



Hands-on Tutorial

● History

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991. It is a Interpreter language which runs line by line.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.
- Data Science

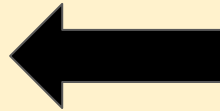
What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.



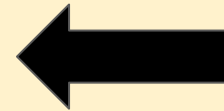
● Installation

1. *winget search python*
2. *winget install (python 3.10 or package id)*
3. *successful installation*



1. *sudo apt update*
2. *sudo apt install python3*
3. *successful Installation*

1. */bin/bash -c "\$(curl -fsSL <https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh>)"*
2. *brew install python*
3. *python3 --version*



● Basic Syntax

Printing Hello World :

Python syntax can be executed by writing directly in the Command Line.

```
>>> print("Hello, World!")  
Hello, World!
```

Or by creating a python file on the server, using the .py file extension, and running it in the Command Line:

```
C:\Users\Your Name>python myfile.py
```

● Comments

Single line Comment

```
#This is a comment  
print("Hello, World!")
```

Multi-line Comment

```
#This is a comment  
#written in  
#more than just one line  
print("Hello, World!")
```

● Variables

Variables are containers for storing data values. Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

Example 1 :

```
x = 5
y = "John"
print(x)
print(y)
```

Example 2 :

```
x = 4          # x is of type int
x = "Sally"    # x is now of type str
print(x)
```

● Casting & Types

Casting : If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)      # x will be '3'
y = int(3)      # y will be 3
z = float(3)    # z will be 3.0
```

Types : You can get the data type of a variable with the `type()` function.

```
x = 5
y = "John"
print(type(x))
print(type(y))
```

● Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alphanumeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

Legal Variable Names:

```
myvar = "John"  
my_var = "John"  
_my_var = "John"  
myVar = "John"  
MYVAR = "John"  
myvar2 = "John"
```


● Assign Multiple Values

Python allows you to assign values to multiple variables in one line:

```
x, y, z = "Orange", "Banana", "Cherry"  
print(x)  
print(y)  
print(z)
```

List Values Assignment

```
fruits = ["apple", "banana", "cherry"]  
x, y, z = fruits  
print(x)  
print(y)  
print(z)
```

● Data Types

In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type: **str**

Numeric Types: **int**, **float**, **complex**

Sequence Types: **list**, **tuple**, **range**

Mapping Type: **dict**

Set Types: **set**

Boolean Type: **bool**, etc.

Example:

```
x = input("Enter a value:");  
print(type(x));
```

● Numbers

There are three numeric types in Python:

- `int`
- `float`
- `complex`

Example: Now, let's take an example for assign all three variables and check the types.

```
x = 1 # int  
y = 2.8 # float  
z = 1j # complex
```

```
print(type(x))  
print(type(y))  
print(type(z))
```

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks. 'hello' is the same as "hello". You can display a string literal with the `print()` function:

```
print("Hello")
```

```
print('Hello')
```

Multiline Strings : You can assign a multiline string to a variable by using three quotes.

```
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)
```

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks. 'hello' is the same as "hello". You can display a string literal with the `print()` function:

```
print("Hello")
```

```
print('Hello')
```

Multiline Strings : You can assign a multiline string to a variable by using three quotes.

```
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)
```

Strings

Strings Are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Example : Get the character at position 1 (remember that the first character has the position 0).

```
a = "Hello, World!"  
print(a[1])
```

Example : Loop through the letters in the word "banana".

```
for x in "banana":  
    print(x)
```

Strings

Strings Length : To get the length of a string, use the `len()` function.

Example:

```
a = "Hello, World!"  
  
print(len(a))
```

Check Stings : To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example:

```
txt = "The best things in life are free!"  
  
print("free" in txt)
```

Strings

Check If NOT : To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example:

```
txt = "The best things in life are free!"
```

```
print("expensive" not in txt)
```

Slicing in Python

You can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.

Example:

```
b = "Hello, World!"
```

```
print(b[2:5])
```


Strings

String Modification: Python has a set of built-in methods that you can use on strings.

1. Uppercase: The `upper()` method returns the string in upper case.

```
a = "Hello, World!"
```

```
print(a.upper())
```

2. Lowercase: The `lower()` method returns the string in lower case.

```
a = "Hello, World!"
```

```
print(a.lower())
```

3. Strip: The `strip()` method removes any whitespace from the beginning or the end.

```
a = " Hello, World! "
```

```
print(a.strip()) # returns "Hello, World!"
```

Strings

String Modification: Python has a set of built-in methods that you can use on strings.

1. Replace: The `replace()` method replaces a string with another string.

```
a = "Hello, World!"
```

```
print(a.replace("H", "J"))
```

2. Split: The `split()` method returns a list where the text between the specified separator becomes the list items.

```
a = "Hello, World!"
```

```
print(a.split(","))
```

Strings

String Concatenation: To concatenate, or combine, two strings you can use the + operator.

```
a = "Hello"
```

```
b = "World"
```

```
c = a + b
```

```
print(c)
```

To add a space between them, add a " ":

```
a = "Hello"
```

```
b = "World"
```

```
c = a + " " + b
```

```
print(c)
```

Strings

String Formatting: As we learned in the Python Variables chapter, we cannot combine strings and numbers like this.

Example:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

F-Strings: F-String was introduced in Python 3.6, and is now the preferred way of formatting strings. To specify a string as an f-string, simply put an `f` in front of the string literal, and add curly brackets `{ }` as placeholders for variables and other operations.

Example:

```
age = 36
txt = f"My name is John, I am {age}"
print(txt)
```

Strings

String Methods:

`capitalize()`: Converts the first character to uppercase.

`casefold()`: Converts string into lowercase.

`count()`: Returns the number of times a specified value occurs in a string.

`find()`: Searches the string for a specified value and returns the position of where it was found.

`index()`: Searches the string for a specified value and returns the position of where it was found.

`isalnum()`: Returns True if all characters in the string are alphanumeric.

`upper()`: Converts a string into upper case.

`lower()`: Converts a string into lower case.

`split()`: Splits the string at the specified separator, and returns a list.

`strip()`: Returns a trimmed version of the string.

`replace()`: Returns a string where a specified value is replaced with a specified value.

`join()`: Joins the elements of an iterable to the end of the string.

`swapcase()`: Swaps cases, lower case becomes upper case and vice versa.

Booleans

In python programming you often need to know if an expression is **True** or **False**. You can evaluate any expression in Python, and get one of two answers, **True** or **False**. The **bool()** function allows you to evaluate any value, and give you **True** or **False** in return.

```
print(10 > 9);
```

```
print(10 == 9);
```

```
print(10 < 9);
```

```
print(bool("Hello"));
```

```
print(bool(15));
```

```
print((bool(15<9)));
```

Operators

Operators are used to perform operations on variables and values. In the example below, we use the **+** operator to add together two values:

```
print(10 + 5)
```

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$

Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3
:=	print(x := 3)	x = 3 print(x)

Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	<code>x < 5 and x < 10</code>
or	Returns True if one of the statements is true	<code>x < 5 or x < 4</code>
not	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	<code>x & y</code>
	OR	Sets each bit to 1 if one of two bits is 1	<code>x y</code>
^	XOR	Sets each bit to 1 if only one of two bits is 1	<code>x ^ y</code>
~	NOT	Inverts all the bits	<code>~x</code>
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	<code>x << 2</code>
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	<code>x >> 2</code>

Python If ... Else

Syntax

if condition:

 # code block if condition is True

elif another_condition:

 # code block if this condition is True

else:

 # code block if none of the above are True

Example:

x = 10

if x > 0:

 print("Positive")

elif x == 0:

 print("Zero")

else:

 print("Negative")

Shorthand If: If you have only one statement to execute, you can put it on the same line as the if statement.

```
if a > b: print("a is greater than b")
```

Shorthand If...Else:

```
a = 2
```

```
b = 330
```

```
c = 4
```

```
print("A") if a > b else print("B")
```

AND, OR, NOT keyword:

```
if a > b and c > a:
```

```
    print("Both conditions are True")
```

```
if a > b or a > c:
```

```
    print("At least one of the conditions is True")
```

```
if not a > b:
```

```
    print("a is NOT greater than b")
```

Python Match



The **match** statement is used to perform different actions based on different conditions. Instead of writing many **if..else** statements, you can use the **match** statement.

The **match** statement selects one of many code blocks to be executed.

Syntax:

```
match expression:
```

```
    case x:
```

```
        code block
```

```
    case y:
```

```
        code block
```

```
    case z:
```

```
        code block
```

Example:

```
day = input("Enter day (1-7):");
```

```
match day:
```

```
    case 1:
```

```
        print("Monday")
```

```
    case 2:
```

```
        print("Tuesday")
```

```
    case 3:
```

```
        print("Wednesday")
```

```
    case 4:
```

```
        print("Thursday")
```

```
    case 5:
```

```
        print("Friday")
```

```
    case 6:
```

```
        print("Saturday")
```

```
    case 7:
```

```
        print("Sunday")
```


While loop



Python has two primitive loop commands:

- **while** loops
- **for** loops

With the **while** loop we can execute a set of statements as long as a condition is true.

Example:

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Break Statement: With the **break** statement we can stop the loop even if the while condition is true.

Break Statement Example:

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

continue Statement: With the **continue** statement we can stop the current iteration, and continue with the next.

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

For loop

A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the **for** keyword in other programming languages, and works more like an iterator method as found in other object-oriented programming languages.

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example 1:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

Example 2: Loop through the letters in the word "banana".

```
for x in "banana":  
    print(x)
```

Example 3: Using the range() function.

```
for x in range(6):  
    print(x)
```

Example 4:

```
for x in range(2, 6):  
    print(x)
```

Example 5: Starts, Ends, Increment

```
for x in range(2, 30, 3):  
    print(x)
```

Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

In Python a function is defined using the `def` keyword:

```
def my_function():  
    print("Hello from a function")
```

```
my_function();
```

Argument: *Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.*

Example:

```
def my_function(fname, lname):  
    print(fname + " " + lname)
```

```
my_function("Emil", "Refsnes")
```

Arbitrary Arguments, *args: If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition. This way the function will receive a *tuple* of arguments, and can access the items accordingly.

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])
```

```
my_function("Emil", "Tobias", "Linus")
```

Lambda Function

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Syntax: `lambda arguments : expression`

Example 1: Add 10 to argument *a*, and return the result

```
x = lambda a : a + 10
print(x(5))
```

Example 2: Multiple Arguments.

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
print(mydoubler(11))
```

Python Collections

There are four collection data types in the Python programming language:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- Dictionary is a collection which is ordered** and changeable. No duplicate members.

Lists

Lists are used to store multiple items in a single variable. Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are [Tuple](#), [Set](#), and [Dictionary](#), all with different qualities and usage. Lists are created using square brackets:

```
thislist = ["apple", "banana", "cherry"]
```

```
print(thislist)
```

List Items

List items are ordered, changeable, and allow duplicate values. List items are indexed, the first item has index **[0]**, the second item has index **[1]** etc.

```
print(thislist[0])
```

```
print(thislist[2])
```

Access List

Positive Indexing : List items are indexed and you can access them by referring to the index number:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

Negative Indexing: Negative indexing means start from the end **-1** refers to the last item, **-2** refers to the second last item etc.

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[-1])
```

Range Indexing: You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])
```

Changing List Items

To change the value of a specific item, refer to the index number:

Example 1:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example 2:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]  
thislist[1:3] = ["blackcurrant", "watermelon"]  
print(thislist)
```


Add List Items

Append Items: To add an item to the end of the list, use the `append()` method.

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)
```

Insert Items: To insert a list item at a specified index, use the `insert()` method. The `insert()` method inserts an item at the specified index:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)
```

Extend List: To append elements from *another list* to the current list, use the `extend()` method.

```
thislist = ["apple", "banana", "cherry"]  
tropical = ["mango", "pineapple", "papaya"]  
thislist.extend(tropical)  
print(thislist)
```

Add List Items

Append Items: To add an item to the end of the list, use the `append()` method.

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)
```

Insert Items: To insert a list item at a specified index, use the `insert()` method. The `insert()` method inserts an item at the specified index:

```
thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)
```

Extend List: To append elements from *another list* to the current list, use the `extend()` method.

```
thislist = ["apple", "banana", "cherry"]  
tropical = ["mango", "pineapple", "papaya"]  
thislist.extend(tropical)  
print(thislist)
```

Remove List Items

Remove Specific Item: The `remove()` method removes the specified item.

```
thislist = ["apple", "banana", "cherry"]  
thislist.remove("banana")  
print(thislist)
```

Remove Specific Index: The `pop()` method removes the specified index. And The `del` keyword also removes the specified index.

```
thislist = ["apple", "banana", "cherry"]  
thislist.pop(1)  
print(thislist)
```

or

```
thislist = ["apple", "banana", "cherry"]  
del thislist[0]  
print(thislist)
```

Remove Entire List: Using `del` keyword or `clear()` method can help to achieve this.

```
del thislist or thislist.clear()
```

Loop List Items

You can loop through the list items by using a **for** loop:

```
thislist = ["apple", "banana", "cherry"]  
for x in thislist:  
    print(x)
```

Using a While Loop: You can loop through the list items by using a **while** loop. Use the **len()** function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.

```
thislist = ["apple", "banana", "cherry"]  
  
i = 0  
  
while i < len(thislist):  
    print(thislist[i])  
  
    i = i + 1
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax:

```
newlist = [expression for item in iterable if condition == True]
```

Example:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x for x in fruits if "a" in x]
```

```
print(newlist)
```

List Sorting

List objects have a `sort()` method that will sort the list alphanumerically, ascending, by default:

Example:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]  
thislist.sort()  
print(thislist)
```

Sort Descending: To sort descending, use the keyword argument `reverse = True`:

Example:

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]  
thislist.sort(reverse = True)  
print(thislist)
```

List Copy

1. You can use the built-in List method **copy()** to copy a list.

```
thislist = ["apple", "banana", "cherry"]  
mylist = thislist.copy()  
print(mylist)
```

2. Another way to make a copy is to use the built-in method **list()**.

```
thislist = ["apple", "banana", "cherry"]  
mylist = list(thislist)  
print(mylist)
```

3. You can also make a copy of a list by using the **:** (slice) operator.

```
thislist = ["apple", "banana", "cherry"]  
mylist = thislist[:]  
print(mylist)
```

List Joins

1. There are several ways to join, or concatenate, two or more lists in Python. One of the easiest ways are by using the **+** operator.

```
list1 = ["a", "b", "c"]
```

```
list2 = [1, 2, 3]
```

```
list3 = list1 + list2
```

```
print(list3)
```

2. Use the **extend()** method to add list2 at the end of list1.

```
list1 = ["a", "b", "c"]
```

```
list2 = [1, 2, 3]
```

```
list1.extend(list2)
```

```
print(list1)
```


List Methods

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Tuples

Tuples are used to store multiple items in a single variable. Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Set](#), and [Dictionary](#), all with different qualities and usage. A tuple is a collection which is ordered and unchangeable. Tuples are written with round brackets.

Example:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple)
```

1. **Tuple Items:** Tuple items are ordered, unchangeable, and allow duplicate values. Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.
2. **Ordered**
3. **Unchangeable**
4. **Allows Duplicates**

Access Tuples

1. **Positive Indexing:** You can access tuple items by referring to the index number, inside square brackets.

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple[1])
```

2. **Negative Indexing:** Negative indexing means start from the end. **-1** refers to the last item, **-2** refers to the second last item etc.

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple[-1])
```

3. **Range Of Indexing:** You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:5])
```

Update Tuples

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called. But there is a way. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Adding Items in Tuples:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Unpack Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

```
fruits = ("apple", "banana", "cherry")
```

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

```
fruits = ("apple", "banana", "cherry")  
(green, yellow, red) = fruits  
print(green)  
print(yellow)  
print(red)
```

Loop Tuples

You can loop through the tuple items by using a **for** loop.

```
thistuple = ("apple", "banana", "cherry")  
  
for x in thistuple:  
    print(x)
```

Loop Through the Index Numbers: You can also loop through the tuple items by referring to their index number. Use the `range()` and `len()` functions to create a suitable iterable.

```
thistuple = ("apple", "banana", "cherry")  
  
for i in range(len(thistuple)):  
    print(thistuple[i])
```

Sets

Sets are used to store multiple items in a single variable. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are [List](#), [Tuple](#), and [Dictionary](#), all with different qualities and usage. A set is a collection which is ***unordered***, ***unchangeable****, and ***unindexed***. Duplicates are not allowed in set.

```
thisset = {"apple", "banana", "cherry"}
```

```
print(thisset)
```

The set() Constructor: It is also possible to use the `set()` constructor to make a set.

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
```

```
print(thisset)
```

Access Sets

You cannot access items in a set by referring to an index or a key. But you can loop through the set items using a **for** loop, or ask if a specified value is present in a set, by using the **in** keyword.

```
thisset = {"apple", "banana", "cherry"}
```

```
for x in thisset:
```

```
    print(x)
```

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}
```

```
print("banana" in thisset)
```

Check if "banana" is NOT present in the set:

```
thisset = {"apple", "banana", "cherry"}
```

```
print("banana" not in thisset)
```


Add Set Items

Once a set is created, you cannot change its items, but you can add new items. To add one item to a set use the **add()** method.

Example:

```
thisset = {"apple", "banana", "cherry"}  
thisset.add("orange")  
print(thisset)
```

To add items from another set into the current set, use the **update() method.**

```
thisset = {"apple", "banana", "cherry"}  
tropical = {"pineapple", "mango", "papaya"}  
thisset.update(tropical)  
print(thisset)
```

Remove Item Set

To remove an item in a set, use the **remove()**, or the **discard()** method.

Example: remove()

```
thisset = {"apple", "banana", "cherry"}  
thisset.remove("banana")  
print(thisset)
```

If the item to remove does not exist, **remove()** will raise an error.

Example: discard()

```
thisset = {"apple", "banana", "cherry"}  
thisset.discard("banana")  
print(thisset)
```

If the item to remove does not exist, **discard()** will NOT raise an error.

Remove Item Set

You can also use the `pop()` method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed. The return value of the `pop()` method is the removed item.

Example: `pop()`

```
thisset = {"apple", "banana", "cherry"}  
x = thisset.pop()  
print(x)  
print(thisset)
```

Note: Sets are *unordered*, so when using the `pop()` method, you do not know which item that gets removed.

The `clear()` method empties the set:

```
thisset.clear()
```

The `del` keyword will delete the set completely:

```
del thisset
```

Loop Set

You can loop through the set items by using a **for** loop.

Example:

```
thisset = {"apple", "banana", "cherry"}  
for x in thisset:  
    print(x)
```

Join Set

There are several ways to join two or more sets in Python. The **union()** and **update()** methods joins all items from both sets.

```
set1 = {"a", "b", "c"}  
set2 = {1, 2, 3}  
set3 = set1.union(set2) #set1.update(set2)  
print(set3)
```

Dictionary in Python

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered*, changeable and do not allow duplicates. As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
print(thisdict)  
  
print(thisdict["brand"])
```

Note: Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created. Dictionaries cannot have two items with the same key.

Change Dictionary Items

You can change the value of a specific item by referring to its key name:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
thisdict["year"] = 2018;  
  
thisdict.update({"year": 2020})
```

Note: The `update()` method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs.

Add Dictionary Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
  
thisdict["color"] = "red";  
  
thisdict.update({"color": "red"})  
  
print(thisdict);
```

Note: The `update()` method will update the dictionary with the items from the given argument. The argument must be a dictionary, or an iterable object with key:value pairs.

Dictionary Methods

Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and value
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns a list containing a tuple for each key value pair
<code>keys()</code>	Returns a list containing the dictionary's keys
<code>pop()</code>	Removes the element with the specified key
<code>popitem()</code>	Removes the last inserted key-value pair
<code>setdefault()</code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<code>update()</code>	Updates the dictionary with the specified key-value pairs
<code>values()</code>	Returns a list of all the values in the dictionary

OOPs in Python

OOP stands for Object-Oriented Programming.

Python is an object-oriented language, allowing you to structure your code using classes and objects for better organization and reusability.

Advantages of OOP

- Provides a clear structure to programs
- Makes code easier to maintain, reuse, and debug
- Helps keep your code DRY (Don't Repeat Yourself)
- Allows you to build reusable applications with less code

What are Classes and Objects?

Classes and objects are the two core concepts in object-oriented programming.

A class defines what an object should look like, and an object is created based on that class. For example:

Class	Objects
Fruit	Apple, Banana, Mango
Car	Volvo, Audi, Toyota

Objects & Class

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Example: Creating class & objects.

```
class MyClass:

    x = 5

p1 = MyClass()

print(p1.x)
```

The `__init__()` Function: All classes have a function called `__init__()`, which is always executed when the class is being initiated.

Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example:

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age

p1 = Person("John", 36)

print(p1.name)

print(p1.age)
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class. Child class is the class that inherits from another class, also called derived class.

Example:

```
class Person:

    def __init__(self, fname, lname):

        self.firstname = fname

        self.lastname = lname

    def printname(self):

        print(self.firstname, self.lastname)

x = Person("John", "Doe")

x.printname()
```

Python Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods / functions / operators with the same name that can be executed on many objects or classes.

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name. For example, say we have three classes: **Car**, **Boat**, and **Plane**, and they all have a method called **move()**:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def move(self):
        print("Drive!")
```

```
class Boat:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def move(self):
        print("Sail!")
```

```
class Plane:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def move(self):
        print("Fly!")
```

```
car1 = Car("Ford", "Mustang")
#Create a Car object
boat1 = Boat("Ibiza", "Touring 20")
#Create a Boat object
plane1 = Plane("Boeing", "747")
#Create a Plane object
```

```
for x in (car1, boat1, plane1):
    x.move()
```

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using:

- The **try** block lets you test a block of code for errors.
- The **except** block lets you handle the error.
- The **else** block lets you execute code when there is no error.
- The **finally** block lets you execute code, regardless of the result of the try- and except blocks.

Syntax:

try:

code that may raise an exception

except SomeError:

code that runs if exception occurs

else:

runs if no exception occurs

finally:

always runs (optional)

try:

num = int(input("Enter a number: "))

result = 10 / num

print("Result:", result)

except ZeroDivisionError:

print("Cannot divide by zero!")

except ValueError:

print("Invalid input! Please enter a number.")

else:

print("Division successful.")

finally:

print("Execution complete.")

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations
AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indentation is not correct

Exception	Description
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific exceptions
ZeroDivisionError	Raised when the second operator in a division is zero

File Handling

File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the `open()` function. The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist.

`"a"` - Append - Append means to add information to the end of an existing file or create a new file if it does not exist.

`"w"` - Write - Opens a file for writing, creates the file if it does not exist.

`"x"` - Create - Creates the specified file, returns an error if the file exists.

In addition you can specify if the file should be handled as binary or text mode.

`"t"` - Text - Default value. Text mode

`"b"` - Binary - Binary mode (e.g. images)

Example Syntax:

```
f = open("demofile.txt", "rt")
```

Because `"r"` for read, and `"t"` for text are the default values, you do not need to specify them.

File Handling

Reading Files

Assume we have the following file, located in the same folder as Python:

demofile.txt

Hello! Welcome to demofile.txt, This file is for testing, purposes. Good Luck!

To open the file, use the built-in `open()` function. The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

```
f = open("demofile.txt")
```

```
print(f.read())
```

```
f.close()
```

If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\\welcome.txt")
```

```
print(f.read())
```

Using the `with` statement: Then you do not have to worry about closing your files, the `with` statement takes care of that.

```
with open("demofile.txt") as f:
```

```
    print(f.read())
```

It reads the first line of the file:

```
with open("demofile.txt") as f:
```

```
    print(f.readline())
```


File Handling

Write Files

To write to an existing file, you must add a parameter to the `open()` function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example:

```
with open("demofile.txt", "a") as f:

    f.write("Now the file has more content!")

#open and read the file after the appending:

with open("demofile.txt") as f:

    print(f.read())
```

Overwrite Existing Content

To overwrite the existing content to the file, use the **w** parameter:

```
with open("demofile.txt", "w") as f:

    f.write("Woops! I have deleted the content!")

#open and read the file after the overwriting:

with open("demofile.txt") as f:

    print(f.read())
```

File Handling

Delete Files

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example:

```
import os

os.remove("demofile.txt")
```

Check If Exists:

```
import os

if os.path.exists("demofile.txt"):

    os.remove("demofile.txt")

else:

    print("The file does not exist")
```

Python Modules

Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application. To create a module just save the code you want in a file with the file extension **.py**:

Save this code in a file named **mymodule.py**

```
def greeting(name):  
  
    print("Hello, " + name)  
  
person1 = {  
  
    "name": "John",  
  
    "age": 36,  
  
    "country": "Norway"  
  
}
```

Now we can use the module we just created, by using the **import** statement. Import the module named mymodule, and call the greeting function:

```
import mymodule  
  
mymodule.greeting("Jonathan")  
  
a = mymodule.person1["age"]  
  
print(a) #Import the module named mymodule,  
and access the person1 dictionary
```

Import only the person1 dictionary from the module:

```
from mymodule import person1  
  
print (person1["age"])
```

Python Numpy

NumPy is a Python library. NumPy is used for working with arrays. NumPy is short for "Numerical Python".



Installation:

\$pip install numpy

Debugging:

\$python -m pip install --upgrade pip
or

\$py -m pip install --upgrade pip

Example:

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

Numpy-Dimensions



In NumPy, the term "dimension" refers to the number of axes along which the data is organized in an array. It's essentially the number of indices needed to access a specific element within the array. Dimensions are also referred to as axes in NumPy. When the array is created, you can define the number of dimensions by using the **ndmin** argument.

0-D Arrays: 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np  
  
arr = np.array(42)  
  
print(arr)
```

1D Arrays: An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

Create a 1-D array containing the values 1,2,3,4,5:

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

2D Arrays:

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

3D Arrays:

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

Numpy-Access Items



NumPy



python™

Access Array Elements

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Access 1D Arrays

```
import numpy as np

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

print(arr[0])

print(arr[2])
```

Access 2D Arrays

```
import numpy as np

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

print('5th element on 2nd row: ', arr[1, 4])
```

Access 3D Arrays: To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

```
import numpy as np

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

print(arr[0, 1, 2])
```

Numpy-Data Types



NumPy has some extra data types, and refer to data types with one character, like **i** for integers, **u** for unsigned integers etc. The NumPy array object has a property called **dtype** that returns the data type of the array. Below is a list of all data types in NumPy and the characters used to represent them.

- **i** - integer
- **b** - boolean
- **u** - unsigned integer
- **f** - float
- **c** - complex float
- **m** - timedelta
- **M** - datetime
- **O** - object
- **S** - string
- **U** - unicode string
- **V** - fixed chunk of memory for other type (void)

Creating Arrays With a Defined Data Type:

We use the **array()** function to create arrays, this function can take an optional argument: **dtype** that allows us to define the expected data type of the array elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)

print(arr.dtype)
```

Create an array with data type 4 bytes integer:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)

print(arr.dtype)
```

Numpy-Array Reshape



NumPy arrays have an attribute called **shape** that returns a tuple with each index having the number of corresponding elements.

Print the shape of a 2-D array:

```
import numpy as np

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

print(arr.shape)
```

Reshaping arrays: Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

Convert the following 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np

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

newarr = arr.reshape(4, 3)

print(newarr)
```

Convert the following 1D array with 12 elements into a 3D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements.

```
import numpy as np

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

newarr = arr.reshape(2, 3, 2)

print(newarr)
```


Numpy-Array Join



Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array. Whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the **concatenate()** function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example:

```
import numpy as np

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

Join two 2-D arrays along rows (axis=1):

```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

In numpy there are more types of functions to join Numpy arrays.

stack(): Stacking is same as concatenation, the only difference is that stacking is done along a new axis. We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

hstack(): NumPy provides a helper function: **hstack()** to stack along rows.

vstack(): to stack along columns.

dstack(): to stack along height, which is the same as depth.

Numpy-Array Split



Splitting is reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

Split the array in 3 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 3)

print(newarr)
```

Note: We also have the method `split()` available but it will not adjust the elements when elements are less in source array for splitting like in example above, `array_split()` worked properly but `split()` would fail.

The example above returns three 2-D arrays. In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the column (`axis=1`).

Split the 2-D array into three 2-D arrays along columns.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

Numpy-Array Search



You can search an array for a certain value, and return the indexes that get a match. To search an array, use the `where()` method.

Find the indexes where the value is 4:

```
import numpy as np

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

x = np.where(arr == 4)

print(x)
```

Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The `searchsorted()` method is assumed to be used on sorted arrays.

Example

Find the indexes where the value 7 should be inserted:

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

Note: By default the leftmost index is returned, but we can give `side='right'` to return the rightmost index instead.

Numpy-Array Sorting



Sorting means putting elements in an *ordered sequence*. *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

Sort the array:

```
import numpy as np

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

print(np.sort(arr))
```

Note: This method returns a copy of the array, leaving the original array unchanged.

You can also sort arrays of strings, or any other data type:

Example

Sort the array alphabetically:

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

Sort a boolean array:

```
import numpy as np

arr = np.array([True, False, True])

print(np.sort(arr))
```

Numpy-Array Filter



Getting some elements out of an existing array and creating a new array out of them is called *filtering*. In NumPy, you filter an array using a *boolean index list*.

A boolean index list is a list of booleans corresponding to indexes in the array.

If the value at an index is **True** that element is contained in the filtered array, if the value at that index is **False** that element is excluded from the filtered array.

```
import numpy as np

arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)
```

Numpy-Random



What is a Random Number?

Random number does NOT mean a different number every time. Random means something that cannot be predicted logically. In simple terms which is uncertain.

Generate Random Number: NumPy offers the `random` module to work with random numbers.

```
from numpy import random
```

```
x = random.randint(100)
```

```
print(x)
```

Generate Random Float: The random module `rand()` method returns a random float between 0 and 1.

```
x = random.rand()
```

Generate Random Array: In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers: The `randint()` method takes a `size` parameter where you can specify the shape of an array.

Generate a 2D array with 3 rows, each row containing 5 random integers from 0 to 100:

```
from numpy import random
```

```
x = random.randint(100, size=(3, 5))
```

```
print(x)
```

Normal (Gaussian) Distribution



Normal Distribution

The Normal Distribution is one of the most important distributions. It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss.

It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc. Use the `random.normal()` method to get a Normal Data Distribution.

It has three parameters:

loc - (Mean) where the peak of the bell exists.

scale - (Standard Deviation) how flat the graph distribution should be.

size - The shape of the returned array.

Example 1: Generate a random normal distribution of size 2x3.

```
from numpy import random  
  
x = random.normal(size=(2, 3))  
  
print(x)
```

Example 2: Generate a random normal distribution of size 2x3 with mean at 1 and standard deviation of 2.

```
from numpy import random  
  
x = random.normal(loc=1, scale=2, size=(2, 3))  
  
print(x)
```

Binomial Distribution



Binomial Distribution is a *Discrete Distribution*. It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.

It has three parameters:

n - number of trials.

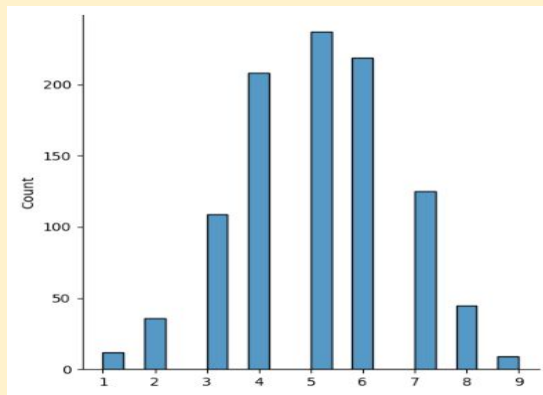
p - probability of occurrence of each trial (e.g. for toss of a coin 0.5 each).

size - The shape of the returned array.

Discrete Distribution: The distribution is defined at separate set of events, e.g. a coin toss result is discrete as it can be only head or tails whereas height of people is continuous as it can be 170, 170.1, 170.11 and so on.

Example 1: Given 10 trials for coin toss generate 1000 data points.

```
from numpy import random  
  
x = random.binomial(n=10, p=0.5, size=1000)  
  
print(x)
```



Pandas



What is Pandas?

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data. The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.



Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories. Pandas can clean messy data sets, and make them readable and relevant. Relevant data is very important in data science.

Pandas-Installation



Installation of Pandas

If you have **Python** and **PIP** already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
$pip install pandas
```

Debugging for pip:

```
$python -m pip install --upgrade pip  
or  
$py -m pip install --upgrade pip
```

Once Pandas is installed, import it in your applications by adding the **import** keyword:

Example for pandas:

```
import pandas  
  
mydataset = {  
    'cars': ["BMW", "Volvo", "Ford"],  
    'passings': [3, 7, 2]  
}  
  
myvar = pandas.DataFrame(mydataset)  
  
print(myvar)
```

Pandas-Series



What is a Series?

A Pandas Series is like a column in a table. It is a one-dimensional array holding data of any type.

Create a simple Pandas Series from a list:

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a)
```

```
print(myvar)
```

Return the first value of the Series:

```
print(myvar[0])
```

Create Labels

With the **index** argument, you can name your own labels.

Example

Create your own labels:

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a, index = ["x", "y", "z"])
```

```
print(myvar)
```

When you have created labels, you can access an item by referring to the label.

```
print(myvar["y"])
```

Pandas DataFrames



What is a Data Frame?

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Create a simple Pandas DataFrame:

```
import pandas as pd

data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

Locate Row

As you can see from the result above, the Data Frame is like a table with rows and columns. Pandas use the **loc** attribute to return one or more specified row(s).

Example

Return row 0:

```
#refer to the row index:
print(df.loc[0])
```

Return row 0 and 1:

```
#use a list of indexes:
print(df.loc[[0, 1]])
```

Read CSV & Excel



A simple way to store big data sets is to use CSV files (comma separated files). CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

Example:

Load the CSV into a DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

To read an Excel file using **Pandas** in Python, you use the `read_excel()` function.

```
import pandas as pd

# Reading the Excel file

df = pd.read_excel("students_data.xlsx") #
Displaying first few rows

print(df.head())

or

print(df)
```

Read JSON



Big data sets are often stored, or extracted as JSON. JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.

Load the JSON file into a DataFrame:

```
import pandas as pd

df = pd.read_json('data.json')

print(df)
```

Analyzing DataFrames



Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the `head()` method.

The `head()` method returns the headers and a specified number of rows, starting from the top.

Example:

Get a quick overview by printing the first 10 rows of the DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head(10))
```

There is also a `tail()` method for viewing the *last* rows of the DataFrame. The `tail()` method returns the headers and a specified number of rows, starting from the bottom.

Example

Print the last 5 rows of the DataFrame:

```
print(df.tail())
```

Info About the Data

The DataFrames object has a method called `info()`, that gives you more information about the data set.

Print information about the data:

```
print(df.info())
```

Data Cleaning



Data Cleaning: Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

Empty cells can potentially give you a wrong result when you analyze data.

Remove all rows with NULL values:

```
import pandas as pd
df = pd.read_csv('data.csv')
df.dropna(inplace = True)
print(df.to_string())
```

Note: Now, the `dropna(inplace = True)` will NOT return a new DataFrame, but it will remove all rows containing NULL values from the original DataFrame.

Replace Empty Values: Another way of dealing with empty cells is to insert a *new* value instead. This way you do not have to delete entire rows just because of some empty cells.

The `fillna()` method allows us to replace empty cells with a value:

Replace NULL values with the number 130:

```
import pandas as pd
df = pd.read_csv('data.csv')
df.fillna(130, inplace = True)
```


Data Cleaning



Removing Rows

The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the **dropna()** method.

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
```

```
df['Date'] = pd.to_datetime(df['Date'])
```

```
df.dropna(subset=['Date'], inplace = True)
```

```
print(df.to_string())
```

Original:

	Duration	Date	Pulse	Max-pulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
3	45		109	175	282.4
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0

After:

	Duration	Date	Pulse	Max-pulse	Calories
0	60	2020-12-01	110	130	409.1
1	60	2020-12-02	117	145	479.0
2	60	2020-12-03	103	135	340.0
4	45	2020-12-05	117	148	406.0
5	60	2020-12-06	102	127	300.0

Data Cleaning



Cleaning Wrong Data

"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".

Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

Replacing Values

One way to fix wrong values is to replace them with something else.

In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

Example

Set "Duration" = 45 in row 7:

```
df.loc[7, 'Duration'] = 45
```

For small data sets you might be able to replace the wrong data one by one, but not for big data sets. To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.

```
for x in df.index:  
    if df.loc[x, "Duration"] > 120:  
        df.loc[x, "Duration"] = 120
```

Data Cleaning



Duplicate Data

Duplicate rows are rows that have been registered more than one time. By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.

To discover duplicates, we can use the `uplicated()` method.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.duplicated().sum())
```

Removing Duplicates

To remove duplicates, use the `drop_duplicates()` method.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')

df.drop_duplicates(inplace = True)

print(df)
```

Remember: The `(inplace = True)` will make sure that the method does NOT return a *new* DataFrame, but it will remove all duplicates from the *original* DataFrame.

Data Correlations



Finding Relationships

A great aspect of the Pandas module is the `corr()` method.

The `corr()` method calculates the relationship between each column in your data set.

Example:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.corr())
```

Perfect Correlation:

We can see that "Duration" and "Duration" got the number `1.000000`, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

"Duration" and "Calories" got a `0.922721` correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

Bad Correlation:

"Duration" and "Maxpulse" got a `0.009403` correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

Python Matplotlib



What is Matplotlib?


Matplotlib is a low level graph plotting library in python that serves as a visualization utility. It was created by **John D. Hunter**.



Matplotlib is open source and we can use it freely. Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Matplotlib Installation

matplotlib

 python™

Installation of Matplotlib

If you have [Python](#) and [PIP](#) already installed on a system, then installation of Matplotlib is very easy.

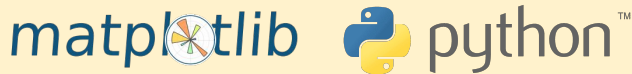
Install it using this command:

```
$pip install matplotlib
```

Debugging for pip:

```
$python -m pip install --upgrade pip  
or  
$py -m pip install --upgrade pip
```

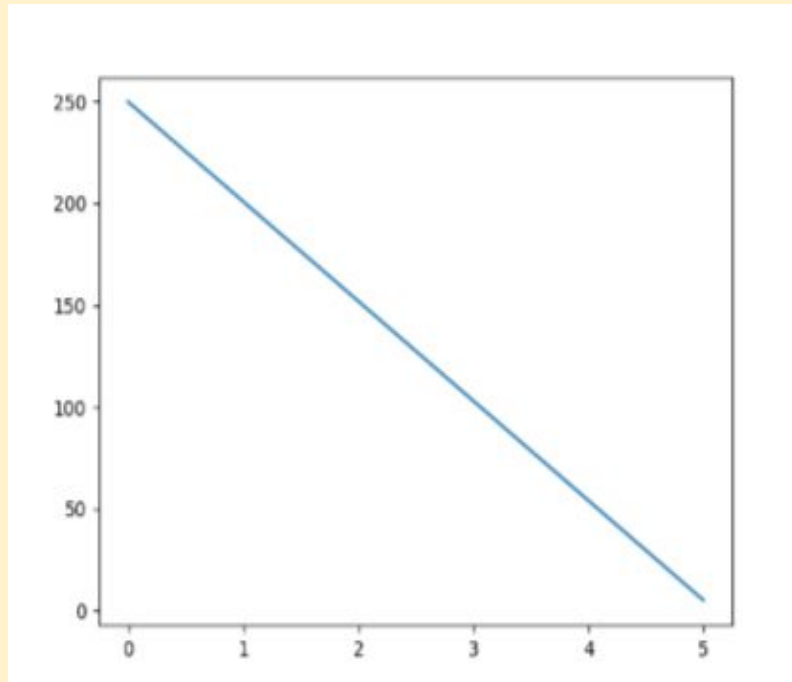
Matplotlib Pyplot



Pyplot: Most of the Matplotlib utilities lies under the **pyplot** submodule, and are usually imported under the **plt** alias.

Example: Draw a line in a diagram from position (5, 5) to position (0, 250):

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([5, 0])
ypoints = np.array([5, 250])
plt.plot(xpoints, ypoints)
plt.show()
```



Matplotlib Plotting



Plotting x and y points

The `plot()` function is used to draw points (markers) in a diagram. By default, the `plot()` function draws a line from point to point. The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the x-axis. Parameter 2 is an array containing the points on the y-axis.

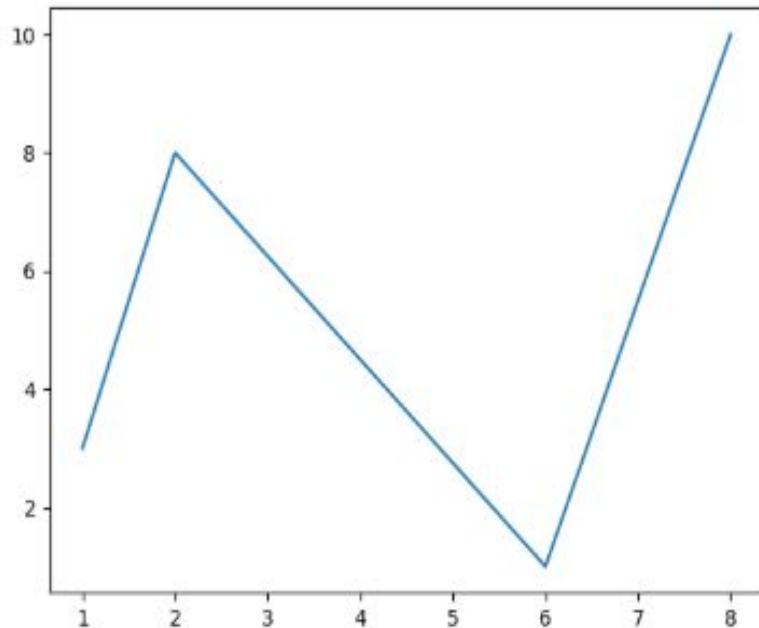
Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

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

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)

plt.show()
```



Matplotlib Markers

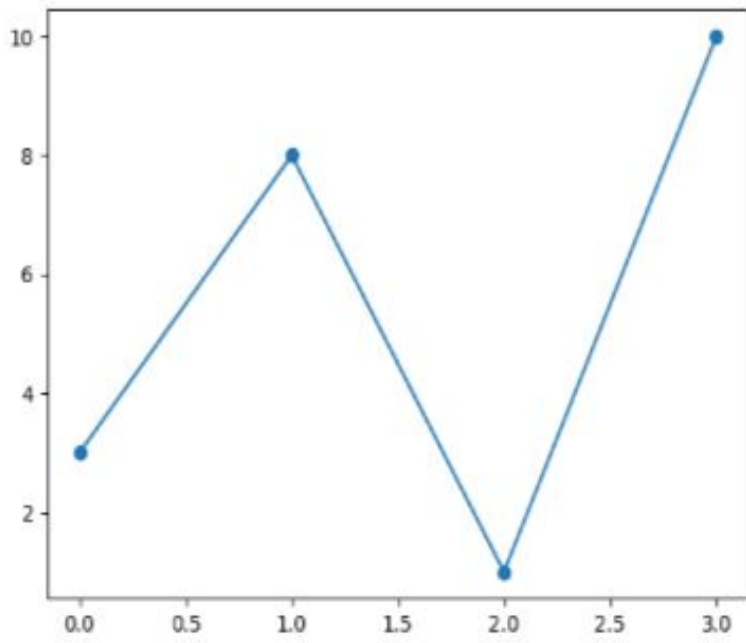
Markers

You can use the keyword argument **marker** to emphasize each point with a specified marker.

Example:

Mark each point with a circle:

```
import matplotlib.pyplot as plt  
  
import numpy as np  
  
ypoints = np.array([3, 8, 1, 10])  
  
plt.plot(ypoints, marker = 'o')  
  
plt.show()
```



Markers List

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
'f'	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'p'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon

'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
' '	Vline
'_'	Hline

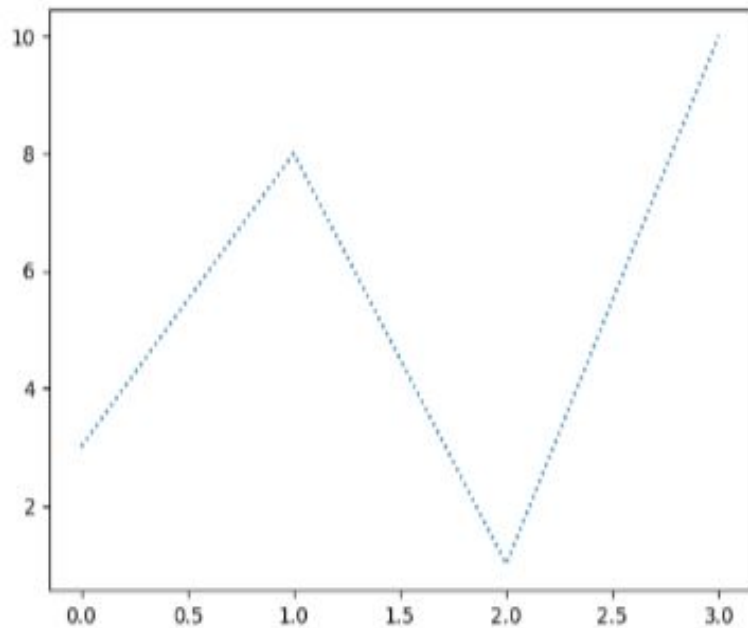
Line Style

Linestyle

You can use the keyword argument **linestyle**, or shorter **ls**, to change the style of the plotted line.

Use a dotted line:

```
import matplotlib.pyplot as plt  
  
import numpy as np  
  
ypoints = np.array([3, 8, 1, 10])  
  
plt.plot(ypoints, linestyle = 'dotted')  
  
plt.show()
```



Line Style

Linestyle

You can use the keyword argument **linestyle**, or shorter **ls**, to change the style of the plotted line.

Use a dotted line:

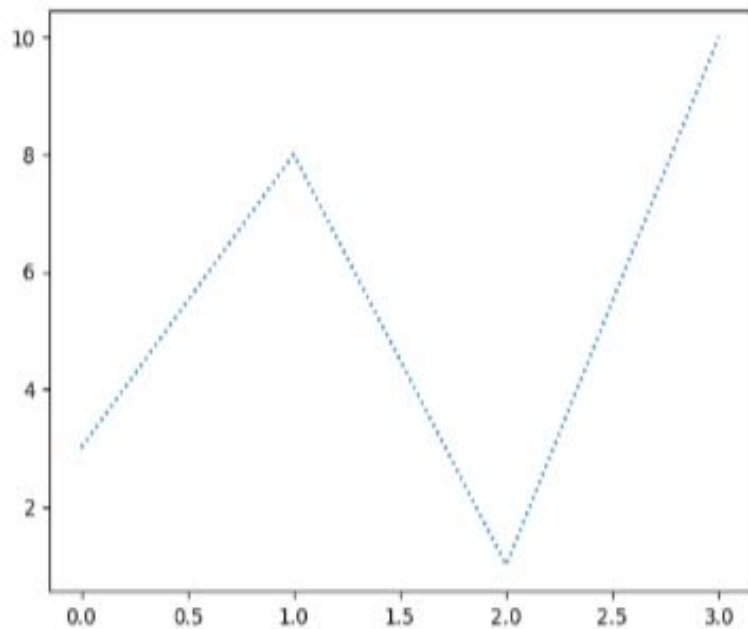
```
import matplotlib.pyplot as plt

import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')

plt.show()
```

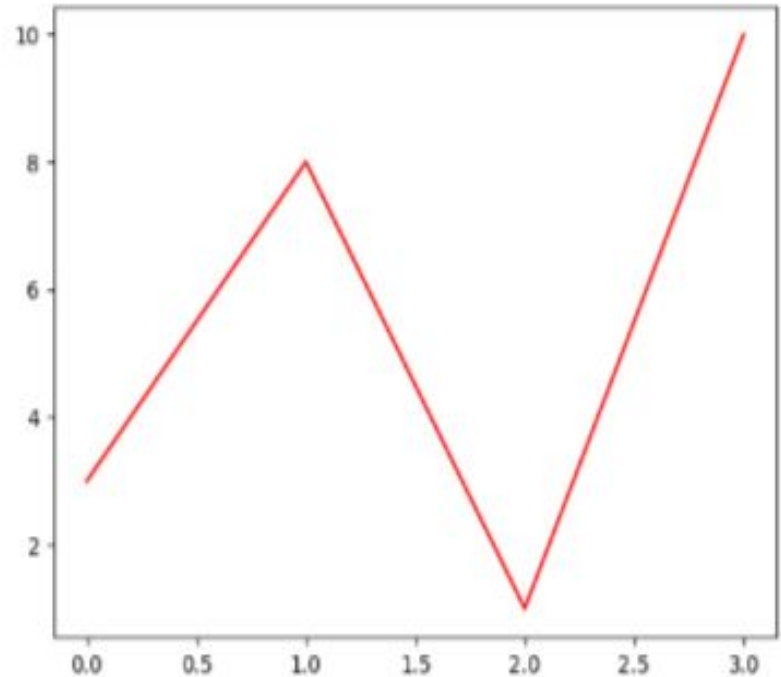


Line Style

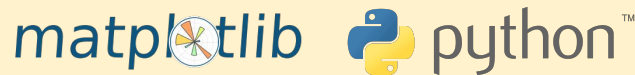
Style	Or
'solid' (default)	'-'
'dotted'	'.'
'dashed'	'--'
'dashdot'	'-.'
'None'	"" or ''

Line Color: You can use the keyword argument **color** or the shorter **c** to set the color of the line.

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, color = 'red')
plt.show()
```



Matplotlib Labels and Title



Create Labels for a Plot

With Pyplot, you can use the `xlabel()` and `ylabel()` functions to set a label for the x- and y-axis.

```
import numpy as np

import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105,
              110, 115, 120, 125])

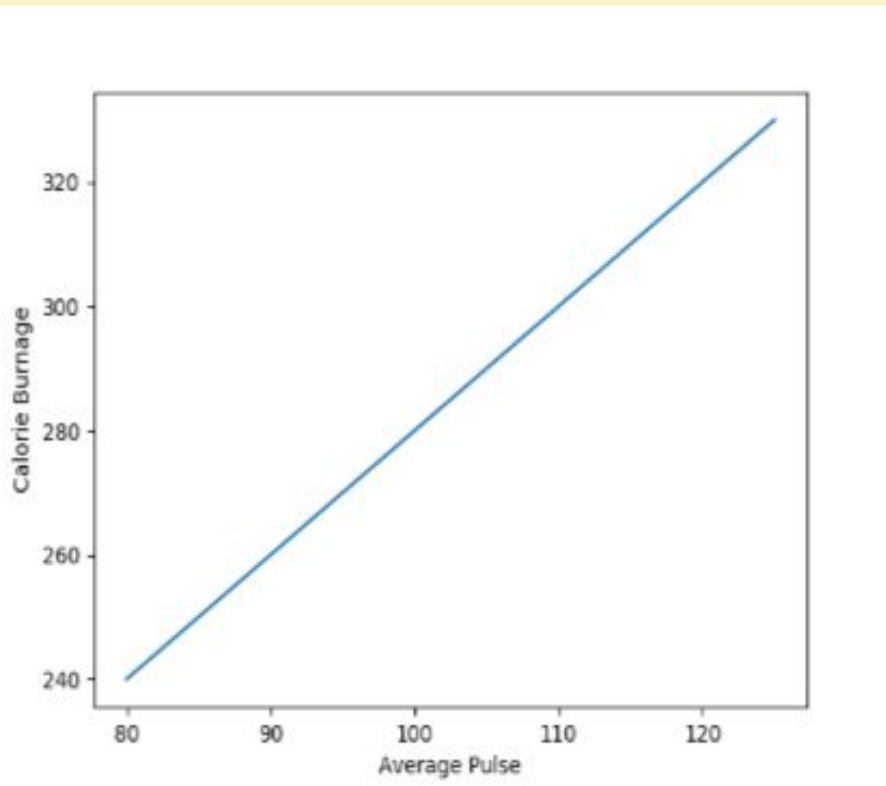
y = np.array([240, 250, 260, 270, 280, 290,
              300, 310, 320, 330])

plt.plot(x, y)

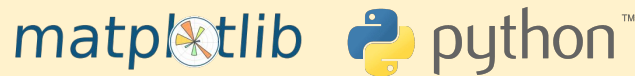
plt.xlabel("Average Pulse")

plt.ylabel("Calorie Burnage")

plt.show()
```



Matplotlib Labels and Title



Create Labels for a Plot

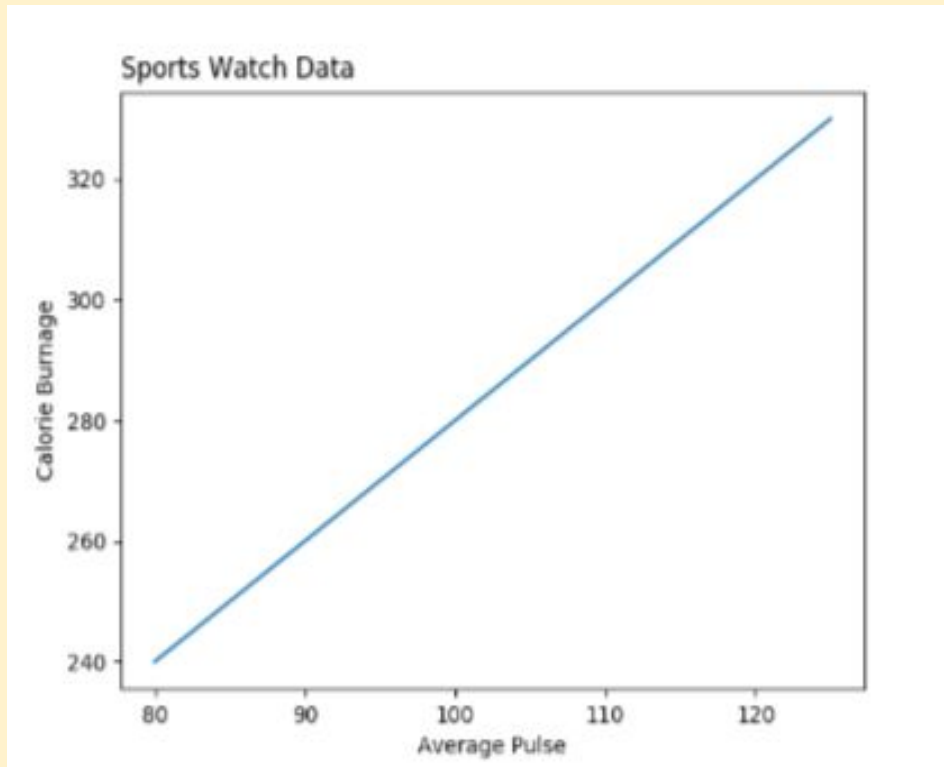
With Pyplot, you can use the `xlabel()` and `ylabel()` functions to set a label for the x- and y-axis.

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

x = np.array([80, 85, 90, 95, 100, 105,
             110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290,
             300, 310, 320, 330])

plt.title("Sports Watch Data", loc =
'left')

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.plot(x, y)
plt.show()
```



Grid Lines

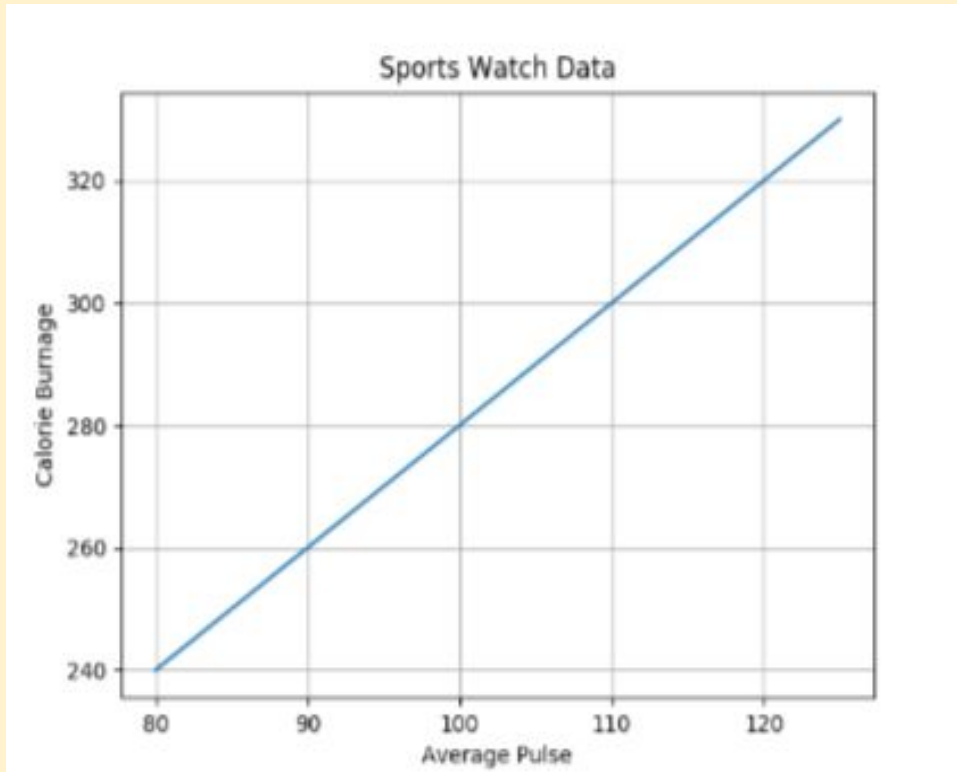
Add Grid Lines to a Plot

With Pyplot, you can use the `grid()` function to add grid lines to the plot.

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

x = np.array([80, 85, 90, 95, 100, 105,
              110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290,
              300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.plot(x, y)
plt.grid()
plt.show()
```



Subplot

The subplot() Function

The `subplot()` function takes three arguments that describes the layout of the figure. The layout is organized in rows and columns, which are represented by the *first* and *second* argument. The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)  
#the figure has 1 row, 2 columns, and this plot is thefirst plot.
```

```
plt.subplot(1, 2, 2)  
#the figure has 1 row, 2 columns, and this plot is thesecond plot.
```

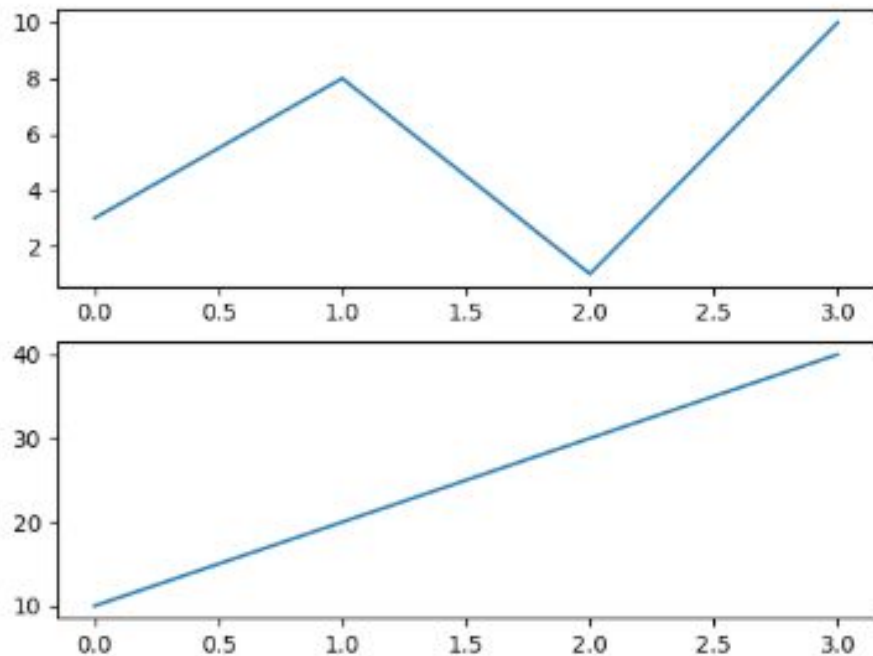
Subplot

Example:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 1, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 1, 2)
plt.plot(x,y)
plt.show()
```



Scatter Plot

Creating Scatter Plots

With Pyplot, you can use the `scatter()` function to draw a scatter plot. The `scatter()` function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis.

Example:

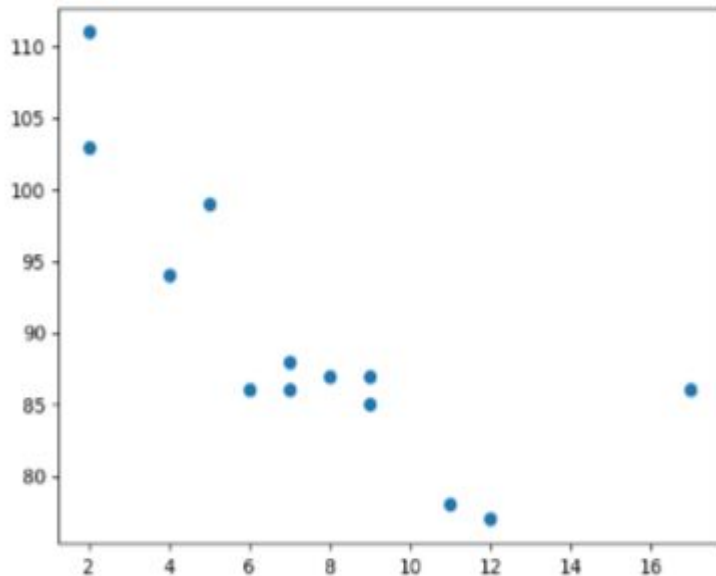
```
import matplotlib.pyplot as plt

import numpy as np

x = np.array([5, 7, 8, 7, 2, 17, 2, 9, 4, 11, 12, 9, 6])
y = np.array([99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86])

plt.scatter(x, y)

plt.show()
```



Bar Plot

Creating Bars

With Pyplot, you can use the `bar()` function to draw bar graphs. The `bar()` function takes arguments that describes the layout of the bars. The categories and their values represented by the *first* and *second* argument as arrays.

```
import matplotlib.pyplot as plt

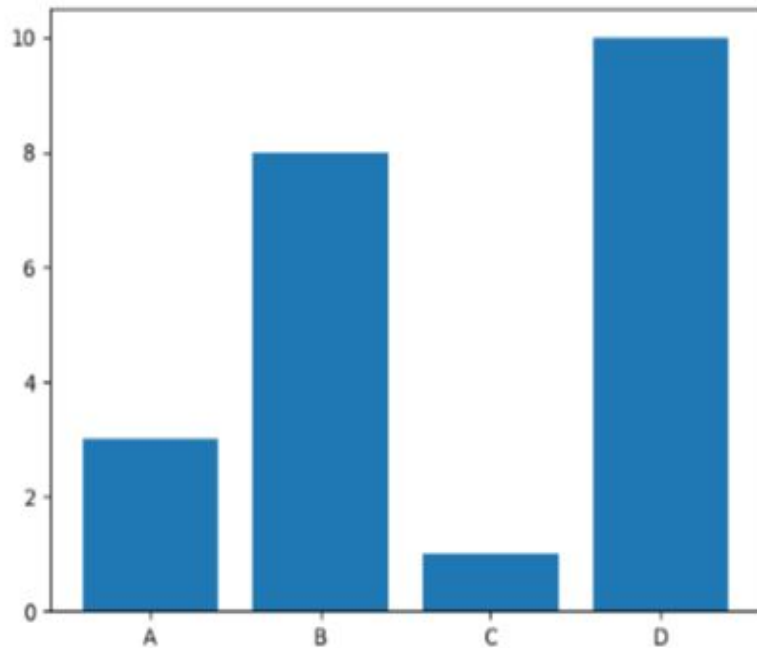
import numpy as np

x = np.array(["A", "B", "C", "D"])

y = np.array([3, 8, 1, 10])

plt.bar(x, y)

plt.show()
```



Bar Plot

Horizontal Bars

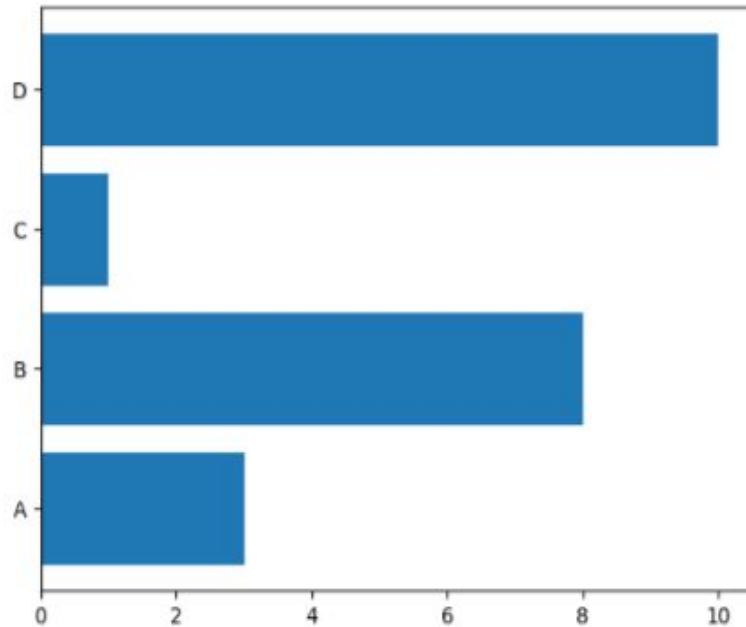
If you want the bars to be displayed horizontally instead of vertically, use the `barh()` function.

Example:

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```

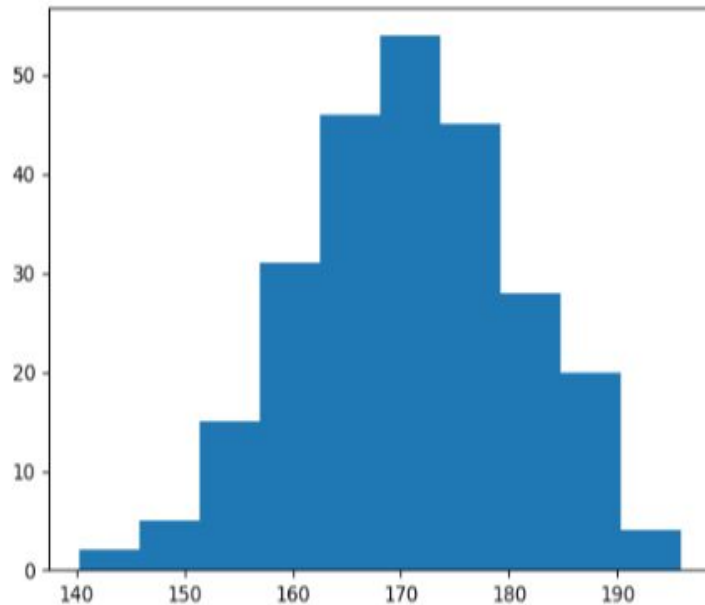


Histogram

A histogram is a graph showing *frequency* distributions. It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this.

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.random.normal(170, 10, 250)  
  
plt.hist(x)  
plt.show()
```



Pie Chart

Creating Pie Charts

With Pyplot, you can use the `pie()` function to draw pie charts.

Example: A simple pie chart.

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

```
y = np.array([35, 25, 25, 15])
```

```
plt.pie(y)
```

```
plt.show()
```



Pie Chart

Labels

Add labels to the pie chart with the `labels` parameter. The `labels` parameter must be an array with one label for each wedge.

```
import matplotlib.pyplot as plt

import numpy as np

y = np.array([35, 25, 25, 15])

mylabels = ["A", "B", "C", "D"]

plt.pie(y, labels = mylabels)

plt.show()
```



Web API in Python



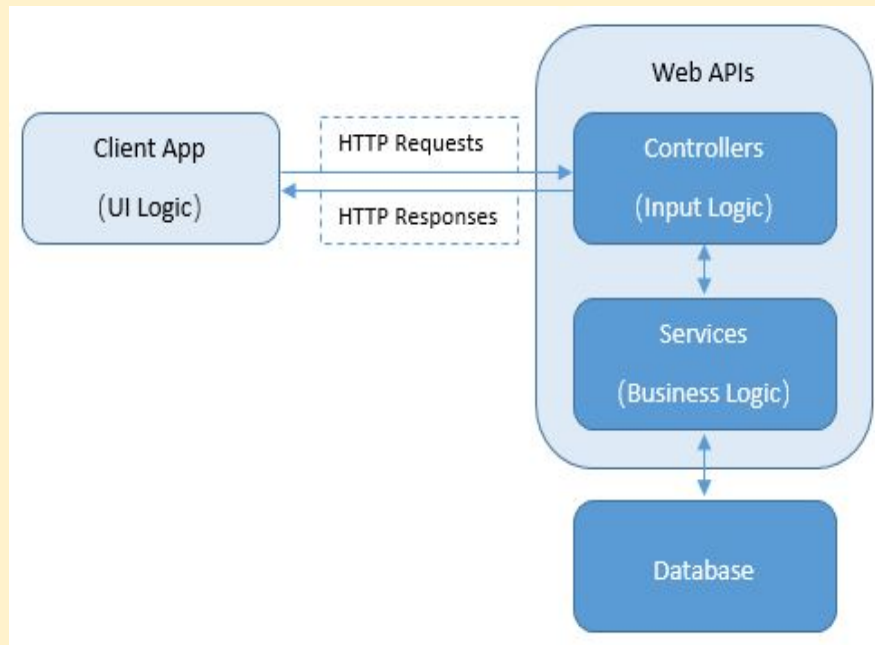
What is Web API in Python?

API: *Application Programming Interface.*

Web API: A web service that allows applications to communicate and share data over the internet using HTTP.

Examples:

- Weather API: Get current temperature of a city.
- Currency Exchange API: Get latest conversion rates.



Web API in Python



Why Use Web APIs?

Explanation:

- Automate data retrieval
- Access real-time information
- Use external services (like Google Maps, Weather, Twitter)

Example Use Cases:

- Stock price updates
- Weather forecasting
- Social media integration



Web API-JSON



JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation. Python has a built-in package called **json**, which can be used to work with JSON data.

Convert from JSON to Python: If you have a JSON string, you can parse it by using the **json.loads()** method.

```
import json

x = '{ "name":"John", "age":30, "city":"New York"}'

y = json.loads(x)

print(y["age"])
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the **json.dumps()** method.

```
import json

x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

y = json.dumps(x)

print(y)
```

Web API-JSON



You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

Convert a Python object containing all the legal data types:

```
import json

x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann", "Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

print(json.dumps(x))
```

Web API-JSON



Convert a Python object containing all the legal data types:

```
import json

x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann", "Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

print(json.dumps(x))
```

Accessing the Json format:

Task	Python Code	Output
Get name	<code>data["name"]</code>	'John'
Get first child	<code>data["children"][0]</code>	'Ann'
Get second car model	<code>data["cars"][1]["model"]</code>	'Ford Edge'
Get MPG of BMW	<code>data["cars"][0]["mpg"]</code>	27.5
Check if married	<code>data["married"]</code>	True
Check if pets exist	<code>data["pets"] is None</code>	True

Web API Python



Python Requests

Make a request to a web page, and print the response text. The **requests** module allows you to send HTTP requests using Python.

The HTTP request returns a Response Object with all the response data (content, encoding, status, etc).

Installation:

```
$pip install requests
```

Usage:

```
import requests  
  
x = requests.get('https://en.wikipedia.org/wiki/Machine_learning')  
print(x.text)
```

Web API Python



Python Requests Methods

Method	Description
<code><u>delete(url, args)</u></code>	Sends a DELETE request to the specified url
<code><u>get(url, params, args)</u></code>	Sends a GET request to the specified url
<code><u>head(url, args)</u></code>	Sends a HEAD request to the specified url
<code>patch(url, data, args)</code>	Sends a PATCH request to the specified url
<code><u>post(url, data, json, args)</u></code>	Sends a POST request to the specified url
<code>put(url, data, args)</code>	Sends a PUT request to the specified url
<code>request(method, url, args)</code>	Sends a request of the specified method to the specified url

Web API Python



Weather Details Fetching Using Web API

Step 01: Visit <https://developer.accuweather.com/>

Step 02: Register Yourself using valid credentials.

Step 03: Login into dashboard

Step 04: Click on My Apps

Step 05: Click on add a new app

Step 06: Click on the app and copy the key



Web API Python



Weather Details Fetching Using Web API

Step 01: Get The location Key using API Call

```
import requests

api_key = "API_Key"
city = input("Enter a City Name:")
url = f"http://dataservice.accuweather.com/locations/v1/cities/search?q={city}&apikey={api_key}"

def fetchLocationKey():
    location_url = url
    response = requests.get(location_url)
    data = response.json()
    if response.status_code == 200 and data:
        location_key = data[0]['Key']
        return location_key
    else:
        print("Error fetching location key")

location_Key = fetchLocationKey()
```

Web API Python



Weather Details Fetching Using Web API

Step 02: Fetch Weather Details by location key

```
weather_url = f"http://dataservice.accuweather.com/currentconditions/v1/{location_Key}?apikey={api_key}"

# Make the request
response = requests.get(weather_url)

# Parse the response
if response.status_code == 200:
    weather_data = response.json()
    if weather_data:
        temperature = weather_data[0]['Temperature']['Metric']['Value']
        unit = weather_data[0]['Temperature']['Metric']['Unit']
        description = weather_data[0]['WeatherText']

        print(f"Current weather in {city}: {description}, {temperature}°{unit}")
    else:
        print("No weather data found")
else:
    print(f"Failed to fetch weather. Status: {response.status_code}")
```

Database Connectivity



Introduction to Database Connectivity





♦ What is Database Connectivity?

It refers to how a program communicates with a **database** to store, retrieve, update, or delete data. Python supports many databases; here we use **SQLite**, which is built-in and file-based.

♦ Introduction to SQLite3

SQLite is a **lightweight, embedded, relational database engine** that is built into Python via the `sqlite3` module. Unlike other database systems like MySQL or PostgreSQL, **SQLite doesn't require a server**—it stores the entire database as a **single file** on disk (`.db`).

♦ Why Use SQLite3?

-  *Self-contained:* Stores the whole database in one file.
-  *Serverless:* No need to install or manage a database server.
-  *Built-in with Python:* No external libraries needed.
-  *Great for learning, small apps, prototyping.*

Database Connectivity



Connecting to SQLite using **sqlite3**

Import & Connect

```
import sqlite3  
  
conn = sqlite3.connect('students.db') # creates file if not exists  
  
cursor = conn.cursor()
```

◆ Create Cursor

`cursor` is used to execute SQL commands

◆ Close Connection

```
conn.close()
```

Database Connectivity



Creating Tables

♦ Create Table Query:

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY,
    name TEXT,
    age INTEGER
)
""")
conn.commit()
```

♦ Notes:

- **IF NOT EXISTS:** avoids duplicate table error
- **commit():** saves changes

Database Connectivity



CRUD Operations (Insert & Read)

◆ INSERT Data:

```
cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)", ("MyName", 18))  
  
conn.commit()
```

◆ SELECT Data:

```
cursor.execute("SELECT * FROM students")  
  
rows = cursor.fetchall()  
  
for row in rows:  
    print(row)
```

Database Connectivity



CRUD Operations (Update & Delete)

◆ UPDATE Data:

```
cursor.execute("UPDATE students SET age = ? WHERE name = ?", (23, "Arun"))
conn.commit()
```

◆ DELETE Data:

```
cursor.execute("DELETE FROM students WHERE name = ?", ("Arun",))
conn.commit()
```



Final Tip:

Always use `commit()` after INSERT, UPDATE, or DELETE because `commit()` is used to **save** the changes (like INSERT, UPDATE, DELETE) **permanently** into the database file. Without calling `commit()`, those changes are only temporary (in memory) and will be **lost when the program ends**.

What We Learned



What You've Learned:

-  Basics of Python: Syntax, Variables, Control Flow
-  Functions & Modules for reusable code
-  Data Structures: List, Tuple, Set, Dictionary
-  String Handling & File Operations
-  Exception Handling & Debugging techniques
-  Object-Oriented Programming (OOP)
-  Working with Libraries: NumPy, pandas
-  Data Visualization using matplotlib
-  Web APIs & JSON for real-time data
-  SQLite Database Connectivity & CRUD operations

What Next?



What's Next?

- ♦ Build Real Projects:
 - Weather App, Expense Tracker, Data Dashboard
- ♦ Learn Advanced Concepts:
 - Web Development (Flask, Django)
 - Automation, Web Scraping, Data Science
- ♦ Explore Career Paths:
 - Python Developer, Data Analyst, ML Engineer
- ♦ Keep Practicing & Share Your Work on GitHub!