

Python Basics

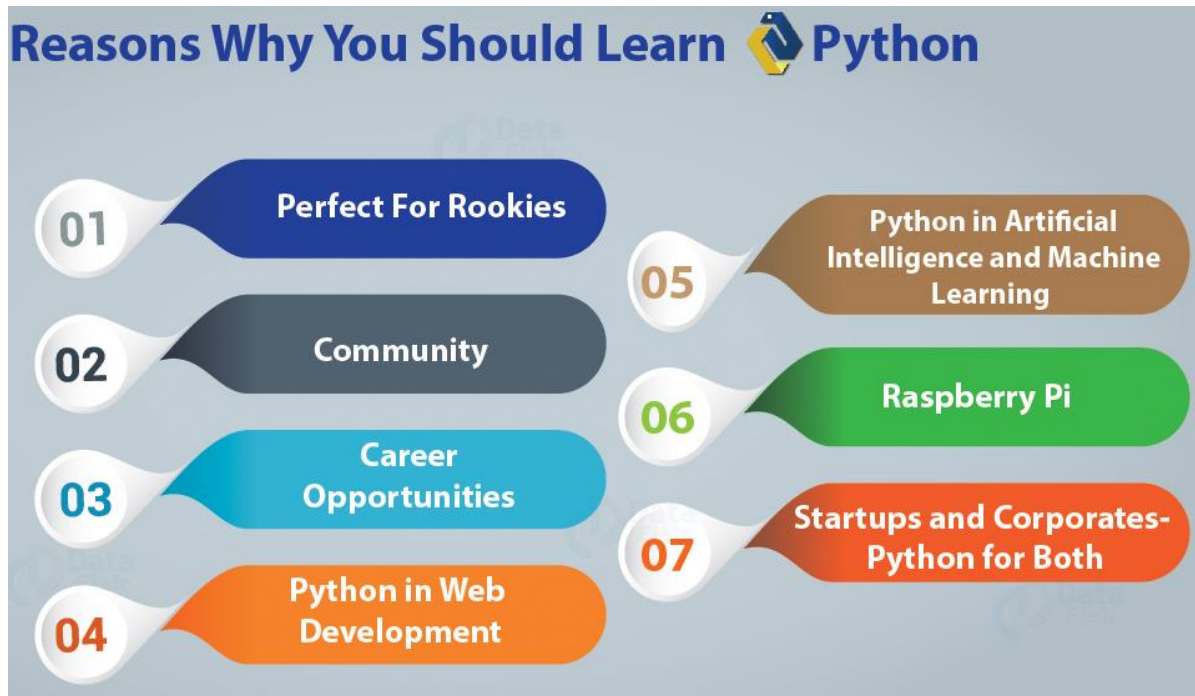
- General information of Python
- Core Python
- Functions and modules
- Mathematics modules
- numpy module overview

Reference:

- Numerical Methods in Engineering with Python 3, Chapter 1.1-1.5
- Python for Everybody: Exploring Data in Python 3

Acknowledgement: Un-copyrighted photos and pictures were taken from the Internet

Why Python?



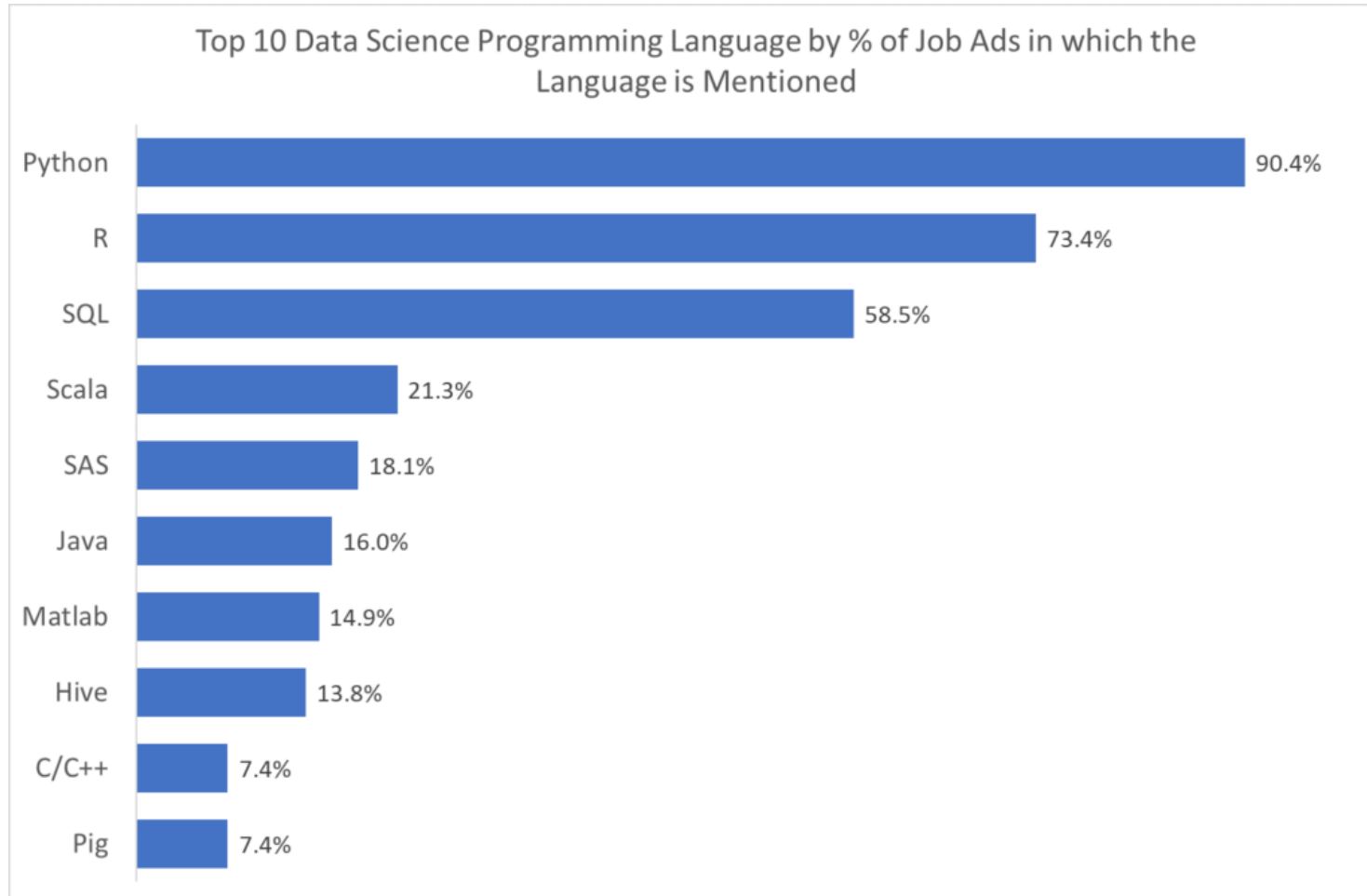
- Easy to program for beginners compared with C/C++.
- A larger community working with Python.
- Many open-source packages and codes are freely available.

Why Python?



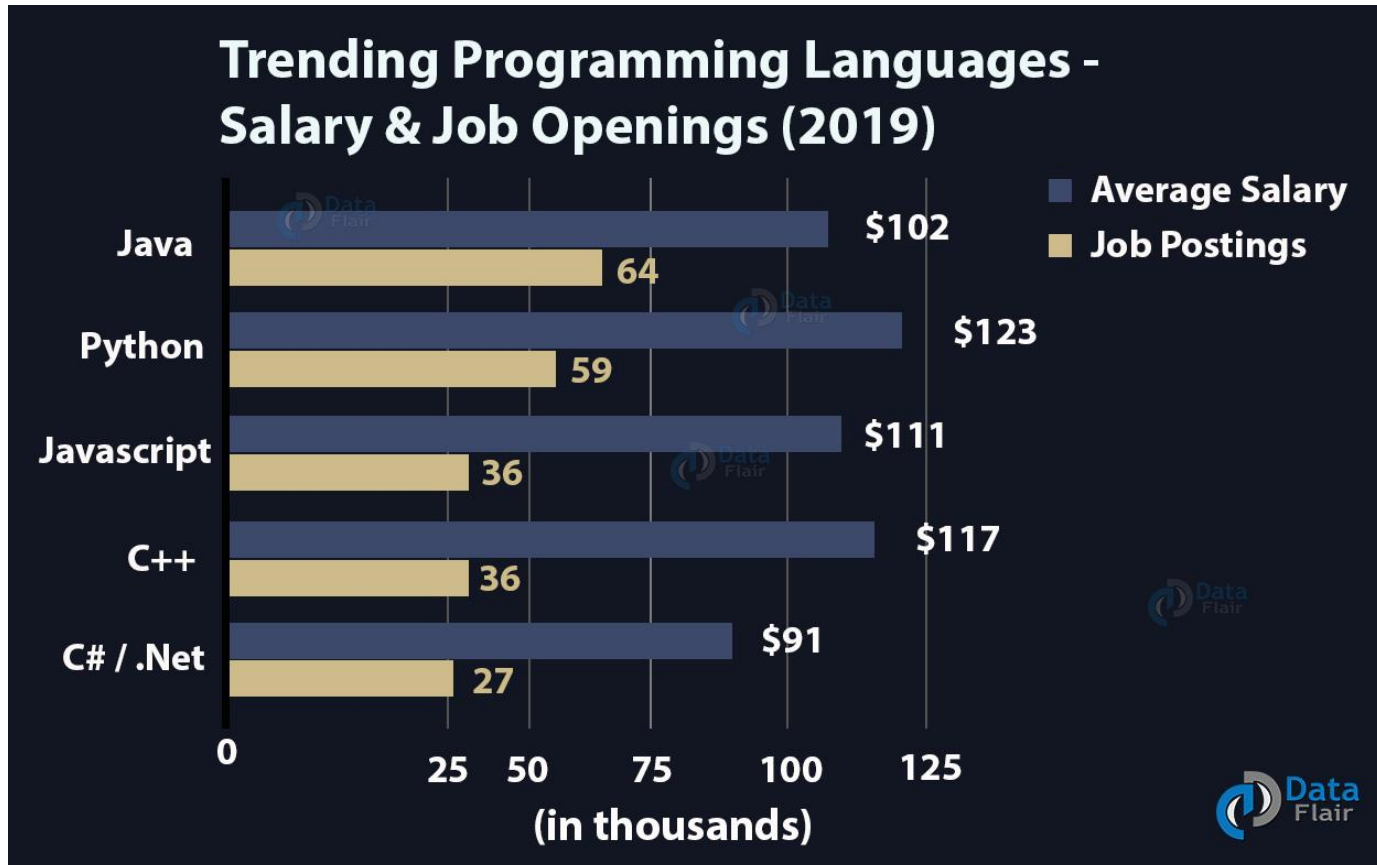
- Giant companies (Apple, Google, Amazon, etc.) and start-ups each year hire many employers with Python skills.

Why Python?



- Numerous career opportunities.

Why Python?



- High-income levels for Python programmers.

Why Python?

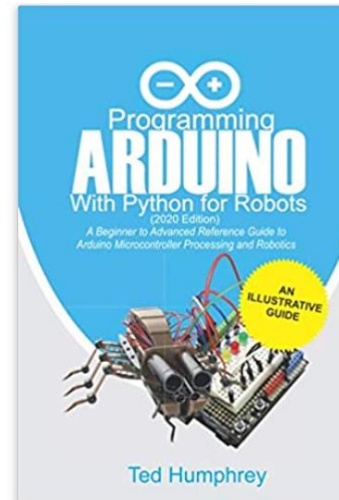
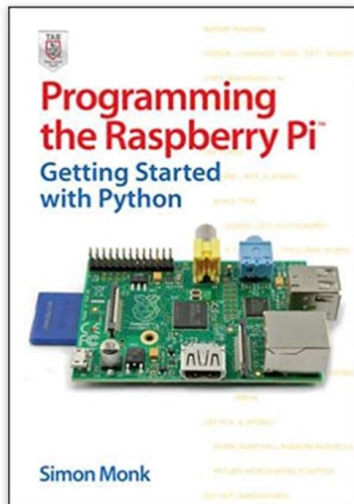
- You can design and implement electronic systems using Python.



Raspberry Pi



Arduino



Why Python?

- Raspberry Pi projects



Drone



Radio transmitters

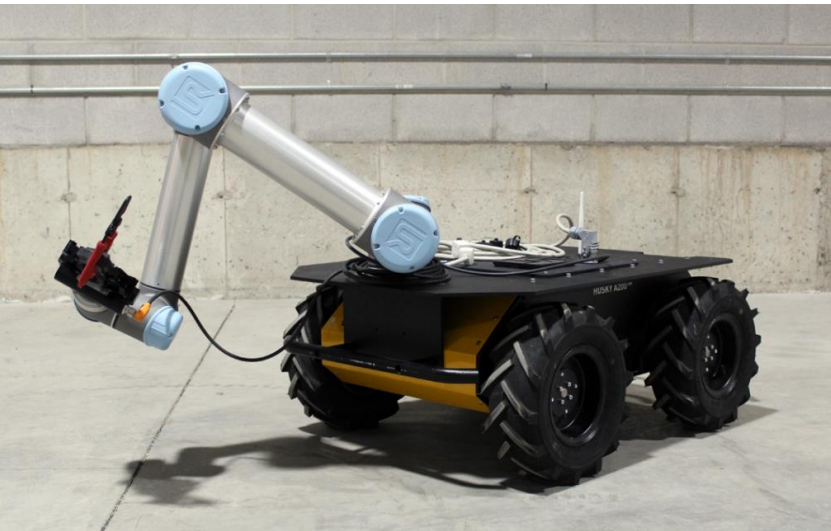


Robot

<https://all3dp.com/1/best-raspberry-pi-projects/>

Why Python?

- **ROS** is short for the robotic operating system.
- **ROS** is a programming environment that allows you to **design, simulate and control robots**.
- You can create **ROS programs using Python**.



- Many advanced **machine learning** and data science algorithms are developed in Python.
- Example: **object detection** for **automated driving**.



Python

- In this course, you will have a chance to **sharpen your Python** skills.
- You will **implement** the **numerical algorithms in Python**.

In the future, you can **use your Python skills** to

- Learn ROS
- Learn TensorFlow
- Program hardware
- Implement machine learning algorithms, etc.

Python: Installation and Description

- To work with Python, you can install **Anaconda** on your computer.
- Anaconda is a **package manager and environment builder**.
- Anaconda installer for different platforms is available here:
<https://www.anaconda.com/products/individual>

To write **Python codes** using **Anaconda**:

1. Open the Anaconda program and launch the Jupyter notebook.
 2. Click on “**new**”, then **Python 3**.
- **Jupyter** is an **integrated development environment** for Python.

Python: Installation and Description

You can also use your web browser to write Python code.

1. Go to this webpage: <https://jupyter.org/try>
2. Select “Try JupyterLab”.

Python Core

Constants

- **Constants** in Python have **fixed values**, e.g., numbers, letters, and strings of alphabets.
- They are called “**constants**” because their values do not change.

Numeric Constants	<pre>print(12.3)</pre> 12.3
String Constants Use single quotes (') or double quotes (")	<pre>print('hello world')</pre> hello world

Variables

- The value of **variables** can be changed.

```
In [5]: b = 3 #b is integer type  
print(b)
```

3

```
In [6]: b = b*3.0 # now b is float type  
print(b)
```

9.0

Variable name rules:

- The name of a variable can start with a letter.
- Avoid** using an **underscore** as it is intended for internal variables.
- Can consist of **letters, numbers and/or underscores**.
- Case sensitive**.

Numeric Operators

Numeric expression

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Augmented assignment operators

<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a*b</code>
<code>a /= b</code>	<code>a = a/b</code>
<code>a **= b</code>	<code>a = a**b</code>
<code>a %= b</code>	<code>a = a%b</code>

Operator Precedence Rules

Which operator “takes precedence” over the others?

`x = 1 + 2 ** 3`

- If we first add 1 to 2 and then raise the result to the power of 3, we get 27.
- If we first raise 2 to the power of 3 and then add 1 to the result, we get 9.
- Which operation is performed first in Python?

Operator Precedence Rules

Which operator “takes precedence” over the others?

$$x = 1 + 2 ** 3 / 4 * 5 + 6$$

Highest precedence rule to lowest precedence rule:

- **Parentheses**
- Exponentiation (raise to a power)
- Multiplication, division, and remainder
- Addition and subtraction
- Left to right



Sentences or Lines

x = 3



Assignment statement

x = x + 3



Assignment with expression

print(x)



Print statement

Variable

Operator

Constant

Function

Lists of constants

A **list** is a **collection of elements**, e.g., numbers, strings, etc.

List of Constants

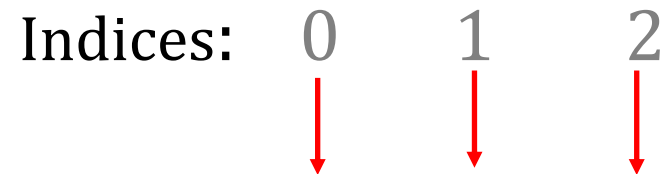
	Examples	Outputs
List constants are surrounded by square brackets . & Elements in the list are separated by commas .	<code>print([1, 3, 5])</code>	<code>[1, 3, 5]</code>
	<code>print(['red', 'white', 'blue'])</code>	<code>['red', 'white', 'blue']</code>
	<code>print(['red', 3, 99.9])</code>	<code>['red', 3, 99.9]</code>
A list element can be another list .	<code>print([1, [7, 9], 5])</code>	<code>[1, [7, 9], 5]</code>
A list can be empty.	<code>print([])</code>	<code>[]</code>

List index

- We can **assign** an **index** (**number**) to **each element** of a **list**.
- Using the **index**, we can get the **value of any single element** in a list using its index.
- The **index of the first element** is 0.
- To refer to an element in a list, use the following command:

“Listname[element index]”

Indices: 0 1 2



```
fruits = ['apple', 'orange', 'banana']  
print(fruits[1])
```

orange

List index

Examples of list operations

1) Create a list	<code>a = [1.0, 2.0, 3.0]</code>	<code>[1.0, 2.0, 3.0]</code>
2) Append 4.0 to the end	<code>a.append(4.0)</code>	<code>[1.0, 2.0, 3.0, 4.0]</code>
3) Insert 0.0 in position 0	<code>a.insert(0, 0.0)</code>	<code>[0.0, 1.0, 2.0, 3.0, 4.0]</code>

`a.insert(0, 0.0)`

Position of
new element

New element

Lists

- The **len()** function: Returns the number of elements of any list or sequence.

```
greeting='Hello World'  
print(len(greeting))
```

11

```
x = [1, 2, 'CityU', 99]  
print(len(x))
```

4

Lists

- The command `b = a` does not result in a new object `b`.
- It simply creates a new reference to `a` (an `alias`).
- Any changes made to `b` will be reflected in `a`
- To create an `independent copy` of a list `a`, use the statement `c = a[:]`.

```
a = [1.0, 2.0, 3.0]
b=a
a[0]=6
print(a)
print(b)
```

```
[6, 2.0, 3.0]
[6, 2.0, 3.0]
```

```
a = [1.0, 2.0, 3.0]
b=a[:]
a[0]=6
print(a)
print(b)
```

```
[6, 2.0, 3.0]
[1.0, 2.0, 3.0]
```

Lists

- Matrices are represented as nested lists with each row being an element of the list.

```
a=[[1.0, 2.0, 3.0],\
   [2.0, 3.0, 6.0],\
   [6.0, 8.0, 7.0]]
print(a)
print(a[0])
print(a[0][1])
```

← Creates 3-by-3 matrix and name it "a"

← Prints the first row

← Prints the element in the first row and second column

```
[[1.0, 2.0, 3.0], [2.0, 3.0, 6.0], [6.0, 8.0, 7.0]]
[1.0, 2.0, 3.0]
2.0
```

- Normally, do not use lists for numerical arrays
- More convenient to employ *array objects* provided by the numpy module

For Loops

- “**for**” loops are a convenient way for repeating certain calculations for a given number of iterations.
- Suppose that we would like to create a list that contains $1, 1/2, 1/3$, and $1/4$.
- One solution is to create the list manually.

```
a=[1,1/2,1/3,1/4]  
print(a)
```

```
[1, 0.5, 0.3333333333333333, 0.25]
```

- This can be tedious. Suppose that you need to create a list that contains $1, 1/2, 1/3, \dots, 1/1000$.

For Loops

- Suppose that we would like to create a **list** that contains **1, 1/2, 1/3, and 1/4**.

```
nMax = 5
a = []
for n in range(1, nMax):
    a.append(1.0/n)
print(a)
```

Diagram annotations:

- start**: points to the first argument `1` in `range(1, nMax)`.
- stop**: points to the second argument `nMax` in `range(1, nMax)`.
- Loop variable**: points to the variable `n` in `for n in range(1, nMax):`.
- For loop**: points to the entire `for` loop block.
- Calculation**: points to the expression `1.0/n` inside the `append` method.

Output: `[1.0, 0.5, 0.3333333333333333, 0.25]`

- Each “**for**” loop has a **variable** that **changes** in a **range**.
- We can **specify** the **values of the loop variable** using the **range** command.
- range(n,m)**: it returns sequence of integers from **n** to **(m-1)**.
- range(1,5)** returns **1,2,3, and 4**.

For Loops

- Python changes the loop variable in the **specified range**.
- It performs the **desired computation** for **each value of the variable**.
- Remember to **add “:”** after the **range arguments**.
- Remember to **indent** the **computations performed** by the **for loops**.

Indent

```
nMax = 5
a = []
for n in range(1, nMax):
    a.append(1.0/n)
print(a)
```

[1.0, 0.5, 0.3333333333333333, 0.25]

n= 1 2 3 4

 [1.0, 0.5, 0.3333333333333333, 0.25]

Conditionals

- Sometimes, we need to **perform computations when a certain condition holds**.
- Suppose that you need to create a list that contains $1, \frac{1}{3}, \frac{1}{5}, \dots, \frac{1}{L}$ where L is an odd number.
- For example, $1, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \frac{1}{9}, \frac{1}{11}$ (when $L = 11$).
- What if we use a “for” loop with a **variable n** to compute this list?

```
nMax=12
a=[]
for n in range(1,nMax):
    a.append(1/n)

print(a)
```

```
[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.16666666666666666,
 0.14285714285714285, 0.125, 0.11111111111111111, 0.1, 0.09090909090909091]
```

Conditionals

- What if we use a “for” loop with a **variable n** to compute this list?

```
nMax=12
a=[]
for n in range(1,nMax):
    a.append(1/n)

print(a)
```

```
[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.16666666666666666,
_ 0.14285714285714285, 0.125, 0.11111111111111111, 0.1, 0.090909090 90909091]
```

- But there is a **condition**. We need to **perform the computations when n is odd**.

How to implement conditions in Python?

Conditionals

- We can use the **if** command to implement conditions.

if condition:
Indent → **Calculations**

- When the **condition holds**, Python will perform the **calculations**.
- Calculations** must be **indented**.

Comparison
operators



Python	Meaning
<	Less than
<=	Less than or Equal to
==	Equal to
>=	Greater than or Equal to
>	Greater than
!=	Not equal

Conditionals

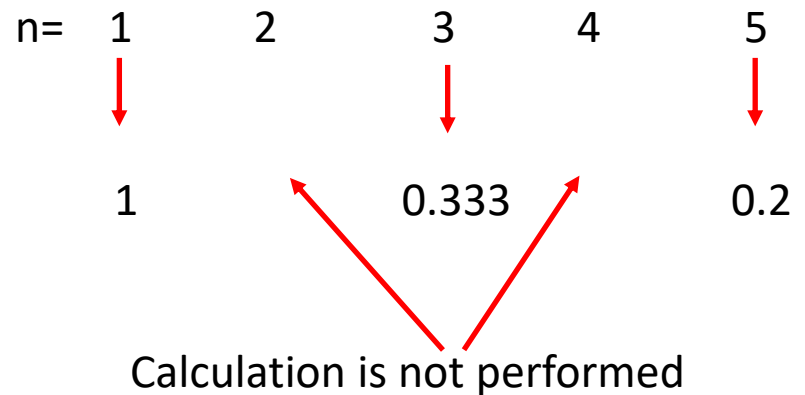
- Suppose that you need to create a list that contains $1, \frac{1}{3}, \frac{1}{5}, \dots, \frac{1}{L}$ where L is an odd number.
- For example, $1, \frac{1}{3}, \frac{1}{5}$ when $L=5$.
- The condition “ **$n\%2!=0$** ” means **when the remainder of n divided by 2 is not zero**. That is, **when n is odd**.

```
nMax=6
a=[]
for n in range(1,nMax):
    if n%2!=0:
        a.append(1/n)
print(a)
```

Condition

Indent

```
[1.0, 0.3333333333333333, 0.2]
```



Mathematical Functions

Mathematical Functions

- Core Python supports only the following mathematical functions:

<code>abs(a)</code>	Absolute value of <i>a</i>
<code>max(sequence)</code>	Largest element of <i>sequence</i>
<code>min(sequence)</code>	Smallest element of <i>sequence</i>
<code>round(a,n)</code>	Round <i>a</i> to <i>n</i> decimal places

- How to compute $\sin(x)$ or $\log(x)$ function?

Module

- The majority of **mathematical functions** are available in the **math module**.
- A **module** is a file **where** the functions **reside**. To **load a module** into the program

```
from module_name import *
```

- This command will **load the whole module**.

math Module

math Module

Statement	Purpose	Example
<code>from math import *</code>	Load all function definitions in the <code>math</code> module	AVOID!! May have module conflicts
<code>from math import func1, func2</code>	Load selected functions	<code>from math import log, sin</code> <code>print(log(sin(0.5)))</code>
<code>import math</code>	Make module accessible and then accessing the definitions in the module by using the module	<code>import math</code> <code>print(math.log(math.sin(0.5)))</code>
<code>Import math as m</code>	Make module accessible under an <i>alias</i>	<code>import math as m</code> <code>print(m.log(m.sin(0.5)))</code>

math Module

- Use the command “`print(dir(math))`” to see the content of the math module.

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

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysig
n', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'f
actorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

- Note that the module includes two constants: π and e

cmath Modules

- The **cmath** module contains functions for **complex numbers**.

```
import cmath
print(dir(cmath))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',  
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp',  
'inf', 'infj', 'isclose', 'isfinite', 'isinf', 'isnan', 'log', 'log10',  
'nan', 'nanj', 'phase', 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan',  
'tanh', 'tau']
```

- Example:

```
from cmath import sin
x = 2.25 - 4.5j
y = 1.6 + 0.8j
z = 0.5
print(x/y)
print(sin(x))
print(sin(z))
```

```
-2.8125j  
(35.024280369063746+28.26970454877851j)  
(0.479425538604203+0j)
```

numpy Module

- The **numpy** Modules supports **matrix computations**.

import numpy as np

- You can use the **array command** to **create matrices**.

Rows of the matrix Type of elements

array (*List, type*)

- Example:

```
from numpy import array
a = array([[1.0, -2.0], [-3.0, 4.0]],int)
print(a)
```

```
[[ 1 -2]
 [-3  4]]
```

type: integer

```
b = array([[1.0, -2.0], [-3.0, 4.0]],float)
print(b)
```

```
[[ 1. -2.]
 [-3.  4.]]
```

type: floating-point

numpy Module

Other functions to create specific arrays

Statement	Purpose
<code>zeros((dim1,dim2),type)</code>	Create a dim1 x dim2 array and fill it with zeroes
<code>ones((dim1,dim2),type)</code>	Fill the array with ones

Example:

```
from numpy import *  
print(zeros((1,4),int))  
print(ones((2,4),int))
```

```
[[0 0 0 0]]  
[[1 1 1 1]  
 [1 1 1 1]]
```

numpy Module

- `a[i,j]` provides access to the **element** in row **i+1** and column **j+1**.
- `a[i]` provides access to row **i+1**.

Example:

```
from numpy import *  
a=zeros((3,3),int)  
print(a)
```

```
[[0 0 0]  
 [0 0 0]  
 [0 0 0]]
```

```
a[0]=[1,2,3]
```

Changes all the elements in the first row

```
a[2,2]=6
```

Changes the element in the third row and third column

```
a[2,0:2]=[14,5]
```

```
print(a)
```

```
[[ 1  2  3]  
 [ 0  0  0]  
 [14  5  6]]
```

Changes the elements in the first and second elements in the third row

numpy Module

- In **numpy** Module, **certain operations** are applied to **every element** of an **array**.

- Example:

```
from numpy import array, sqrt, sin
a=array([1,2,3],float)
print(a)
print(a+4)
print(sin(a))
print(sqrt(a))
```

```
[1.  2.  3.]
[5.  6.  7.]
[0.84147098  0.90929743  0.14112001]
[1.          1.41421356  1.73205081]
```

- This is not the case in the **mathModule**.

Linear Algebra Modules

- The numpy module comes with a linear algebra module.
- It can perform **matrix inversion**, **solve systems of linear equations**, etc.

```
from numpy import array, transpose
from numpy.linalg import inv, solve
a = array([[4.0, 7.0], \
           [2.0, 6.0]])
print(a)
print(inv(a)) # Matrix inverse
```

```
[[4. 7.]
 [2. 6.]]
[[ 0.6 -0.7]
 [-0.2  0.4]]
```

```
b = transpose(array([[1.0, 4.0]]))
print(b)
```

```
[[1.]
 [4.]]
```

```
print(solve(a,b)) # Solve a x = b
```

```
[[ -2.2]
 [  1.4]]
```

numpy Module

- Example

```
from numpy import *  
a = array([[1,-2,3],[-4,5,-6],[7,-8,9]],float)  
b = array([3,2,1],float)  
print(a)  
print(b)
```

```
[[ 1. -2.  3.]  
 [-4.  5. -6.]  
 [ 7. -8.  9.]]  
[3. 2. 1.]
```

```
print(diagonal(a)) # Principal diagonal
```

```
[1. 5. 9.]
```

```
print(diagonal(a,1)) # 1st subdiagonal above the main diagonal
```

```
[-2. -6.]
```

```
print(trace(a)) # Sum of diagonal elements
```

```
15.0
```

```
print(argmax(b)) # Index of largest element
```

```
0
```

```
print(argmin(a,axis=0)) # Indices of smallest col. elements
```

```
[1 2 1]
```

```
print(identity(3)) # Identity matrix
```

```
[[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```

Copying arrays

- `b = a` does not result in a new object *b*, but simply creates a new reference to *a*, called a *deep copy*.
- To make an independent copy of an array *a*, use the `copy` method in the `numpy` module:

```
b = a.copy()
```

- Similar to list assignment (on page 15):

```
c = a[:]
```

Functions

- Suppose that you would like to write a **Python program** to compute the **derivative** of **$\sin(x)$** at a **$x = 0.5$** .

```
from math import sin, cos, log
h=0.001#step size
x=0.5
df=(sin(x+h)-sin(x-h))/(2*h)
print('The derivative of sin(x) at x=0.5 is', df)
```

The derivative of $\sin(x)$ at $x=0.5$ is 0.8775824156266321

Functions

- Suppose that you need to compute the derivative of $\sin(x)$, $\cos(x)$, $\log(x)$.

```
from math import sin, cos, log
h=0.001#step size
x=0.5
df=(sin(x+h)-sin(x-h))/(2*h)
print('The derivative of sin(x) at x=0.5 is', df)
df=(cos(x+h)-cos(x-h))/(2*h)
print('The derivative of cos(x) at x=0.5 is', df)
df=(log(x+h)-log(x-h))/(2*h)
print('The derivative of log(x) at x=0.5 is', df)
```

```
The derivative of sin(x) at x=0.5 is 0.8775824156266321
The derivative of cos(x) at x=0.5 is -0.4794254586999447
The derivative of log(x) at x=0.5 is 2.000002666673051
```

- If you need to compute the **derivative of many functions**, your **code** will be **very long** because you need to **repeat** the **code** for the **derivative calculations** for **each** function.

Functions

- You can **reduce the size of your code** by writing a **Python function** that **computes the derivative**.
- The **Python function** takes the point x and the **function $f(x)$** as input. It then **computes the derivative of $f(x)$ at x** and **returns the result**.

Inputs are $f(x)$ and x

```
def derivative(f,x):  
    h=0.0001  
    df=(f(x+h)-f(x-h))/(2*h)  
    return df
```

Name of the function

Computes the derivative

Returns the result

Calls the function

Inputs

```
from math import sin, cos, log  
print('The derivative of sin(x) at x=0.5 is', derivative(sin,0.5))  
print('The derivative of cos(x) at x=0.5 is', derivative(cos,0.5))  
print('The derivative of log(x) at x=0.5 is', derivative(log,0.5))
```

The derivative of $\sin(x)$ at $x=0.5$ is 0.8775825604276366
The derivative of $\cos(x)$ at $x=0.5$ is -0.4794255378054446
The derivative of $\log(x)$ at $x=0.5$ is 2.0000000266662266

Functions

- Any time that you **call the function**, you need to **specify its inputs**.
- The function takes the inputs and returns the result.
- Using **functions**, your Python codes become **much shorter and easier** to debug.

```
def derivative(f,x):  
    h=0.0001  
    df=(f(x+h)-f(x-h))/(2*h)  
    return df  
  
from math import sin, cos, log  
print('The derivate of sin(x) at x=0.5 is', derivative(sin,0.5))  
print('The derivate of cos(x) at x=0.5 is', derivative(cos,0.5))  
print('The derivate of log(x) at x=0.5 is', derivative(log,0.5))
```

```
The derivate of sin(x) at x=0.5 is 0.8775825604276366  
The derivate of cos(x) at x=0.5 is -0.4794255378054446  
The derivate of log(x) at x=0.5 is 2.0000000266662266
```

Functions

- You can define function in Python using the following structure

```
def func_name(param1, param2, ...):  
    statements  
    return return_values
```

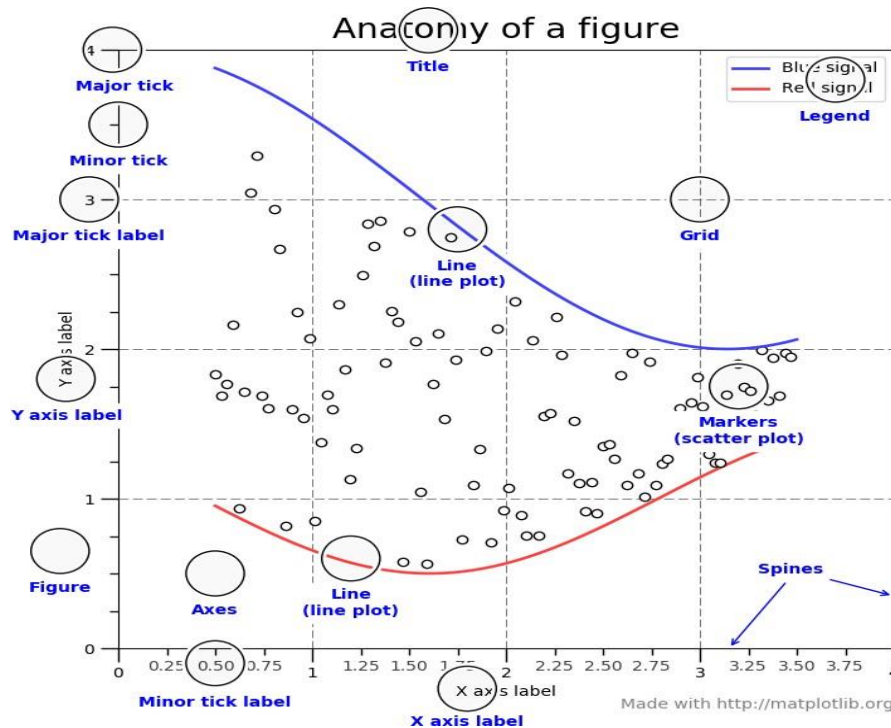
- *param1*, *param2*, are parameters
- A parameter can be any Python objects, including a function

matplotlib

- **matplotlib** is a **powerful module** for creating **plots**. It is **not** in the Python core.

import matplotlib.pyplot as plt

Imports the **matplotlib** functions and name them as **plt**

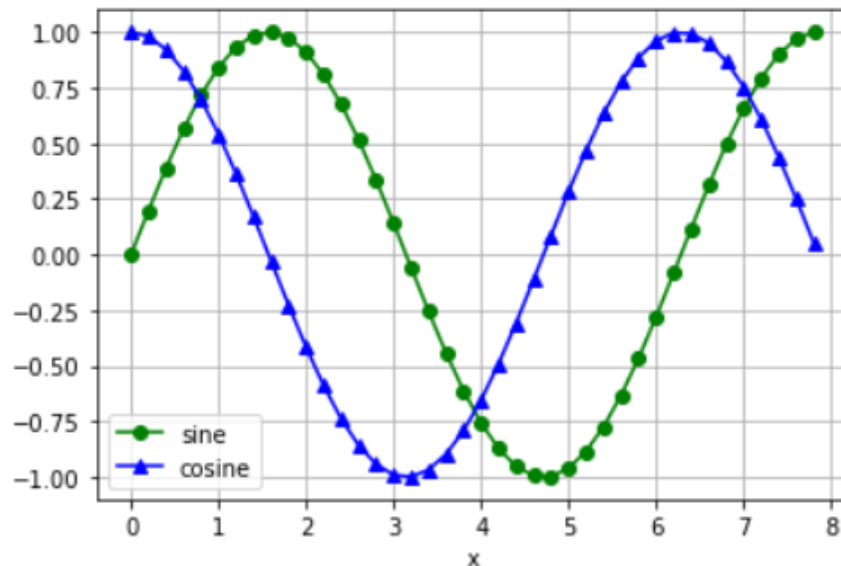


matplotlib

Matplotlib: Line Plot

```
import matplotlib.pyplot as plt
from numpy import arange,sin,cos
x = arange(0.0,8,0.2)
plt.plot(x,sin(x),'go-',x,cos(x),'b^-' ) # Plot with specified line and marker style
#(g==green, b=blue, -=line, o=circle, ^=triangle)
plt.xlabel('x') # Add label to x-axis
plt.legend(('sine','cosine'),loc = 0) # Add Legend in loc. 0
plt.grid(True) # Add coordinate grid
plt.show() # Show plot on screen
```

Creates a list of numbers from 0 to 8 with spacing equal to 0.2



Matplotlib: subplot

- Subplot is used to include **more than one plot** in a **figure**.

```
import matplotlib.pyplot as plt
from numpy import arange, sin, cos
x = arange(0.0, 8, 0.2)
plt.subplot(2,1,1)
plt.plot(x, sin(x), 'go-')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)
plt.subplot(2,1,2)
plt.plot(x, cos(x), 'b^-')
plt.xlabel('x')
plt.ylabel('cos(x)')
plt.grid(True)
plt.show()
```

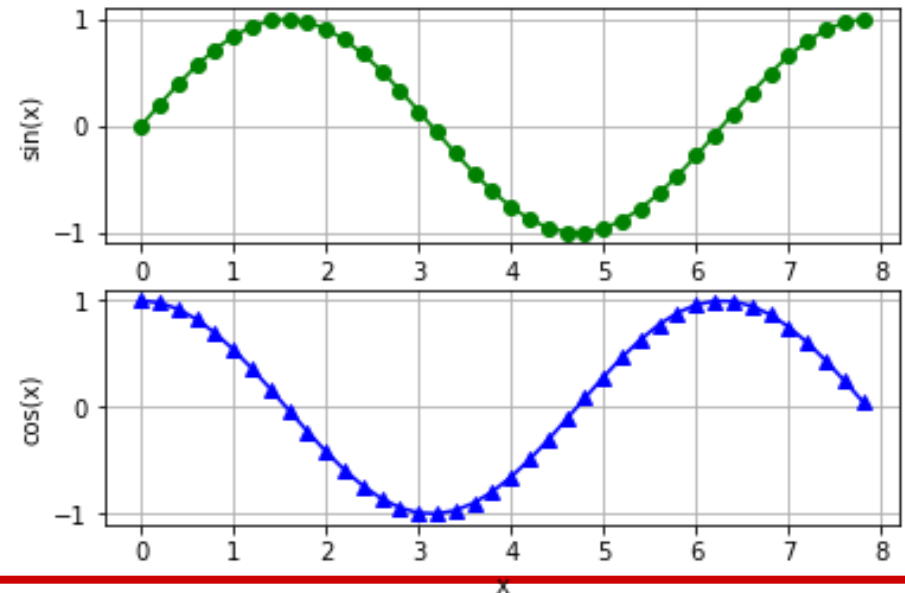
Creates the first
subplot

Creates the second
subplot

`plt.subplot(2,1,1)`

Number of
subplots

`plt.subplot(2,1,2)`

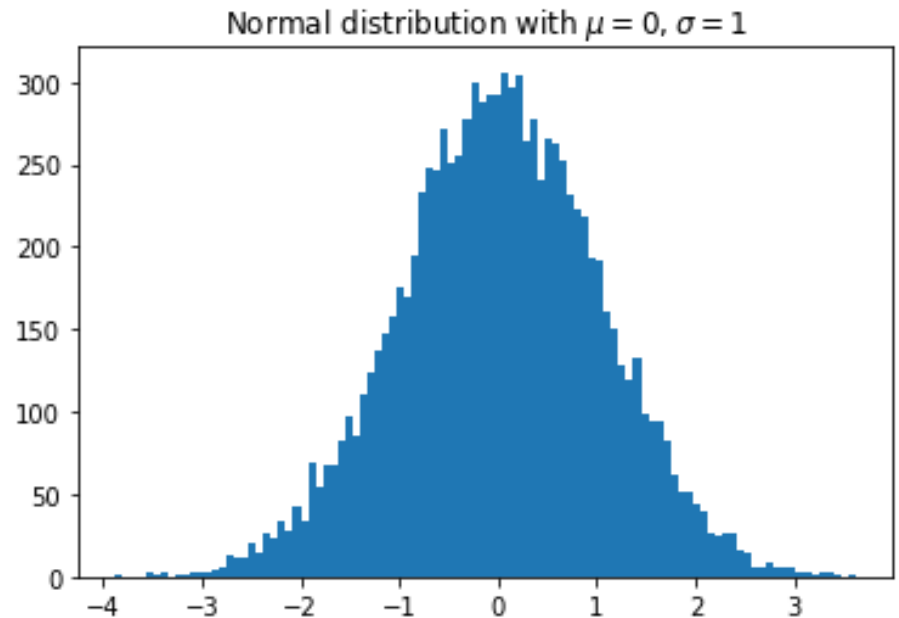


Matplotlib: Histograms

- You can plot the **frequency distribution of your data** in Python.

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

x = np.random.randn(10000) # generates 10000 random numbers
# from a normal distribution with mean 0 and variance 1
plt.hist(x, 100) # 100 bins
plt.title(r'Normal distribution with  $\mu=0$ ,  $\sigma=1$ ')
plt.savefig('matplotlib_histogram.png')
plt.show()
```



END

See you next week 😊!