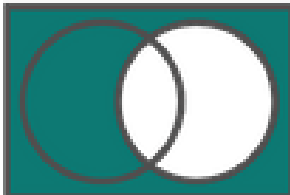
A without B



B without A



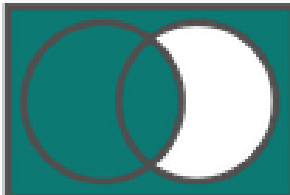
NOT A



NOT B



A IMPL B



B IMPL A

# Boolean Operators

- There are three main Boolean operators in Python:

<u>Operation:</u>	<u>Operator:</u>	<u>Example:</u>
Not	<code>not</code>	<code>not x</code>
And	<code>and</code>	<code>x and y</code>
Or	<code>or</code>	<code>x or y</code>

- Operations can be combined. Use brackets to make things clear.

```
a = True  
b = False
```

```
(a and b) or not (b or (a or b))
```

# Comparison Operators

- The basic comparison operators are:

<u>Operation:</u>	<u>Operator:</u>	<u>Example:</u>
Less Than	<	x < 1e5
Less Than or Equal to	<=	x <= -1
Greater Than	>	x > 25
Greater Than or Equal to	>=	x >= 1
Equal to	==	x == 0
Not Equal to	!=	x != 10
Identity	is	x is y
Non-Identity	is not	x is not None
Inclusion	in	i in [1, 2, 3]
Non-Inclusion	not in	i not in [4, 5]
Other	isinstance()	x.isinstance(str)

# Truthy and Falsy

- Certain values are equivalent to **True**. These are known as truthy values. Some are equivalent to **False**. These are falsy.

## Truthy:

### True

Non-zero numbers: `1`, `2`, etc.

`np.inf`, `NaN`

Non-empty strings: `"hello"`

Non-empty lists: `[1, 2, 3]`

Non-empty tuples: `(4, 5)`

Other: `{"a": 1}`, etc.

## Falsy:

### False

Zeros: `0`, `0.0`, `0j`

### None

Empty string: `""`

Empty list: `[]`

Empty tuple: `()`

Other: `{}`, `range(0)`, etc.

- Check the truthiness of a variable using `bool(var)`.

# Python Standard Library

- There are a number of built-in function in Python. Some commone ones include:

<code>abs()</code>	<code>complex()</code>	<code>frozenset()</code>	<code>iter()</code>	<code>print()</code>
<code>all()</code>	<code>delattr()</code>	<code>getattr()</code>	<code>len()</code>	<code>property()</code>
<code>any()</code>	<code>dict()</code>	<code>globals()</code>	<code>list()</code>	<code>range()</code>
<code>bin()</code>	<code>dir()</code>	<code>hasattr()</code>	<code>locals()</code>	<code>round()</code>
<code>bool()</code>	<code>divmod()</code>	<code>help()</code>	<code>max()</code>	<code>set()</code>
<code>bytearray()</code>	<code>enumerate()</code>	<code>hex()</code>	<code>memoryview()</code>	<code>sorted()</code>
<code>bytes()</code>	<code>eval()</code>	<code>id()</code>	<code>min()</code>	<code>sum()</code>
<code>callable()</code>	<code>exec()</code>	<code>input()</code>	<code>open()</code>	<code>type()</code>
<code>chr()</code>	<code>float()</code>	<code>int()</code>	<code>ord()</code>	<code>tuple()</code>
<code>classmethod()</code>	<code>format()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>zip()</code>

# Python Standard Library

- There are a number of built-in function in Python. Some commone ones include:

<code>abs()</code>	<code>complex()</code>	<code>frozenset()</code>	<code>iter()</code>	<code>print()</code>
<code>all()</code>	<code>delattr()</code>	<code>getattr()</code>	<code>len()</code>	<code>property()</code>
<code>any()</code>	<code>dict()</code>	<code>globals()</code>	<code>list()</code>	<code>range()</code>
<code>bin()</code>	<code>dir()</code>	<code>hasattr()</code>	<code>locals()</code>	<code>round()</code>
<code>bool()</code>	<code>divmod()</code>	<code>help()</code>	<code>max()</code>	<code>set()</code>
<code>bytearray()</code>	<code>enumerate()</code>	<code>hex()</code>	<code>memoryview()</code>	<code>sorted()</code>
<code>bytes()</code>	<code>eval()</code>	<code>id()</code>	<code>min()</code>	<code>sum()</code>
<code>callable()</code>	<code>exec()</code>	<code>input()</code>	<code>open()</code>	<code>type()</code>
<code>chr()</code>	<code>float()</code>	<code>int()</code>	<code>ord()</code>	<code>tuple()</code>
<code>classmethod()</code>	<code>format()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>zip()</code>

**Essential**; Helpful for math and operations; Converting data types; Other

# Libraries

- Base Python doesn't have functions for all of the things we want to do.
- Luckily, there are thousands of libraries out there that we can import. Then we have access to all of the functions in those libraries.
- Some of the most common libraries are:
  - `os` Working with files, folders, etc.
  - `sys` Working with files, folders, etc.
  - `numpy` Math, arrays, matrices
  - `scipy` Data analysis (optimisation, fitting, interpolation, etc.)
  - `matplotlib` Making graphs
  - `pandas` Importing data from files
  - `tensorflow` Neural networks
- And thousands more!

# Installing Libraries

```
$ pip install iminuit
$
$ pip install numpy==1.23.0
$
$ pip show numpy
$
$ python3 --version
$
$ which python3
$
$ python3
```

- Python libraries are stored on PyPI: the Python Project Interface.

<https://pypi.org>

- They can be installed from a terminal using pip.
- Many are open-source. The source codes can be found on GitHub, along with examples and documentation.

<https://github.com>



# Documentation

- Widely used Python libraries have full documentation for all functions.

LAPACK functions for Cython  
Interpolative matrix decomposition  
( `scipy.linalg.interpolative` )  
Miscellaneous routines ( `scipy.misc` )  
Multidimensional image processing  
( `scipy.ndimage` )  
Orthogonal distance regression  
( `scipy.odr` )  
**Optimization and root finding**  
( `scipy.optimize` )  
Cython optimize zeros API  
Signal processing ( `scipy.signal` )  
Sparse matrices ( `scipy.sparse` )  
Sparse linear algebra  
( `scipy.sparse.linalg` )  
Compressed sparse graph routines  
( `scipy.sparse.csgraph` )  
Spatial algorithms and data structures  
( `scipy.spatial` )  
Distance computations  
( `scipy.spatial.distance` )  
Special functions ( `scipy.special` )  
Statistical functions ( `scipy.stats` )  
Result classes

## scipy.optimize.curve\_fit

```
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=(-inf, inf), method=None, jac=None, *, full_output=False,
**kwargs) \[source\]
```

Use non-linear least squares to fit a function, `f`, to data.

Assumes `ydata = f(xdata, *params) + eps`.

### Parameters: `f` : callable

The model function, `f(x, ...)`. It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

### `xdata` : array\_like

The independent variable where the data is measured. Should usually be an M-length sequence or an (k,M)-shaped array for functions with `k` predictors, and each element should be float convertible if it is an array like object.

### `ydata` : array\_like

The dependent data, a length M array - nominally `f(xdata, ...)`.

### `p0` : array\_like, optional

Initial guess for the parameters (length N). If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a `ValueError` is raised).

# Importing Libraries

- There are different syntaxes for importing libraries:

```
import numpy
numpy.linspace(0, 10, 100)
```

```
import matplotlib.pyplot as plt
plt.plot(x, y)
```

```
from scipy.optimize import curve_fit, minimize
curve_fit(function, x, y)
```

```
from scipy.interpolate import interpn as inn
inn(points, values, xi)
```

```
from pandas import *
read_csv("file.csv")
```

# Arrays

- Arrays are one of the most versatile and useful data types in Python. Almost everything I do in Python is done using arrays.
- Arrays have **elements** and indices.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

0    1    2    3    4    5    6    7    8

# Arrays

- Arrays are one of the most versatile and useful data types in Python. Almost everything I do in Python is done using arrays.
- Arrays have **elements** and **indices**.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

0    1    2    3    4    5    6    7    8

# Arrays

- Arrays are one of the most versatile and useful data types in Python. Almost everything I do in Python is done using arrays.
- Arrays have **elements** and **indices**.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

                  0   1   2   3   4   5   6   7   8

# Arrays

- Arrays are one of the most versatile and useful data types in Python. Almost everything I do in Python is done using arrays.
- Arrays have **elements** and **indices**.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

0    1    2    3    4    5    6    7    8

- To pull out a certain element of an array, put the index of that element in square brackets after the variable name.

```
[In]: print(arr[2])  
[Out]: 8
```

- **Important:** Python begins counting at 0!

# Other Ways of Indexing

- You can count backwards in Python by indexing with negative numbers.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

	0	1	2	3	4	5	6	7	8
	-9	-8	-7	-6	-5	-4	-3	-2	-1

# Other Ways of Indexing

- You can count backwards in Python by indexing with negative numbers.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

	0	1	2	3	4	5	6	7	8
	-9	-8	-7	-6	-5	-4	-3	-2	-1

- You can use colons to slice sections of an array.

```
[In]: print(arr[6:])  
[Out]: [2, 6, 15]
```

```
[In]: print(arr[:3])  
[Out]: [5, 3, 8]
```

```
[In]: print(arr[2:-4])  
[Out]: [8, 2, 11]
```



# Other Ways of Indexing

- You can count backwards in Python by indexing with negative numbers.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

- You can use colons to slice sections of an array.

```
[In]: print(arr[6:])  
[Out]: [2, 6, 15]
```

```
[In]: print(arr[:3])  
[Out]: [5, 3, 8]
```

```
[In]: print(arr[2:-4])  
[Out]: [8, 2, 11]
```

# Other Ways of Indexing

- You can count backwards in Python by indexing with negative numbers.

```
arr = np.array([5, 3, 8, 2, 11, 4, 2, 6, 15])
```

	0	1	2	3	4	5	6	7	8
	-9	-8	-7	-6	-5	-4	-3	-2	-1

- You can use colons to slice sections of an array.

```
[In]: print(arr[6:])  
[Out]: [2, 6, 15]
```

```
[In]: print(arr[:3])  
[Out]: [5, 3, 8]
```

```
[In]: print(arr[2:-4])  
[Out]: [8, 2, 11]
```

# Boolean Indexing

- You can select the elements of an array that satisfy a given condition by passing that condition as an index to the array.

```
[In]: arr[arr > 4]
[Out]: [5, 8, 11, 6, 15]
```

- The condition may be specified with a different array:

```
[In]: sel = np.array([10, 10, 1, 1, 1, 1, 1, 1, 1])
      arr[sel > 4]
[Out]: [5, 3]
```

- The elements of `arr` corresponding to the indices for which the `sel` array are greater than 4 are selected.

# Defining Arrays

- There are a number of ways to create arrays.
- Almost every function in the NumPy library return arrays.
- You can do math with arrays!

```
my_list = [1, 2, 3]

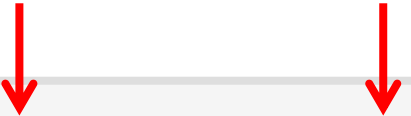
arr1 = np.array(my_list)           # Converts a list into an array
arr2 = np.linspace(0, 10, 100)    # Range defined by number of elements
arr3 = np.arange(0, 10, 2)        # Range defined by step size
arr4 = np.zeros(my_list)          # Array or matrix of zeros of given shape

print(my_list*2)
print(arr1*2)
```

# Dictionaries

- Dictionaries are like lists and arrays, but instead of indexing with integers, you index with strings.
- Dictionaries have **keys** and **values**.

```
ingredients = {"hot_peppers": 12,  
              "onions": 3,  
              "cilantro": 1,  
              "tomatoes": 4}
```



```
[In]: print(ingredients["tomatoes"])
```

```
[Out]: 4
```

# Matrices

- Arrays can have multiple dimensions. Matrices are 2D rectangular arrays.

$$\left[ \begin{array}{ccc} [1 & 2 & 3] \\ [4 & 5 & 6] \\ [8 & 9] \end{array} \right] 7$$

**List of Lists**

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

**Matrix**

- They are indexed with two indices: `matrix[i, j]`.
  - **First Index:** Row
  - **Second Index:** Column

# Shaping Matrices

- You can reshape matrices from arrays or other matrices:

```
arr = np.linspace(1, 24, 24)
matrix = np.reshape(arr, newshape = (6, 4))
```

- You can print the shape of the matrix using the shape attribute:

```
[In]: print(matrix.shape)
[Out]: (6, 4)
```

# Indexing Matrices

- You can select specific elements of a matrix:

$$\text{matrix}[-1, 2] = \begin{matrix} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \mathbf{0} & 1 & 2 & 3 & 4 \\ \mathbf{1} & 5 & 6 & 7 & 8 \\ \mathbf{2} & 9 & 10 & 11 & 12 \\ \mathbf{3} & 13 & 14 & 15 & 16 \\ \mathbf{4} & 17 & 18 & 19 & 20 \\ \mathbf{5} & 21 & 22 & 23 & 24 \end{matrix} = 23$$



# Indexing Matrices

- You can slice ranges of rows and columns:

`matrix[2:4, 1:]` =

**[2, 4)**

Not included in the selection

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16
4	17	18	19	20
5	21	22	23	24

=

10	11	12
14	15	16

# Indexing Matrices

- You can select multiple rows and columns:

$$\text{matrix}[3, [0, 2]] = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix} = \begin{bmatrix} 13 & 15 \end{bmatrix}$$

# Indexing Matrices

- You can combine all of these if you need to:

$$\text{matrix}[[0, 4], 1:-1] = \begin{matrix} & \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \\ \mathbf{0} & 1 & 2 & 3 & 4 \\ \mathbf{1} & 5 & 6 & 7 & 8 \\ \mathbf{2} & 9 & 10 & 11 & 12 \\ \mathbf{3} & 13 & 14 & 15 & 16 \\ \mathbf{4} & 17 & 18 & 19 & 20 \\ \mathbf{5} & 21 & 22 & 23 & 24 \end{matrix} = \begin{bmatrix} 2 & 3 \\ 18 & 19 \end{bmatrix}$$

# More Math with Arrays

- You can find the maximum, minimum, find the mean, take sums, etc.

```
np.min(matrix)
```

```
np.max(matrix)
```

```
np.mean(matrix)
```

```
np.mean(matrix, axis = 0)
```

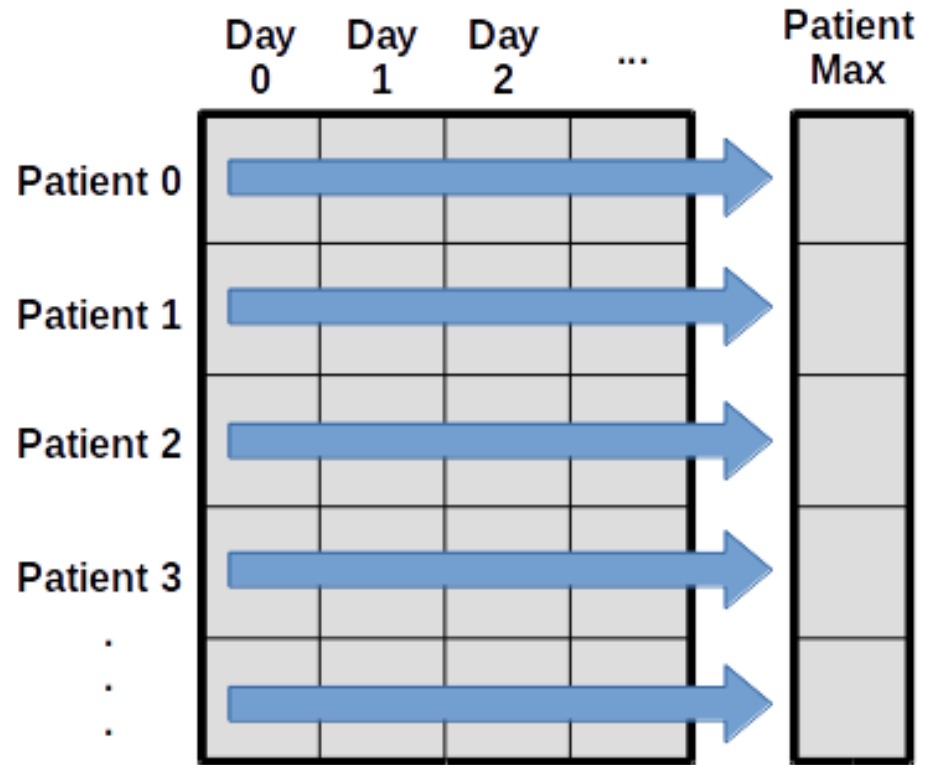
```
np.mean(matrix, axis = 1)
```

```
np.sum(matrix)
```

```
np.sum(matrix, axis = 0)
```

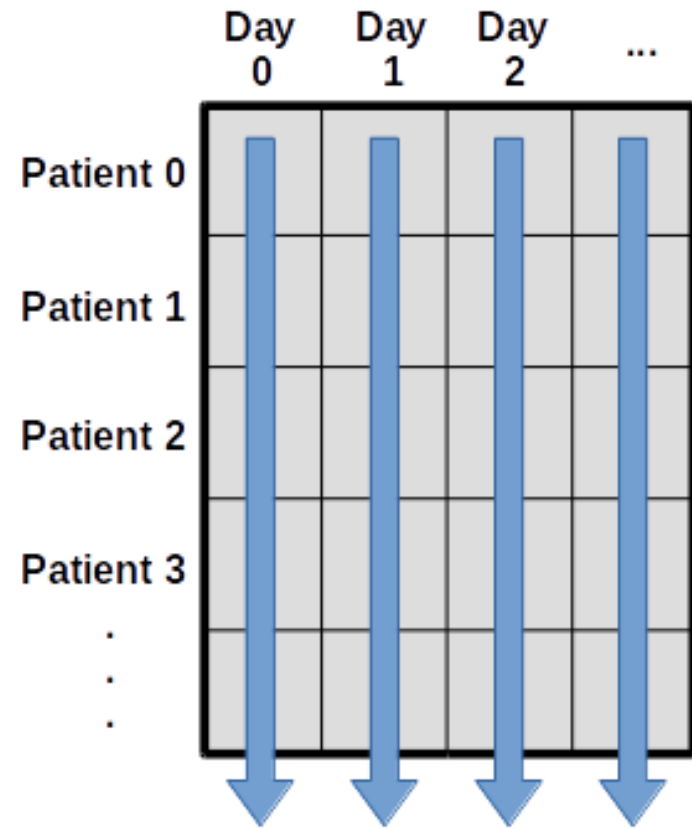
```
np.sum(matrix, axis = 1)
```

# More Math with Arrays



Max for each patient

`numpy.max(data, axis=1)`



Average for each day

`numpy.mean(data, axis=0)`

# Syntax

- A lot of programming languages use curly brackets and semicolons to indicate syntax. Python, however, uses whitespace.

## C++:

```
for (int i = 0; i < 10; i++) {  
    cout << "i = " << i << "\n";  
}
```

## Python:

```
for i in range(10):           # for-Loop  
    print("i = ", i)
```

- You can use single ( ' ) or double ( " ) quotation marks in Python interchangeably.
- Lines starting with # are comments. Python completely ignores them when running your code.

# Syntax

- A lot of programming languages use curly brackets and semicolons to indicate syntax. Python, however, uses whitespace.

## C++:

```
for (int i = 0; i < 10; i++) {  
    cout << "i = " << i << "\n";  
}
```

## Python:

```
for i in range(10):           # for-Loop  
    print("i = ", i)
```



**4 spaces (NOT a tab)**

- You can use single ( ' ) or double ( " ) quotation marks in Python interchangeably.
- Lines starting with # are comments. Python completely ignores them when running your code.

# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]

for i in range(len(my_list)):

    print(my_list[i])
```

[Out]:

```
4
3
8
2
9
```



# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]

for i in range(len(my_list)): Creates iterator to loop through  
the values [0, 1, 2, 3, 4].
    print(my_list[i])
```

[Out]:

# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]
```

```
for i in range(len(my_list)): Sets the value of i to the first  
value in the iterator: 0.
```

```
    print(my_list[i])
```

[Out]:

# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]
```

```
for i in range(len(my_list)):
```

```
    print(my_list[i])
```

} Executes all the code  
inside the loop.

[Out]:

```
4
```

# for-Loops


- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]
```

```
for i in range(len(my_list)):
```

```
    print(my_list[i])
```



**Sets the value of i to the next value in the iterator: 1.**

[Out]:

```
4
```

# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]

for i in range(len(my_list)):
    print(my_list[i])
```

} Executes all the code  
inside the loop again.

[Out]:

```
4
3
```

# for-Loops

- We often want to iterate through lists of values. The best way of doing this is usually by using a for-loop.

[In]:

```
my_list = [4, 3, 8, 2, 9]
```

```
for i in range(len(my_list)):
```

```
    print(my_list[i])
```



This happens until all values of the iterator are looped through.

[Out]:

```
4  
3  
8  
2  
9
```

# Filling Lists and Arrays

- We can use for-loops as a way to fill lists and arrays.

```
numbers = [1, 2, 3, 4, 5, 6, 7]
squares = []

for n in numbers:
    squares.append(n**2)

print(squares)
```

# Filling Lists and Arrays

- This is similar to:

```
numbers = [1, 2, 3, 4, 5, 6, 7]
squares = np.zeros(len(numbers))

for n in range(len(numbers)):
    squares[n] = numbers[n]**2

print(squares)
```



# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])        # This will print every element
```

	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<b>0</b>	1	2	3	4
<b>1</b>	5	6	7	8
<b>2</b>	9	10	11	12
<b>3</b>	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])        # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16



# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16



# Nested for-Loops

- To loop through all elements of a matrix, put a for-loop inside another for-loop:

```
print(matrix.shape)           # This will return (6, 4)
for i in range(matrix.shape[0]): # The same as range(6)
    for j in range(matrix.shape[1]): # The same as range(4)
        print(matrix[i, j])       # This will print every element
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

# if-Statements

- You can control the flow of your program with `if`-statements. Code inside an `if`-statement will run only if the given condition evaluates to **True**.

```
x = 1
y = 2

if x < y:
    print("x is less than y.")
```

# if-Statements

- You can also specify what to do if the condition is not met. You can define multiple options for different cases.

```
x = 1
y = 2

if x < y:
    print("x is less than y.")

elif x > y:
    print("x is greater than y.")

else:
    print("x is neither less than nor greater than y.")
```

# Functions

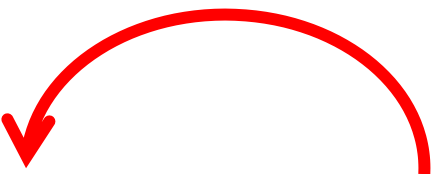
- Functions are used when you want to run the same piece of code on different inputs at different points in time.
- Functions act as wrappers around big algorithms so you can reuse them again and again.

```
def function_name(input_arguments, go, here):  
  
    # Function calculations go here  
  
    return function_output_goes_here  
  
# Call the function and store the output in a variable called result  
  
result = function_name(1, 2, 3)
```

# Functions

- Functions usually take in input and give you output:

result = my\_function(5000)



# Functions

- Functions usually take in input and give you output:

```
result = my_function(5000)
```



# Function Example

- To write a function to calculate force values with  $F = ma$ , you could write:

```
def net_force(mass_in, acc_in):  
    force = mass_in*acc_in  
    return force  
  
# Use the function  
  
my_mass = 10  
my_accelerations = np.linspace(0, 10, 100)  
  
my_forces = net_force(mass_in = my_mass, acc_in = my_accelerations)  
print(my_forces)
```

# Good Coding Practices

- We write code for others to be able to understand, not ourselves. Though your future self can count as an other.
- Use clear and descriptive variable names.
- Break up complicated calculations into multiple steps.
- Comment your code as much as possible.
- Follow the standard formatting conventions from the PEP 8 style guide. There are Python formatters out there that will reformat your code for you.





# Data Analysis Basics

# Importing Data from Files

- You can import data from Excel or CSV files using Pandas:

```
import pandas as pd

Excel_data = pd.read_excel("file.xlsx")
csv_data    = pd.read_csv("file.csv")
```

- This loads the data into a dataframe, which is similar to a dictionary. Index the dataframe and convert the results to arrays:

```
print (csv_data.columns)

time      = np.array(csv_data["time (s)"])
position  = np.array(csv_data["position (cm)"])
velocity  = np.array(csv_data["velocity (cm/s)"])
```

# Making Plots

- The most common plotting library is Matplotlib.

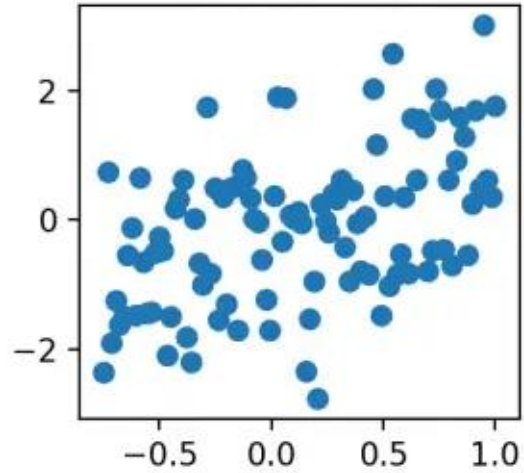
```
import matplotlib.pyplot as plt

plt.plot(x, y)           # Line plots for fits
plt.scatter(x, y)       # Scatter plots for data
plt.errorbar(x, y, yerr) # Line or scatter plots with error bars
plt.hist(x)             # Histogram for displaying counts of variables
```

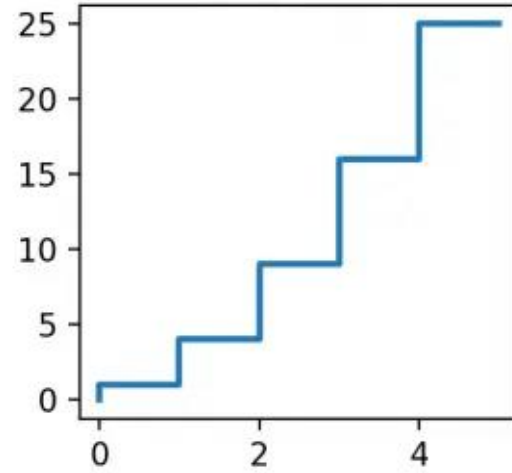
- Explore the different functions and arguments for styling your plots with Matplotlib. You should also look up the Matplotlib documentation to find other useful functions and plot types.

# Types of Plots

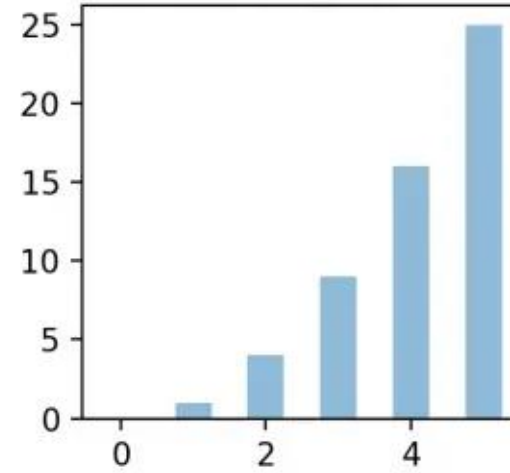
Scatter Plot



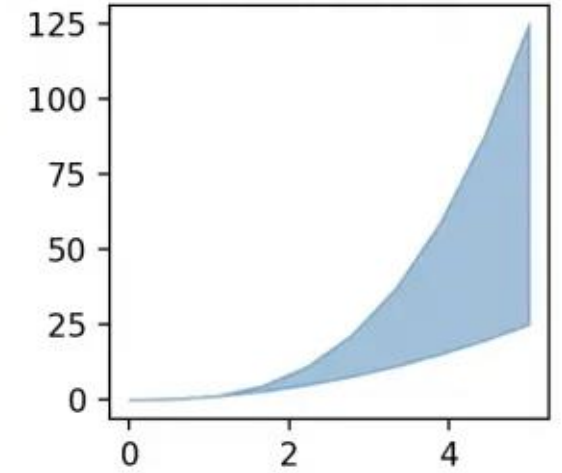
Step Plot



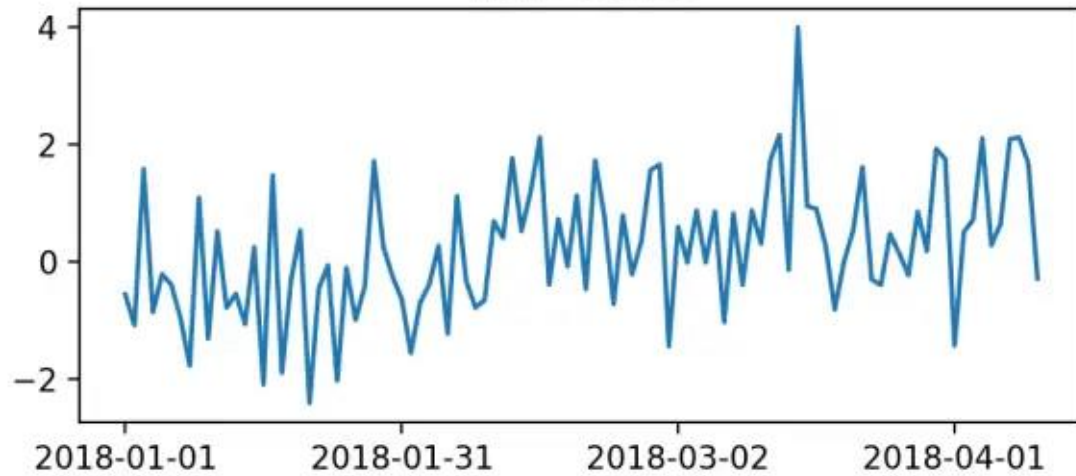
Bar Chart



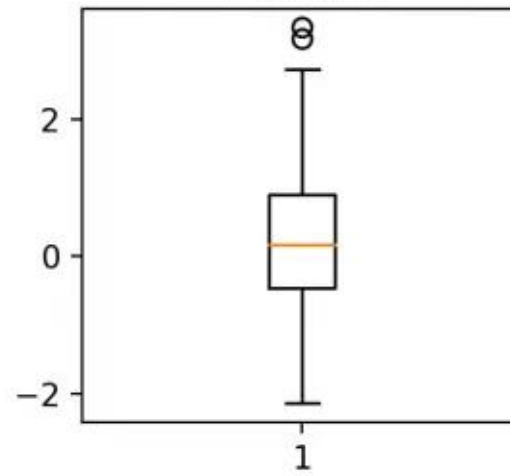
Fill Between



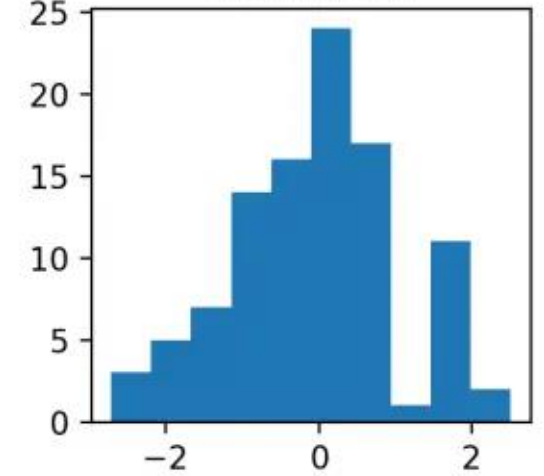
Time Series



Box Plot



Histogram



# Other Useful Plotting Commands

```
plt.figure(figsize = (16, 12)) # Set the size of your canvas

# The main plotting command
# You can use abbreviations for some of the keyword arguments

plt.plot(x, y, color = "blue", lw = 2, ls = "", marker = "D", ms = 12, label =
"Text to go on legend")

plt.title("Title")           # Give your plot a title (do NOT do this)
plt.xlabel(r"$x$")           # Label your x- (or y-) axis with formatted math
plt.legend(loc = "best")    # Draw the Legend
plt.savefig("plot.png")     # Save as a PNG, JPG, PDF (before plt.show())
plt.show()                  # Show your plot
```

# Fitting Polynomial Data

- The `np.polyfit()` function can be used to fit polynomials of the form:

$$F(x) = a_n x^n + \dots + a_2 x^2 + a_1 x + a_0$$

- The  $a_i$  are fit parameters, which are returned by the `polyfit` function.

```
import numpy as np

fitpar = np.polyfit(x_data, y_data, deg = 4)
fitpar = np.polyfit(x, y, 4)

# To include uncertainties:

fitpar, uncer = np.polyfit(x_data, y_data, order, cov = True)
```

# Fitting Polynomial Data

- The uncertainties come from the diagonals of the covariance matrix:

$$\text{cov} = \begin{bmatrix} \text{cov}(a_0, a_0) & \cdots & \text{cov}(a_0, a_N) \\ \vdots & \ddots & \vdots \\ \text{cov}(a_N, a_0) & \cdots & \text{cov}(a_N, a_N) \end{bmatrix}$$

- The covariance gives a measure of how much one parameter changes as another parameter changes.

$$\text{cov}(a, b) = \frac{\sum_{i=0}^N (a_i - \bar{a})(b_i - \bar{b})}{N - 1}$$

# Importing and Plotting Histogram Data

- Histograms are very useful in particle physics. Let's try plotting one. First, load the data from the CSV file.

```
import pandas as pd

hist_data = pd.read_csv("Histogram.csv")
hist_y     = np.array(hist_data["Counts"])
```

- Now let's plot it.

```
import matplotlib.pyplot as plt

counts, bins, _ = plt.hist(hist_y, bins = 100)
```



# Fitting the Histogram Data

- The `scipy.optimize.curve_fit()` function lets you fit to a function of any form. Let's try fitting our CSV data to a Gaussian.

$$f(x) = Ae^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

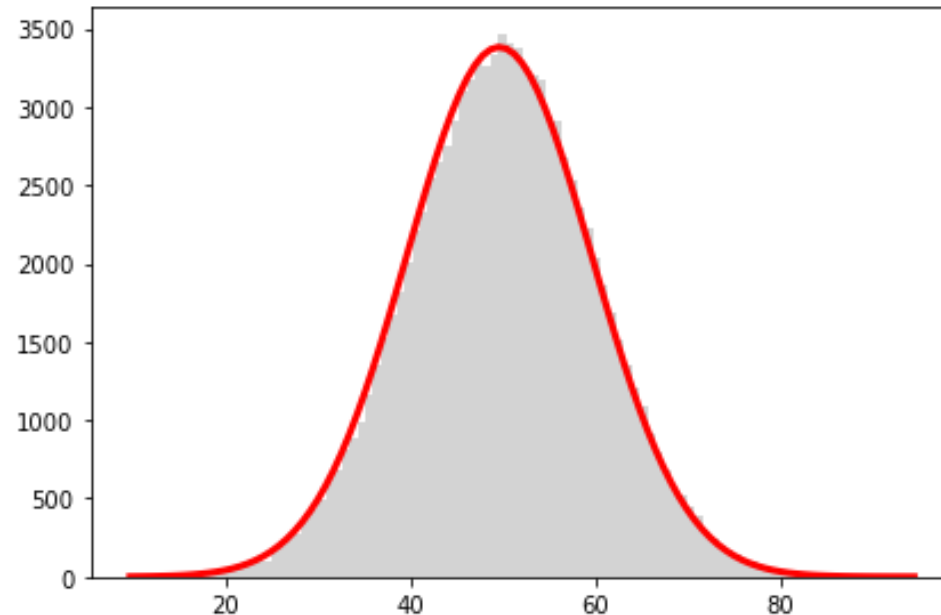
```
def gauss_fit(x, A, mu, sigma):  
    return A*np.exp(-0.5*((x - mu)/sigma)**2)  
  
# We can get our xdata from the plt.hist() function  
  
guess = [1, 45, 100]  
  
A, mu, sigma = scio.curve_fit(gauss_fit, bins[:-1], counts, p0 = guess)[0]
```

# Plotting the Results of the Fit

- We can pass the fit parameters the `curve_fit()` function gave us back into the Gaussian function we wrote.

```
fit_y = gauss_fit(bins, A, mu, sigma)

plt.plot(bins, fit_y, color = "red", lw = 3)
```





**More Information**

# There is a Lot More to Learn

- While-loops
- Classes
- List comprehensions
- Lambda functions
- Keyword arguments
- Try-catch statements and exception handling
- Generators
- Decorators
- Namespaces and scope
- Global, local, and protected variables
- Packages and modules
- Parameters, arguments, methods, attributes, properties

# Useful Resources

- [StackOverflow](#): Answers to almost every question you could imagine.
- [Codecademy](#): Beginner Python tutorials.
- [Real Python](#): Comprehensive beginner Python tutorials.
- [W3Schools](#): Convenient lists of Python functions and concepts.
- [Python Tutor](#): Visualise Python code execution.

# Useful Tools

- **IDE:** Integrated development environment; graphical user interfaces for advanced programming. **Ex:** VS Code, GitKraken, Atom, etc.
- **Linters:** A code analysis tool that can help find syntax errors and lines of code that look suspicious. Linters can help find bugs before you run your program.
- **Debugger:** Helps you find bugs after you run your program.
- **Profiler:** Runs through your code and times how long each part takes to execute. This finds bottlenecks in your algorithms to help you figure out how to optimise your code. **Ex:** cProfile.
- **Formatter:** Formats your code to be more readable and in accordance with standard conventions (like PEP 8). **Ex:** black.

# Other Concepts to Learn About

- **Simulations:** A program that aims to recreate the physical processes of reality. We compare simulation results with experimental measurements.
- **Monte Carlo:** A specific type of simulation based on simulating huge numbers of random processes to approximate reality.
- **Parallel Computing:** Opposed to serial computing, parallel computing is using multiple cores of your CPU to do series of calculations in parallel. This is necessary when running highly intensive simulations or working with huge amounts of data.
- **Machine Learning:** A branch of AI that uses neural networks to give computers the ability to learn from data and make predictions.
- **GPUs:** Graphical processing units for fast computation.

# Essential Skills for Many Careers

- Programming and data analysis are essential skills for doing physics in the modern world, no matter what field you go into.
- It can be difficult to learn, but it is worth it!
- It can open up a lot of job opportunities in the future:
  - Physicist
  - Research scientist
  - Data analyst
  - Data architect
  - Systems engineer
  - Software developer
  - Video game developer
  - Web developer
  - And many more!