**Introduction to python programming**[**¶**](https://devu.in/tutorial/#Introduction-to-python-programming)

Programming is an act of giving instructions to computer for processing and publishing information. Their are multiple languages to do this task. But an easy and powerful langauage is what the world is looking forward to. Python programming language is just the right choice, with a very easy *English* like syntax and computing abilities in multiple different fields like testing automation, network scripting, devops, data analytics and machine learning, web-framework etc.

Python language was created in *1989* by Guido van Rossum. Python3 was realased on 3rd December, 2008. Since the beginning python programming language has featured object-oriented, interactive and interpreted coding style. The emphasis was always on clean and readable code with sacrificing performance. Python is also a portable and open-source language with a very supportive and active community base. This is the reason why builing a software in python is fun for everyone including beginners. *Dropbox, BitTorrent, Quora, Instagram, Cinema 4D* are some of the popular applications in python. We encourage you to do a *wikipedia* search for *list of python softwares*. In recent past the interest in python has significantly increased globally and the trend is spreading.

Key features of python language includes:[¶](https://devu.in/tutorial/#Key-features-of-python-language-includes:)

1. Easy and powerful : Python code is very 'English' like and uses words directly from English vocabulary. Being simple however should not be understood as less technical or powerful. For analogy consider a circus artist performing a task so much perfection that it looks simple and comprehensive. Inspiration of *Python*name also came from 'Monte Python' circus and it clearly depicts the idea of simplicity with power.
2. Object-oriented : Object oriented languages provide a reusable code template, like a mini-module defined as *class*. Python has always features classes and instances but it is not strict imposed in the code syntax. So, a python programmer can write python code just with fucnctions which further adds to the simplicity of the language.
3. Interpreted : Interpreted languages are processed line by line at runtime by an interpretor. This saves the effort of precompilation of program and also makes the code more portable. Python is interpreted language but the code may get compiled to *bytecode* contained within \*.pyc file which can then be interpreted by PVM (Python Virtual Machine).
4. Interactive : In python apart from normal mode of code execution i.e, \*.py file, their also exists an interactive mode >>>. This mode of code execution is very simple and direct like human interaction, thus the name interactive mode.
5. Open source : Greatness of python comes without a price tag for both personal and industrial use. Python source code is available as open source under the GNU General Public License (GPL).

Installating python[¶](https://devu.in/tutorial/#Installating-python)

**On windows:**[**¶**](https://devu.in/tutorial/#On-windows:)

* Step 1 : Visit www.python.org/downloads and get the latest compatible version of python.
* Step 2 : Install the \*.msi file.
* Step 3 : Set the path of *python.exe* under system environment variables.

**On macbook:**[**¶**](https://devu.in/tutorial/#On-macbook:)

Macbook comes preloaded with Python2.7, To get Python3 do the following:

* Step 1 : Visit www.python.org/downloads and get the latest compatible version of python.
* Step 2 : Install the \*.dmg file and access it via python3 command.

**On linux:**[**¶**](https://devu.in/tutorial/#On-linux:)

Most linux distributions also comes preloaded with Python2.7 and Python3 preloaded. However following commands may help:

* sudo apt-get install python3;sudo apt-get install idle3;sudo apt-get install python3-pip

First *Hello world* program excution in python[¶](https://devu.in/tutorial/#First-Hello-world-program-excution-in-python)

To verify the installation, we write the *hello world* program in every language. After installing python create a python file \*.py with your preffered name and path. You can choose *notepad, idle, vi editor, PyCharm* etc. to write python code but our recommendation will be to use *Sublime Text* or *Jupyter notebook*. You can also visit [*https://devu.in/tutorial/*](https://devu.in/tutorial/) for help.

Assume the filename for our program is *hello.py* and directory is desktop:

print("Hello World")

To run the code:[¶](https://devu.in/tutorial/#To-run-the-code:)

Navigate to the relevant directory, in our case *desktop*. Then run:

* On Windows: python hello.py
* On Mac or Linux: python3 hello.py

Alternatively one can also open python shell or interactive mode and directly enter:

>>> print("hello world")

Important python programming syntaxes[¶](https://devu.in/tutorial/#Important-python-programming-syntaxes)

Before we begin voyaging into python languages featues, it would be handy to know some important syntaxes of python programming language:

* No semi-colon ';' is required in python code for line seperation or termination. Python code thus is clean and readable and the feature of ';' is seldom used.
* # symbol is used for commenting an instruction.
* Python language is case-sensitive and any change of case results in a distinct object.
* Indentation is the practice is python for defining a code block. Typically, a code block follwing a definition is written after a 'tab space' or indent.
* No type declaration of variables is required and the memory allocation is handled implicitly.
* To access additional library use import keyword and to install use pip install module\_name
* Never make a \*.py file with name conflicting with built-in python library.
* Multi-line string definition is implemented using three pairs of quote (single or double).
* Program control flow moves left to right in general but assignment operator works right to left, nested function defition work inside out and operators have a certain precedance.
* Keywords don't use quotes, string can use single or double quote and function names must be suffixed with ().

Different types of data in python3[¶](https://devu.in/tutorial/#Different-types-of-data-in-python3)

Python3 features Numbers(int,float,complex), String, Lists, Tuples, Dictionary and Sets type of data. long interger type numbers from python2 is no longer required. Below are examples of all python3 data types:

>>> x = 10 # <class 'int'>

>>> x = 10.0 # <class 'float'>

>>> x = 2+3j # <class 'complex'>

>>> x.real # 2.0

>>> x.imag # 3.0

>>> x = "hello @1234#" # <class 'str'>

>>> x = [1,2,3] # <class 'list'>

>>> x = (1,2,3) # <class 'tuple'>

>>> x = {1:2,2:3} # <class 'dict'>

>>> x = {1,2,3} # <class 'set'>

Basic Input/Output[¶](https://devu.in/tutorial/#Basic-Input/Output)

Data input and output is integral part of programming. Python3 provides *input()* and *eval(input())* functions for taking data and *print()* function to publish them.

input()[**¶**](https://devu.in/tutorial/#input())

In python3 *input()* fuction takes data input from i/o console in the form of string whereas in python2.7 *input()* function was called *raw\_input()*. User can enter any number of digits, alphabets or special characters until the *Enter/Return* key is hit. Usually a prompt message is used with input function as argument, suggesting users on what to enter. The data type is always <string> irrespective of what user enters, so an external type casting might be needed in some cases. Incorrect type casting or user input may lead to ValueError thus an exception handling block may also be needed.

age = input("Enter your age: ")

print(type(age))

<class 'str'> or ValueError when type casting fails

eval(input())[**¶**](https://devu.in/tutorial/#eval(input()))

This function from python3 can take any type of python data but demands string input to be given with quotes. It is very unlikely for enduser to do so, hence it is less popular. The same feature in python2 is available via *input()* function.

print()[**¶**](https://devu.in/tutorial/#print())

In python3 *print()* is used to print any value on i/o console. In python2.7 *print* keyword is used for the same.

* *print()* can evaluate expressions and print the resultant value.
* *print()* does not perform any formatting of value, if required string formatting can be used.
* *print()* has one newline \n character already included in it's definition as end character and a whitespace ' ' as value seperator.

print(value,sep=" ",end="\n")

**Function definition and argument values as input/output**[**¶**](https://devu.in/tutorial/#Function-definition-and-argument-values-as-input/output)

The *input()* function is seemingly easy way of taking user data but the overhead of type casting can make it difficult at times. Taking sequence output from sequence is also very difficult and requires external evaluation. Functions although used for struture and reusabiliy in program, can also be used for just agrument passing with the need for type casting. Observe below examples doing the same task of adding two numbers:

def function(a,b): # known variables

... # more instructions is required

return a+b # unknown variable/expression

result = function(2,3) # known values

print(result) # unknown variable

5

**Variables & operators**[**¶**](https://devu.in/tutorial/#Variables-&-operators)

Variables[¶](https://devu.in/tutorial/#Variables)

Variables are tags associated with data. These identifier tags are needed since data is not sequentially stored in memory. Operating system puts data in random memory location using *hashing algorithms* to boost multi-tasking and memory management. On the other hand, when we assign or retrieve data it is not processed immediately and independently. Thus variables are used as reference. Variable names are also more human friendly compared to memory indexes.

**Variable numenclature**[**¶**](https://devu.in/tutorial/#Variable-numenclature)

Variable names in python are case-sensitive words.

Nomenclature rules:[¶](https://devu.in/tutorial/#Nomenclature-rules:)

* Rule 1: Variable name should contain only alpha-numeric characters and underscore, i.e, [A-Za-z0-9]
* Rule 2: Variable name sould begin with only alphabets or underscore, i.e, [^A-Za-z]
* Rule 3: Variable name should not be same as a keyword in that python version

Conventions:[¶](https://devu.in/tutorial/#Conventions:)

* Variable names should preferably be in lower case: abc
* Function names are defined with camelcase suffixing '()': myFunction()
* Class names are preferably written in title case: MyFirstClass

**Python3 keyword list:**[**¶**](https://devu.in/tutorial/#Python3-keyword-list:)

You can check keywords quickly in python shell by running:

>>> import keyword

>>> keyword.kwlist

['False', 'None', 'True', 'and',

'as', 'assert', 'break', 'class',

'continue', 'def', 'del', 'elif',

'else', 'except', 'finally', 'for',

'from', 'global', 'if', 'import', 'in',

'is', 'lambda', 'nonlocal', 'not',

'or', 'pass', 'raise', 'return',

'try', 'while', 'with', 'yield']

**Implicit declaration**[**¶**](https://devu.in/tutorial/#Implicit-declaration)

In python variables need not be declaration explicitly. Also, the memory allocation for different data types is implicitly managed.

For example, compare the syntax of assigning variables in three popular languages:

| C | Java | Python |
| --- | --- | --- |
| int x;x = 0; | int x=0; | x=0 |

**Dynamically declared**[**¶**](https://devu.in/tutorial/#Dynamically-declared)

The variables introduced in python program need not bind to a type immediately, upon interpretation they take the form of the data they refer.

def fun(a,b):

return a+b

x = fun(1,2)

y = fun(1.5,3)

z = fun("A","B")

**Multiple assignment**[**¶**](https://devu.in/tutorial/#Multiple-assignment)

In python several variables can be assigned with single value or different values simultaneously.

>>> x = y = z = 1

>>> x,y,z = 1,12.5,"hello"

**Strongly typed**[**¶**](https://devu.in/tutorial/#Strongly-typed)

Varibales firmly maintain the type of data they were assigned initially and an explicit type conversion is needed to change the type.

>>> x = "23"

>>> y = int(x)+2

**Variable type**[**¶**](https://devu.in/tutorial/#Variable-type)

type() : This function takes a variable name or data object as an argument and tells it's data type. The primitive data types in python and their type declarations are shown below:

>>> type("hello")

<class 'str'>

**Type casting**[**¶**](https://devu.in/tutorial/#Type-casting)

Sometimes the data declared or obtained is not in the correct type for processing, hence an external type conversion is required. All types of object do not convert to all other type. The compatible type casting formulas are listed below:

* int(x [,base]): Converts x to an integer with base specified if x is a string.
* float(x): Converts x to a floating point number.
* complex(real,imag): Creates a complex number.
* str(x): Converts object x to a string representation.
* repr(x): Converts object x to an expression string.
* tuple(x): Converts x to a tuple.
* list(x): Converts x to a list.
* set(x): Converts x to a set.
* dict(d): Creates a dictionary. d must be a sequence of (key,value) tuples.
* frozenset(s): Converts s to a frozen set.
* chr(x): Converts an integer to a character.
* unichr(x): Converts an integer to a Unicode character.
* ord(x): Converts a single character to its integer value.
* hex(x): Converts an integer to a hexadecimal string.
* oct(x): Converts an integer to an octal string.
* eval(input()): Converts the given input to it's correct type

**Scope of variable**[**¶**](https://devu.in/tutorial/#Scope-of-variable)

The extent of variable definition can be understood with scope. In python variables can have one of the three scopes:

* 1. global 2. local 3. nonlocal
* *global* : The variable defined outside a function whose definition is valid throghout the module is global.
* *local* : The variable defined inside a function whose definition is limited to that function is local.
* *nonlocal*: The variable defined inside a nested function definition is non-local.
* x = 10 # global
* def outer():
* a = 11 # local
* def inner():

b = 20 # nonlocal

**Deleting a variable**[**¶**](https://devu.in/tutorial/#Deleting-a-variable)

*del* keyword is used to delete a variable, after which the variable is garbage collected.

>>> x = 10

>>> del x

**A special variable *\_***[**¶**](https://devu.in/tutorial/#A-special-variable-_)

\_ (underscope) can be used in python interactive mode as last computed value, similar to 'last answer' on calculator.

>>> 1 + 10

>>> \_ \*\* 2

121

Operators[¶](https://devu.in/tutorial/#Operators)

Input data is brought in to program, usually to be operated and published as output. Function and classes can define bigger structures for data processing. However some builtin functions which performs frequent and elementory data transformation or comparision are given shorter symbols, called operators.

Operators perform the opeartions on operands as per their definition. Different types of operators, their meaning and function names are given below:

**Arithamtic operators**[**¶**](https://devu.in/tutorial/#Arithamtic-operators)

| **Operation** | **Syntax** | **Function** |
| --- | --- | --- |
| Addition | a + b | add(a, b) |
| Subtraction | a - b | sub(a, b) |
| Multiplication | a \* b | mul(a, b) |
| Division | a / b | truediv(a, b) |
| Concatenation | seq1 + seq2 | concat(seq1, seq2) |
| Floor Division | a // b | floordiv(a, b) |
| Modulo | a % b | mod(a, b) |
| Exponentiation | a \*\* b | pow(a, b) |
| Unary Negation | - a | neg(a) |
| Matrix Multiplication | a @ b | matmul(a, b) |

**Arithmatic operator precedance**[**¶**](https://devu.in/tutorial/#Arithmatic-operator-precedance)

Highest to lowest priority levels of operator selection preference is listed below. At same level priority is left to right.

()

\*\*

\* / % //

+ -

**Assignment operators**[**¶**](https://devu.in/tutorial/#Assignment-operators)

| **Operation** | **Syntax** | **Comment** |
| --- | --- | --- |
| Assignment | = | Control flow right to left |
| Add AND | += | Incremental |
| Subtract AND | -= | Decremental |
| Multiply AND | \*= | Multiplicative increment |
| Division AND | /= | Divisional assignment |
| Floor division AND | //= | Floor division assignment |
| Modulo AND | %= | Modulo assignment |
| Exponenet AND | \*\*= | Exponential assignment |

**Bitwise operators**[**¶**](https://devu.in/tutorial/#Bitwise-operators)

| **Operation** | **Syntax** | **Function** |
| --- | --- | --- |
| Bitwise And | a & b | and\_(a, b) |
| Bitwise Exclusive Or | a ^ b | xor(a, b) |
| Bitwise Inversion | ~ a | invert(a) |
| Bitwise Or | a | b | or\_(a, b) |
| Left Shift | a << b | lshift(a, b) |  |
| Right Shift | a >> b | rshift(a, b) |  |

**Identity operators**[**¶**](https://devu.in/tutorial/#Identity-operators)

| **Operation** | **Syntax** | **Function** |
| --- | --- | --- |
| Identity | a is b | is\_(a, b) |

**Membership operators**[**¶**](https://devu.in/tutorial/#Membership-operators)

| **Operation** | **Syntax** | **Function** |
| --- | --- | --- |
| Containment | obj in seq | contains(seq, obj) |

**Relational operators**[**¶**](https://devu.in/tutorial/#Relational-operators)

| **Operation** | **Syntax** | **Function** |
| --- | --- | --- |
| Ordering | a < b | lt(a, b) |
| Ordering | a <= b | le(a, b) |
| Equality | a == b | eq(a, b) |
| Ineqality | a != b | ne(a, b) |
| Ordering | a >= b | ge(a, b) |
| Ordering | a > b | gt(a, b) |

**Control flow statements**[**¶**](https://devu.in/tutorial/#Control-flow-statements)

It is important to make decision in programming on how your code will be looked at. You may be seletive, iterative or repetitive with the code statements. Python provides us some control flow statements as mantioned below to do this task.

* *if-elif-else* ladder to make decisions
* *for* loop for iteration and repetition
* *while* loop for selective repetition

In addition to this it also offers some keywords like *break, continue, pass* for further control.

if-else condition[¶](https://devu.in/tutorial/#if-else-condition)

For decision making if-else condition is the option python provides. To understand it's importance let us look at the example below:

age = int(input("Dear applicant, please enter your age: "))

if age>18:

print("Your application is accepted.")

else:

print("You are underage now. Please come back later.")

In this example the program control does not flow over both the *print()* statements. The selection here is based on a Boolean condition following *if* keyword. If the condition outcome is *True* it selects the statement under *if block* otherwise under *else block*. The same would have been very difficult without it.

Furthermore, if more such conditions participate in the selection process the keyword use is *elif*, which is like *else if*.

Syntax:[¶](https://devu.in/tutorial/#Syntax:)

if Boolean\_condition:

True statement

elif Boolean\_condition:

True statement

elif Boolean\_condition:

True statement

else:

False statement

Note: *else* nevers takes a condition and is not compulsory in the above decision making ladder.

**Ternary operator: A one-liner shortcut to if-else condition**[**¶**](https://devu.in/tutorial/#Ternary-operator:-A-one-liner-shortcut-to-if-else-condition)

Syntax:[¶](https://devu.in/tutorial/#Syntax:)

* if-else: *True if condition else False*
* if-elif-else: *True\_for\_if if cond1 else True\_for\_elif if cond2 else False*

Example: Equilateral triangle[¶](https://devu.in/tutorial/#Example:-Equilateral-triangle)

sideA, sideB, sideC = -3,-3,-3

if sideA == sideB == sideC and sideA>0:

print("It is equilateral")

else:

print("Not an equilateral")

Equilateral triangle with ternary operator[¶](https://devu.in/tutorial/#Equilateral-triangle-with-ternary-operator)

print("Equilateral") if sideA == sideB == sideC and sideA>0 else print("Not equilateral")

for loop[¶](https://devu.in/tutorial/#for-loop)

*for* loop servers two important purposes:

* repeated execution of a code block
* iteration over sequence

for i in range(3):

print(i,'hello')

for i in "hello":

print(i)

for i in range(len("hello")):

print(i,"hello"[i])

**Deep-dive into for loop**[**¶**](https://devu.in/tutorial/#Deep-dive-into-for-loop)

1. *for* is a keyword to define 'for loop'.
2. i is just a variable name which can be replaced with any other varibale not defined in the same scope.
3. *in* is another keyword and membership operator which returns *True* if the operand is present in the sequence, otherwise it return *False*.
4. Last part in the 'for loop' definition is a sequence object like string, list or tuple. *range()* function also defines a sequence object.

Syntax: range(start=0,end=len(sequence),step=1)[¶](https://devu.in/tutorial/#Syntax:-range(start=0,end=len(sequence),step=1))

list(range(1,5))

[0,1,2,3,4]

list(range(1,10,2))

[1,3,5,7,9]

while loop[¶](https://devu.in/tutorial/#while-loop)

It repeats over the code block as long as entry condition is *True*.

Example:

n = 3

i = 0

while i<n:

print(i,"Hello world")

i += 1

Output:

0 Hello world

1 Hello world

2 Hello world

**Infinite loop**[**¶**](https://devu.in/tutorial/#Infinite-loop)

while True:

...

Loop keywords: break and continue[¶](https://devu.in/tutorial/#Loop-keywords:-break-and-continue)

* break : It use used to break the loop at any given condition. As a result, the program control comes out of the loop. In case of nested loop break breaks only the innermost loop.
* continue : It skips the current iteration of loop and rejoins from the next iteration. The program control does not leave the loop here, it merely skips the current iteration.

for i in "Python":

if i=="t":

break

print(i)

P

y

for i in "Python":

if i in "aeiou":

continue

print(i)

P

y

t

h

n

* pass is another keyword that makes no direct change to program control but can be used in place of no definition.
* for i in "Python":
* pass

pass can also be used outside loop but not break and continue

try:

print(1/0)

except:

pass

**Numbers**[**¶**](https://devu.in/tutorial/#Numbers)

Number data type contains numeric values like int, float and complex. *long* type is excluded in python3. Numbers are non mutable and every time you change them a new number object gets created. Numbers can easily be created with avariable assignment and deleted with *del* keyword as shown below:

>>> x = 10

>>> x = 11

>>> del x

Types of numbers[¶](https://devu.in/tutorial/#Types-of-numbers)

* int : It represents positive or negative whole numbers.
* float : It is a real numbers with integer and fraction part. It contains the floating point '.' , hence the name float. can also be represented with E or e indicating the power of 10. Eg. 1.2e3 = 1.2\*10\*\*3 = 1200
* complex : It is of the form a+bj where *a* and *b* are int or float and j is imaginary number which is also the square root of *-1*.

Type casting to number types[¶](https://devu.in/tutorial/#Type-casting-to-number-types)

* *int(value)* : It converts the given value to interger.
* *float(value)* : It converts the given value to floating point number.
* *complex(value)* : It converts to complex number with given value as real part and imaginary part as zero.
* *complex(value1,value2)* : It converts to complex number with given value1 as real part and value2 as imaginary part.

Numeric functions[¶](https://devu.in/tutorial/#Numeric-functions)

* abs(x) : The positive distance between zero and x.
* >>> abs(-7)
* 7
* >>> abs(7)

7

* max(...) : It returns the maximum among given numbers.
* >>> max(345,34,6,7)

345

* min(...) : It returns the minimum among given numbers.
* >>> min(345,34,6,7)

6

* pow(x,y) : It returns y multiplication of x, similar to x\*\*y.
* >>> pow(2,3)

8

*math* module[¶](https://devu.in/tutorial/#math-module)

*math* library in python provides some handy functions for computing numbers. A list of such functions and their help documentations can be availed with:

>>> import math

>>> dir(math)

>>> help(math)

*math* module also defined global constants pi and e and they can be used as math.pi and math.e.

**Frequently used math functions**[**¶**](https://devu.in/tutorial/#Frequently-used-math-functions)

* sqrt(x) : It is used to find the square root of a given number.
* floor(x) : It returns the integer value of given float with fraction part zero, similar to //
* ceil(x) : It returns the next float value with next integer and zero floating number.
* round(x) : It rounds of the fraction point above and equal to 0.5 as next integer and below 0.5 as previous integer.

>>> from math import sqrt

>>> sqrt(16)

4

>>> import math

>>> math.sqrt(16)

4

>>> math.ceil(9.1)

10

>>> math.floor(8.99)

8

>>> math.round(8.5)

9

Random numbers[¶](https://devu.in/tutorial/#Random-numbers)

*random* module in python provides selection in random order. Below examples explain some key properties:

>>> import random

>>> random.random() # Random float x, 0.0 <= x < 1.0

0.37444887175646646

>>> random.uniform(1, 10) # Random float x, 1.0 <= x < 10.0

1.1800146073117523

>>> random.randint(1, 10) # Integer from 1 to 10, endpoints included

7

>>> random.randrange(0, 101, 2) # Even integer from 0 to 100

26

>>> random.choice('abcdefghij') # Choose a random element

'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]

>>> random.shuffle(items)

>>> items

[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5], 3) # Choose 3 elements

[4, 1, 5]

**String**[**¶**](https://devu.in/tutorial/#String)

String is a sequence of characters marked under quotes. One can interchangeably use one or three pairs of quotes, also because their is no character type of data in python. However, three pair of quotes are frequently used for defining multiline string. String is a very likely source of data since *input()* function, most files and databases, web scraping etc. results in string type object.

>>> a = 'string'

>>> b = "string"

>>> c = '''string'''

>>> d = """string"""

>>> a==b==c==d

True

Properties of a string[¶](https://devu.in/tutorial/#Properties-of-a-string)

**Case sensitive**[**¶**](https://devu.in/tutorial/#Case-sensitive)

Python is a case sensitive language and so is string object.

>>> "hello" == "Hello"

False

**Non-mutable**[**¶**](https://devu.in/tutorial/#Non-mutable)

Original string object never chages it's definition. If we apply any function to string object it simply returns a shallow copy of it.

>>> x = "hello"

>>> x.upper()

'Hello'

>>> print(x)

'hello'

**Concatenation**[**¶**](https://devu.in/tutorial/#Concatenation)

Upon adding two string objects they simply are concatenated.

>>> "a" + "bc"

'abc'

**Repetition**[**¶**](https://devu.in/tutorial/#Repetition)

If we multiply a string with an interger, it simply repeats the interger times.

>>> "ha" \* 2

'haha'

**Sequence**[**¶**](https://devu.in/tutorial/#Sequence)

Since string indexes are always in sequence starting from 0 amd -1, it can be called a sequence type object. Sequence type objects exhibit below properties:

Indexing:[¶](https://devu.in/tutorial/#Indexing:)

str[position]

>>> x = "hello"

>>> x[0]

'h'

>>> x[1]

'e'

>>> x[-1]

'o'

>>> x[-2]

'l'

Slicing:[¶](https://devu.in/tutorial/#Slicing:)

str[start=0:end=len(str):step=1]

>>> x = "hello world"

>>> x[1:5]

'ello'

>>> x[::]

'hello world'

Reversing a string[¶](https://devu.in/tutorial/#Reversing-a-string)

>>> x[::-1]

'dlrow olleh'

Pallindrome[¶](https://devu.in/tutorial/#Pallindrome)

>>> x = "hello"

>>> x==x[::-1]

False

>>> x = "madam"

>>> x==x[::-1]

True

String to list[¶](https://devu.in/tutorial/#String-to-list)

>>> x = "India is great"

>>> x.split()

['India','is','great']

Character count[¶](https://devu.in/tutorial/#Character-count)

>>> x = "hello"

>>> len(x)

5

Word count[¶](https://devu.in/tutorial/#Word-count)

>>> x = "India is great"

>>> len(x.split())

3

Sorted string[¶](https://devu.in/tutorial/#Sorted-string)

>>> x = "apple"

>>> sorted(x)

['a','e','l','p','p']

>>> "".join(sorted(x))

'aelpp'

Built-in string functions: dir(str)[¶](https://devu.in/tutorial/#Built-in-string-functions:-dir(str))

* capitalize() - Returns the string with first letter capitalized and the rest lowercased.
* casefold() - Returns a lowercase string, generally used for caseless matching. This is more aggressive than the lower() method.
* center() - Center the string within the specified width with optional fill character.
* count() - Count the non-overlapping occurrence of supplied substring in the string.
* encode() - Return the encoded version of the string as a bytes object.
* endswith() - Returns ture if the string ends with the supplied substring.
* expandtabs() - Return a string where all the tab characters are replaced by the supplied number of spaces.
* find() - Return the index of the first occurrence of supplied substring in the string. Return -1 if not found.
* format() - Format the given string.
* format\_map() - Format the given string.
* index() - Return the index of the first occurrence of supplied substring in the string. Raise ValueError if not found.
* isalnum() - Return true if the string is non-empty and all characters are alphanumeric.
* isalpha() - Return true if the string is non-empty and all characters are alphabetic.
* isdecimal() - Return true if the string is non-empty and all characters are decimal characters.
* isdigit() - Return true if the string is non-empty and all characters are digits.
* isidentifier() - Return true if the string is a valid identifier.
* islower() - Return true if the string has all lowercased characters and at least one is cased character.
* isnumeric() - Return true if the string is non-empty and all characters are numeric.
* isprintable() - Return true if the string is empty or all characters are printable.
* isspace() - Return true if the string is non-empty and all characters are whitespaces.
* istitle() - Return true if the string is non-empty and titlecased.
* isupper() - Return true if the string has all uppercased characters and at least one is cased character.
* join() - Concatenate strings in the provided iterable with separator between them being the string providing this method.
* ljust() - Left justify the string in the provided width with optional fill characters.
* lower() - Return a copy of all lowercased string.
* lstrip() - Return a string with provided leading characters removed.
* maketrans() - Return a translation table.
* partition() - Partition the string at first occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator.
* replace() - Replace all old substrings with new substrings.
* rfind() - Return the index of the last occurrence of supplied substring in the string. Return -1 if not found.
* rindex() - Return the index of the last occurrence of supplied substring in the string. Raise ValueError if not found.
* rjust() - Right justify the string in the provided width with optional fill characters.
* rpartition() - Partition the string at last occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator.
* rsplit() - Return a list of words delimited by the provided subtring. If maximum number of split is specified, it is done from the right.
* rstrip() - Return a string with provided trailing characters removed.
* split() - Return a list of words delimited by the provided subtring. If maximum number of split is specified, it is done from the left.
* splitlines() - Return a list of lines in the string.
* startswith() - Return true if the string starts with the provided substring.
* strip() - Return a string with provided leading and trailing characters removed.
* swapcase() - Return a string with lowercase characters converted to uppercase and vice versa.
* title() - Return a title (first character of each word capitalized, others lowercased) cased string.
* translate() - Return a copy of string that has been mapped according to the provided map.
* upper() - Return a copy of all uppercased string.
* zfill() - Return a numeric string left filled with zeros in the provided width.

**List**[**¶**](https://devu.in/tutorial/#List)

List is a mutable collection of values seperated by commas ',' and enclosed with square bracket []. It has sequential and unique inexes but unordered and redundant values.

>>> l = `[1,'A',2,4,5.5,'Apple']`

>>> type(l)

<class 'list'>

>>> a = [1,2,3]

>>> b = [1,2,3,]

>>> c = [

1,

2,

3,

]

>>> d = [

1,

2,

3

]

>>> a == b == c == d

True

Properties of list[¶](https://devu.in/tutorial/#Properties-of-list)

* List indexes are sequential starting from 0 or -1.
* >>> l = ['A','B','C']
* >>> l[0]
* 'A'
* >>> l[-1]

'C'

* List may contain duplicate values.
* >>> l = [1,2,3,2,1]
* >>> sorted(set(l))

[1,2,3]

* List values can be unordered.

>>> l = [1,3,2,4]

* List is mutable or changeable.
* >>> l = [1,2,3]
* >>> l.append(4)
* >>> l

[1,2,3,4]

* List can be concatenated.
* >>> [1,2,3] + [4,5]

[1,2,3,4,5]

* List objects repeat when multiplied with integer.
* >>> [1,2] \* 3

[1,2,1,2,1,2]

* List is iterable
* >>> for i in range(10): # [0,1,2,3,4,5,6,7,8,9]

...

**Built-in functions for a list object : dir(list)**[**¶**](https://devu.in/tutorial/#Built-in-functions-for-a-list-object-:-dir(list))

* CRUD operations wise classification of list functions
* CREATE : 'append', 'extend', 'insert', 'copy'
* Read : 'count', 'index'
* Update : 'reverse', 'sort'

Delete : 'clear', 'pop', 'remove'

* append(value): It puts the given value at the end of the list object
* extend(value): It concatenates another list at the end of the first list.
* insert(index,value): It assigns the given value at a given index with the list. First parameter is index, second is value.
* copy(): It returns a shallow/seperate copy of list object.
* count(value): It returns the count of a given value within the list.
* index(value): It returns the first position of a given value within the list.
* sort(): It sorts the given list in ascending order. reverse=True allows the list to sorted in descending order.
* reverse(): It reverses the given list horizontally.
* pop(index): It pops the value at given index, default index is -1
* remove(value): It removes first occurance of a given value.
* clear(): It clears all values from the list making it empty.

**Additional list attributes**[**¶**](https://devu.in/tutorial/#Additional-list-attributes)

* max(list) : It gives the maximum value from a given list of numbers.
* min(list) : It gives the minimum value from a given list of numbers.
* sum(list) : It gives the sum of values from a given list of numbers.
* random.shuffle() : It shuffles the list values randomly. Make sure to import random module.
* enumerate(list) : It alows us to loop over a list with automatic counter.
* for index,value in enumerate():

...

deleting list[¶](https://devu.in/tutorial/#deleting-list)

* *clear()* function only deletes list values to delete the complete list object we can use del keyword.

Incrementing list[¶](https://devu.in/tutorial/#Incrementing-list)

* list += [value] can sometimes be used instead of list.append(value).

Comparing list[¶](https://devu.in/tutorial/#Comparing-list)

* Lists are compared index-wise.
* >>> [2,3,4] > [1,100,1000]

True

List comprehension[¶](https://devu.in/tutorial/#List-comprehension)

It is a one liner shortcut for creating a new list, starting from an existing sequence. It is used for mapping and filtering a sequence.

Syntax: [expression for loop if condition]

>>> l = [1,2,3,4,5]

>>> [i\*\*2 for i in l]

[1,4,9,16,25]

>>> [i\*\*2 for i in l if i>3]

[16,25]

>>> [i for i in l if i>3]

[4,5]

>>> [i.upper() for i in "hello" if i in "aeiouAEIOU"]

['E','O']

>>> "".join([str(i) for i in [1,2,3,4]])

'1234'

**Tuple**[**¶**](https://devu.in/tutorial/#Tuple)

It is a read-only/non-mutable list. It can also be redundant, unordered and sequential collection like list.

>>> t = 1,2,2,'A',5,4,1,-90.8

>>> t = (1,2,2,'A',5,4,1,-90.8)

>>> print(type(t))

<class 'tuple'>

>>> x = (1,2,5)

>>> x = (1,2,5,)

>>> x = (

1,

2,

5,

)

>>> x = 1,2,5

Properties of tuple[¶](https://devu.in/tutorial/#Properties-of-tuple)

* Tuple indexes are sequential starting from 0 or -1.
* >>> l = ['A','B','C']
* >>> l[0]
* 'A'
* >>> l[-1]

'C'

* Tuple may contain duplicate values.
* >>> l = [1,2,3,2,1]
* >>> sorted(set(l))

[1,2,3]

* Tuple values can be unordered.

>>> l = [1,3,2,4]

* Tuple can be concatenated.
* >>> [1,2,3] + [4,5]

[1,2,3,4,5]

* Tuple objects repeat when multiplied with integer.
* >>> [1,2] \* 3

[1,2,1,2,1,2]

* Tuple is iterable
* >>> for i in range(10): # [0,1,2,3,4,5,6,7,8,9]

...

* Tuple is non-mutable unlike list.
* >>> t = (1,2,3,4)
* >>> t += 5,
* >>> t

(1,2,3,4,5)

* Tuple can be used as dictionary keys unlike list.

>>> d = {('a','b'):100}

* Tuple of arguments can be used to overload functions
* def function(\*args):

...

#### Type casting to tuple

tuple(list)

#### Sorting a tuple

tuple(sorted(tuple))

#### One value tuple

>>> t = (1,)

>>> t = 1,

Built-in functions in tuple: dir(tuple)[¶](https://devu.in/tutorial/#Built-in-functions-in-tuple:-dir(tuple))

* count(value) : It gives the count of individual tuple item, juts like list.
* index(value) : Just like list it gives the index for a value in tuple or IndexError

**Dictionary**[**¶**](https://devu.in/tutorial/#Dictionary)

It is an non-sequential, unordered, redundant and mutable collection as key:value pairs. Keys are always unique but values need not be unique.

List vs Dictionary[¶](https://devu.in/tutorial/#List-vs-Dictionary)

* Unlike a list, dictionary is not a sequence.
* 0 1 2 3 4 5
* list = [1,'A',2,3,2,8]

dict = {0:1,1:'A',2:2,3:3,4:2,5:8.7}

* List has numeric index only in sequential order starting from zero.
* Dictionary is has keys instead of index which can be numeric, string or tuple. They are unordered but they are always unique.
* List to dictionary transition is mainly due to the improved accesibility of data items.

Properties of dictionary[¶](https://devu.in/tutorial/#Properties-of-dictionary)

* Dictionary is a key:value pair collection for mapping information.
* Keys are unique and non-mutable(numbers, string and tuple)
* Values can be rudundant and of any dype.
* Dictionary is not a sequence but *d.keys()*, *d.values()* and *d.items()* are iterable.
* Dictionary is mutable like list
* >>> d = {'A':'Apple'}
* >>> d['A'] = "Aeroplane"
* >>> d
* {'A':'Aeroplane'}
* >>> d.update({'A':'Apple'})
* >>> d

{'A':'Apple'}

Built-in functions for a dictionary object: *dir(dict)*[¶](https://devu.in/tutorial/#Built-in-functions-for-a-dictionary-object:-dir(dict))

CREATE : 'setdefault', 'fromkeys', 'copy'

READ : 'get', 'items', 'keys', 'values'

UPDATE : 'update'

DELETE : 'clear','pop', 'popitem'

* setdefault(key,value) : It inserts a new key value pair to dictionary.
* fromkeys(sequence,value=None) : It gets the keys for a dictionary from a list/tuple. Values by default is None or same for all the keys when given.
* get(key,alt.value) : It returns the value for a given key or None or alternate value if given.
* keys() : It returns iterable sequence of keys without sequence.
* values() : It returns iterable sequence of values without sequence.
* items() : It returns iterable sequence of key:value pairs without sequence.
* update() : updates the value for a given key.
* pop(key) : It pops the value for a given key.
* popitem() : delets a random key value pair.
* clear() : It delets all key:value pairs.
* copy() : It creates a shallow/seperate copy for the object.

dict comprehension[¶](https://devu.in/tutorial/#dict-comprehension)

It is a one liner shortcut like list comprehension to create a new dict object from a given sequence.

Syntax:

- a) {key\_exp:value\_exp for key,value in dict\_object}

- b) {key\_exp:value\_exp for key,value in dict\_object if cond}

>>> l = [1,2,3,4,5]

>>> [i\*\*2 for i in l]

[1, 4, 9, 16, 25]

>>>

>>> d = {1:10,2:20,3:30}

>>> {k:v for k,v in d.items()}

{1: 10, 2: 20, 3: 30}

>>>

>>> {k\*\*3:v\*\*2 for k,v in d.items()}

{8: 400, 1: 100, 27: 900}

>>>

Swapping the key:value pair of a dictionary without duplicate[¶](https://devu.in/tutorial/#Swapping-the-key:value-pair-of-a-dictionary-without-duplicate)

>>> {v:k for k,v in d.items()}

Swapping the key:value pair of a dictionary with duplicate[¶](https://devu.in/tutorial/#Swapping-the-key:value-pair-of-a-dictionary-with-duplicate)

>>> {v:[i for i in d.keys() if d[i]==v] for k,v in d.items()}

**zip list of keys and values to a dictionary**[**¶**](https://devu.in/tutorial/#zip-list-of-keys-and-values-to-a-dictionary)

>>> x = ['A','B','D','C','E']

>>> y = [1,2,3,4,5]

>>> dict(zip(x,y))

{'A': 1, 'C': 4, 'E': 5, 'B': 2, 'D': 3}

**Set**[**¶**](https://devu.in/tutorial/#Set)

Set is mutable and unique collection. It resembles dictionary key in properties.

>>> s = {1,2,3,4,5,6}

>>> type(s)

<class 'set'>

Empty set[¶](https://devu.in/tutorial/#Empty-set)

>>> s = set()

>>> type(s)

<class 'set'>

>>> len(s)

0

Properties of set[¶](https://devu.in/tutorial/#Properties-of-set)

* Like list, set has unordered values.
* Unlike list, set in non-sequential and does not support indexing.
* Unlike list, set is non-redundant and does not allow duplicate values. set() function can be used to remove duplicates form a list.
* >>> l = [1,2,3,2,3,1]
* >>> list(set(l))

[1, 2, 3]

* Like list and dictionary set is also mutable.

*frozenset*[¶](https://devu.in/tutorial/#frozenset)

It is a non-mutable version of set.

frozenset(set\_object)

Built-in functions of set: dir(set)[¶](https://devu.in/tutorial/#Built-in-functions-of-set:-dir(set))

* add : adds a new value to the set
* clear : deletes all set values
* copy : makes a shallow copy of set
* difference or -: finds the unique items is first set compared to the second set.
* difference\_update : finds the unique items is first set compared to the second set and updates the vfirst set
* discard() : deletes a value from the set if present, returns None if not
* intersection or & : finds mutual data in two given sets
* intersection\_update : finds mutual data in two given sets and updates the first one
* isdisjoint : It means mutually exclusive. Two sets are disjoint when they have nothing in common.
* issubset : It is a booelan opearation for finding if all the items is the given data set exists in some other data set.
* issuperset : It is a booelan opearation for finding if the given data set contains all elements from the other data set.
* pop() : pops a random value from the set, like dict.popitem()
* remove() : deletes a value from the set if present, returns KeyError if not
* symmetric\_difference or ^: It is the sum of difference from both sides, i.e, non-mutual or uncommon data sets
* symmetric\_difference\_update: It finds the symmetric difference and updates value to first set object.
* union or |: joins the data sets with only one occurance of duplicate. It is like 'or' operation.
* update() : updates the set with values from a sequence.

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**#**[**¶**](https://devu.in/tutorial/##)

**Exception Handling**[**¶**](https://devu.in/tutorial/#Exception-Handling)

What are exceptions?[¶](https://devu.in/tutorial/#What-are-exceptions?)

Exceptions are unplanned events that interrupts the flow of program control. Exceptions may occur due to syntax or logical error. Stimulus for an exception may be the end-user input, vulnerable program logic, environment etc. Exceptions can also be planted by using 'raise' and 'assert' keywords.

Traceback (most recent call last):

File ..., line 1, ...

NameError: name '...' is not defined

*'Traceback (most recent call last)'* is always the first line of exception messages. It is followed by *line number* and *line content*. And finally by the *type of exception*followed by possible explanation of the exception.

Why should exception be handled?[¶](https://devu.in/tutorial/#Why-should-exception-be-handled?)

Traceback messages are meant for developer of the app or testers but not for the end user. End user might fail to understand or act upon it. Even when the end user are very smart the can use it for malicious hacking practices. Irrespective of the source of exception, a developer must be accountable for all the exceptions in its code. This can be achieved by handling exceptions.

try:

pass

except Exception as e:

raise

else:

pass

finally:

pass

* Keywords like try, except, else and finally are used in exception handling. try-except block is similar to if-else block to some extent and we can also have multiple and nested try blocks in a program.
* try-except block We can write the suspected lines of code under the 'try' block. If any exception occurs in the 'try' block, program control goes to the 'except' block. Any python code can be written within the exception block but we mostly settle for an exception message (on console or inside log). since a code failure in except block may lead to a traceback message.
* finally block Code written in the finally block are always executed when the control reaches the try block. Things like closing a file, database connection or socket objects fits right in the 'finally' block because these operations needs to be performed both in pass and fail cases.
* else block Code inside try block can sometimes be spread across 'else' block. 'else' block is executed when the 'try' block is True.
* Control flow in try-except-else-finally
* ------ try ------
* | |
* True False
* | |
* else except
* | |

--- finally -----

**Points to remember**[**¶**](https://devu.in/tutorial/#Points-to-remember)

* Exception handling does not guarantee exception prevention, they only act as contingency plan
* *try* block without *except* block will give SyntaxError
* default except block must be the last
* *finally* and *else* block are not mandatory but if used it should be in the sequence of try-except-else-finally
* *try-finally* can also be used for exception handling
* *pass* keyword in the exception block hints at ignoring the occured exception
* print("Before")
* try:
* print(1/0)
* except:
* pass

print("After")

Logging[¶](https://devu.in/tutorial/#Logging)

Logging module in python provides ability to capture and classify the exceptions. A .log file similar to .txt file is used for capturing the error log. logging module can also be used to print warning, error or critical exceptions.

Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

import logging

logging.warning("Something went wrong")

logging.warning("Something went wrong")

logging.warning("Something went wrong")

**Levels of logging**[**¶**](https://devu.in/tutorial/#Levels-of-logging)

* Exceptions can classified under various severity levels.

Level Code

logging.DEBUG 10

logging.INFO 20

logging.WARNING 30

logging.ERROR 40

logging.CRITICAL 50

| **Type of logging** | **Description** |
| --- | --- |
| DEBUG | Detailed information, typically of interest only when diagnosing problems. |
| INFO | Confirmation that things are working as expected. |
| WARNING | An indication that something unexpected happened, or may happen in near future |
| ERROR | Due to a more serious problem, the software has not been able to perform some function. |
| CRITICAL | A serious error, indicating that the program itself may be unable to continue running. |

try:

print(1/0)

except Exception as e:

logging.basicConfig(filename="xyz.log",level=50)

logging.critical(str(e))

**Causing an exceptions**[**¶**](https://devu.in/tutorial/#Causing-an-exceptions)

If required developer can also cause exceptions with keywords like 'raise' and 'assert'.

* raise : By default it causes *RuntimeError*
* raise 'ExceptionName' : It can be used to raise an specific error.
* class UserDefinedError(Exception):pass

raise UserDefinedError

* The above syntax can be used to define a new exception type and raise it
* assert : It works like a mandatory condition. If False it raises *'AssertionError'*.
* assert condition,"error message" : This syntax is used as a checkpoint. In the event of condition being False it raises 'AssertionError' with given message.
* assert is also called *'raise if not'*, simply because it raises exception if condition is not *True*.

Python exception hierarchy and warnings[¶](https://devu.in/tutorial/#Python-exception-hierarchy-and-warnings)

BaseException

SystemExit

KeyboardInterrupt

GeneratorExit

Exception

StopIteration

StopAsyncIteration

ArithmeticError

FloatingPointError

OverflowError

ZeroDivisionError

AssertionError

AttributeError

BufferError

EOFError

ImportError

ModuleNotFoundError

LookupError

IndexError

KeyError

MemoryError

NameError

UnboundLocalError

OSError

BlockingIOError

ChildProcessError

ConnectionError

BrokenPipeError

ConnectionAbortedError

ConnectionRefusedError

ConnectionResetError

FileExistsError

FileNotFoundError

InterruptedError

IsADirectoryError

NotADirectoryError

PermissionError

ProcessLookupError

TimeoutError

ReferenceError

RuntimeError

NotImplementedError

RecursionError

SyntaxError

IndentationError

TabError

SystemError

TypeError

ValueError

UnicodeError

UnicodeDecodeError

UnicodeEncodeError

UnicodeTranslateError

Warning

DeprecationWarning

PendingDeprecationWarning

RuntimeWarning

SyntaxWarning

UserWarning

FutureWarning

ImportWarning

UnicodeWarning

BytesWarning

ResourceWarning

**File handling**[**¶**](https://devu.in/tutorial/#File-handling)

File is a data storage unit that keeps data in non-volatile manner. To work with a file we need to open it first, then we can perform read or write operations, after which we close the file and freeup the system resources tied up with file operations.

As part of standard input/output operations, we have seen the functions input() and print(). For file handling, operations take place in following manner:

* Open a file
* Read or write
* Close the file
* Opening a file

We are required to open a file before any file operation with the help of open() function.

* open() : It opens a file and return a data stream. It raises IOError upon failure.

with open(file,mode='r') as f:

...

File name and path[¶](https://devu.in/tutorial/#File-name-and-path)

* filename : filename is either a text or byte string given a name with a particular extension. We are required to mention the absolute file path if the data file and python file are not in the same directory.

**File modes**[**¶**](https://devu.in/tutorial/#File-modes)

* mode : It is an optional string that specifies the mode in which the file is opened. Read 'r' is the default mode.

| **Mode** | **Description** |
| --- | --- |
| 'r' | (default) read mode |
| 'w' | write mode where file is truncated first and created if not found |
| 'a' | append mode where content is added in the end of file and file is created if not found |
| 'x' | create a file and open it for writing |
| 't' | text mode (default with read, write or append) |
| 'b' | binary mode (read, write or append) |
| '+' | updating a file (read, write or append) |

| **File attributes** | **File attribute Description** |
| --- | --- |
| f.name | returns the filename as string |
| f.mode | returns the mode, default is 'r' |
| f.encoding | returns file encoding, default is "UTF-8" |
| f.errors | default strict means raise an exception in case of an encoding error |

Binary vs text files[¶](https://devu.in/tutorial/#Binary-vs-text-files)

* Binary files are buffered in fixed-size blocks typically 4096 or 8192 bytes. During file handing we can open images, pdf etc in binary mode.
* Text files however uses line buffering. .csv, .txt etc. can be understood as text files.

Closing a file[¶](https://devu.in/tutorial/#Closing-a-file)

Closing a file object after file operations is important to release the system resources tied up with the file object.

* close() : closes the file object.
* closed : It is a boolean attribute that returns True if the file is closed, otherwise False.
* *with* keyword : *with* is a keyword that can be attached to file object. File object definition containing with keyword can have a block of nested for file operations. Upon closure of the nested block with keyword automatically closes the file object. Even if an error is raise in the nested block under with keyword, it still guarantees the closing of file.

Reading a file:[¶](https://devu.in/tutorial/#Reading-a-file:)

We can read a file content in following ways:

with open(file,'r'):

...

* 'r' mode is applied to a file object by default is nothing is mentiontioned. Read mode sets the file cursor in the beginning index of '0'.
* readable() : returns True if file is opened in read related modes, otherwise False.
* read() : returns the entire file content as single string object, if file is pre
* readline() : returns single line of a file from current cursor location.
* readlines() : returns a list object with lines of file as string.

Writing into a file[¶](https://devu.in/tutorial/#Writing-into-a-file)

Only string sequences or stream can be written to a text file in following manners:

with open(file,'w'):

...

* 'w' mode specifies truncate and write
* 'a' mode specifies appending in the end
* writeable() : returns True if file is opened in write related modes, otherwise False.
* write() : It takes a string sequence and writes it in a file.
* writelines() : It takes list of string lines to write at once.

Other file methods[¶](https://devu.in/tutorial/#Other-file-methods)

* seek(bytes,offset) : changes cursor position by given bytes offset. Offset is 0 by default for beginning of file, 1 for current cursor location and 2 for end of file. Non-zero seek in python3.x can be achieved by opened the file in binary mode.
* tell() : tells the cursor position index.

**Connecting Python to SQLite3 database**[**¶**](https://devu.in/tutorial/#Connecting-Python-to-SQLite3-database)

Python has a built-in support for sqlite3 database. Just begin using with import sqlite3. A connection to existing database is firstly created with connect() function. The good news is, if the database doesnot exists, it gets created. Next one can define cursor and execute the query by using cursor() and execute() functions. Finally the connection object is closed with close() function.

* Step 1: Make database connection object
* import sqlite3
* connection = sqlite3.connect("test.db")

print("Connected to database successfully.")

* Step 2: Define cursor with connection object

cursor = connection.cursor()

* Step 3: Execute database query for CREATE, READ, UPDATE or DELETE

cursor.execute('''

CREATE TABLE test\_table(

name CHAR(30) NOT NULL,

age int NOT NULL,

salary REAL,

address TEXT

)

''')

print("Query OK")

Python datatype vs mysql datatype[¶](https://devu.in/tutorial/#Python-datatype-vs-mysql-datatype)

- str --> TEXT

- float --> REAL

- int --> INT

- str --> CHAR(max\_length)

cursor.execute("INSERT INTO test\_table VALUES('Person1',27,34250.56,'Bangalore')")

print("Query OK")

* commit() is used to finally deploy the new writing operations, if deleted we can rollback() changes.

conn.commit()

cursor.execute("SELECT \* FROM test\_table")

for row in cursor.fetchall():

print(row)

print("Query OK")

cursor.execute("DELETE FROM test\_table where name='Person1'")

print("Query OK")

* rollback()

conn.rollback()

* Step 4: Closing the connection
* conn.close()

print("Database connection closed. Bye!")

**Regular expression**[**¶**](https://devu.in/tutorial/#Regular-expression)

re is used for filtering data based on pattern.

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways. Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

Pattern making special characters:[¶](https://devu.in/tutorial/#Pattern-making-special-characters:)

"." Matches any character except a newline.

"^" Matches the start of the string.

"\*" Matches 0 or more (greedy) repetitions of the preceding RE.

Greedy means that it will match as many repetitions as possible.

"+" Matches 1 or more (greedy) repetitions of the preceding RE.

"?" Matches 0 or 1 (greedy) of the preceding RE.

\*?,+?,?? Non-greedy versions of the previous three special characters.

{m,n} Matches from m to n repetitions of the preceding RE.

{m,n}? Non-greedy version of the above.

"\\" Either escapes special characters or signals a special sequence.

[] Indicates a set of characters.

A "^" as the first character indicates a complementing set.

"|" A|B, creates an RE that will match either A or B.

(...) Matches the RE inside the parentheses.

The contents can be retrieved or matched later in the string.

(?aiLmsux) Set the A, I, L, M, S, U, or X flag for the RE (see below).

(?:...) Non-grouping version of regular parentheses.

(?P<name>...) The substring matched by the group is accessible by name.

(?P=name) Matches the text matched earlier by the group named name.

(?#...) A comment; ignored.

(?=...) Matches if ... matches next, but doesn't consume the string.

(?!...) Matches if ... doesn't match next.

(?<=...) Matches if preceded by ... (must be fixed length).

(?<!...) Matches if not preceded by ... (must be fixed length).

(?(id/name)yes|no) Matches yes pattern if the group with id/name matched,

the (optional) no pattern otherwise.

"$" Matches the end of the string or just before the newline at

the end of the string.

re functions[¶](https://devu.in/tutorial/#re-functions)

import re

a) re.search(pattern, string\_datasource, flag)

finds the first occurance of the pattern or gives AttributeError

b) re.match(pattern, string\_datasource, flag):

searches pattern in the beginning of the string

c) re.findall(pattern, string\_datasource, flag) :

finds all occurances of the pattern as list object.

d) re.sub(old\_pattern, new\_pattern, string\_datasource, count= maxcount):

It replaces the old\_patterns with new ones

e) re.compile(pattern) : It defines a pattern

re flags[¶](https://devu.in/tutorial/#re-flags)

re.I : Ignorecase

re.X : Ignore whitespaces and comment

re.M : Multiline search

re.S : DOTALL adds newline ('\n') to the '.' definition

re.A : Ascii

re.U : Unicode

**Classes and Objects**[**¶**](https://devu.in/tutorial/#Classes-and-Objects)

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state

Terminologies[¶](https://devu.in/tutorial/#Terminologies)

* Classes: A class is a mini-module with it's namespace structure. Classes are created as reusable code templates.
* Obejcts: An instance of a class that calls the class template with some arguments.
* init(): is the initializer method of the class because it is the initial/first function to execute when class is instantiated. It is called automatically when the instance of the class gets created. It houses instance variables which can be shared by methods inside the class. It is not mandatory to define init() but if defined it is conventionally the first function in the class.
* self: It represents the current object of the class. It can be understood as an explicit marker for object inside class template. *self* can be seen as the first parameter for every member function of the class and in the representation of class or instance variables. *self* is not a keyword so can be replaced with something it. Although use of self is highly recommended.

**Attribute of a class**[**¶**](https://devu.in/tutorial/#Attribute-of-a-class)

It refers to the data members and member functions defined inside a class.

* str : String representation of class object.
* doc : Class documentation string.
* name : Name of the class. When called from within the module it is main
* bases : Represents base classes for a sub class. The parent class of all classes is Object class.
* dict : A dictionary of docstring, bases and attributes of a class.
* init : Commonly referred as constructor.
* del : Commonly referred as destructor.

**function vs class definition**[**¶**](https://devu.in/tutorial/#function-vs-class-definition)

def keyword is used to define a function whereas class keyword is used to define a class.

def greeting():

print("Hello World")

x = greeting()

class Greeting:

print("Hello World")

obj = Greeting()

A calculator class[¶](https://devu.in/tutorial/#A-calculator-class)

class Calc:

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

def add(self):

return self.a + self.b

def sub(self):

return self.a - self.b

def mul(self):

return self.a \* self.b

def div(self):

return self.a / self.b

obj = Calc(1,2)

print(obj.add())

print(obj.sub())

print(obj.mul())

print(obj.div())

Static vs instance variables[¶](https://devu.in/tutorial/#Static-vs-instance-variables)

Static variables are the variables inside class that do not change with different objects of the class. They are typically defined outside any function of the class.

Instance variables are the variables defined inside the \_\_init\_\_(). They can change their value with differernt class objects(instances).

class Dogs:

instruction = [] # bad use of static/class variable

def \_\_init\_\_(self,name):

self.name = name

# self.instruction = [] # good use of instance variable

def do(self,command):

return self.instruction.append(command)

dog1 = Dogs("Tom")

dog1.do("roll over")

dog2 = Dogs("Fido")

dog2.do("play dead")

print(dog1.name, dog1.instruction)

print(dog2.name, dog2.instruction)

Output[¶](https://devu.in/tutorial/#Output)

Tom ['roll over', 'play dead']

Fido ['roll over', 'play dead']

class Dogs:

# instruction = [] # bad use of static/class variable

def \_\_init\_\_(self,name):

self.name = name

self.instruction = [] # good use of instance variable

def do(self,command):

return self.instruction.append(command)

dog1 = Dogs("Tom")

dog1.do("roll over")

dog2 = Dogs("Fido")

dog2.do("play dead")

print(dog1.name, dog1.instruction)

print(dog2.name, dog2.instruction)

Output[¶](https://devu.in/tutorial/#Output)

Tom ['roll over']

Fido ['play dead']

Inheritance[¶](https://devu.in/tutorial/#Inheritance)

It means sharing of class attributes, among different classes. It has two major types:

* 1. Simple Inheritance
  2. Multiple inheritance
  3. Multilevel inheritance

class A:

def fun(self):

return "I am a function f1 from class A"

class B(A): # with inheritance

pass

obj = B()

obj.fun()

Output[¶](https://devu.in/tutorial/#Output)

'I am a function f1 from class A'

Multiple inheritance[¶](https://devu.in/tutorial/#Multiple-inheritance)

It is an inheritance scenarion where a class gets attributes from multiple different classes. It is not so common among object oriented languages.

class A:

def f1(self):

return "I am a function from class A"

class B:

def f2(self):

return "I am a function from class B"

class C(A,B):

def f3(self):

return "I am a function from class C"

obj = C()

print(obj.f3())

print(obj.f1())

print(obj.f2())

Output[¶](https://devu.in/tutorial/#Output)

I am a function from class C

I am a function from class A

I am a function from class B

Multilevel inheritance[¶](https://devu.in/tutorial/#Multilevel-inheritance)

It is an inheritance scenarion where a class gets attributes from its parent or super class and their parents or super classes. It is common among most object oriented languages.

class A:

def f1(self):

return "I am a function from class A"

class B(A):

def f2(self):

return "I am a function from class B"

class C(B):

def f3(self):

return "I am a function from class C"

obj = C()

print(obj.f3())

print(obj.f1())

print(obj.f2())

Output[¶](https://devu.in/tutorial/#Output)

I am a function from class C

I am a function from class A

I am a function from class B

Overriding[¶](https://devu.in/tutorial/#Overriding)

Sometimes inheritance leads to multiple occurances of same named objects within a class namespace. In such scenario the selection is made not with the implementation of object but with their position during inheritance. This preference in inheritance for resolving namespace collision is called overriding.

Three rules of overriding:

* Prefer child definition over parent defition.
* In multiple inheritance, the priority decreases left to right in the order of mention of class names for inheritance.
* In multilevel inheritance the priority flows from immediate level parent to distant parent class.

**Simple inheritance and overriding**[**¶**](https://devu.in/tutorial/#Simple-inheritance-and-overriding)

class A:

def fun(self):

return "I am a function from class A"

class B(A):

def fun(self):

return "I am a function from class B"

obj = B()

print(obj.fun())

Output[¶](https://devu.in/tutorial/#Output)

'I am a function from class B'

**Multiple inheritance & overriding**[**¶**](https://devu.in/tutorial/#Multiple-inheritance-&-overriding)

class A:

def fun(self):

return "I am a function from class A"

class B:

def fun(self):

return "I am a function from class B"

class C(B,A): # Priority L-R

pass

obj = C()

print(obj.fun())

Output[¶](https://devu.in/tutorial/#Output)

I am a function from class B

**Multilevel inheritance & overriding**[**¶**](https://devu.in/tutorial/#Multilevel-inheritance-&-overriding)

class A:

def fun(self):

return "I am a function from class A"

class B(A):

def fun(self):

return "I am a function from class B"

class C(B):

pass

obj = C()

print(obj.fun())

Output[¶](https://devu.in/tutorial/#Output)

I am a function from class B

Method Overloading[¶](https://devu.in/tutorial/#Method-Overloading)

It is the ability to call a class method with different numbers of arguments. For equal or less arguments than defined we can use default values for arguments. But if we wish to pass more argument values than defined, wheather normal or keyworded argument, we can use \*args and \*\*kwargs respectively.

Methods can easily be overloaded by assigning default value to parameters. The argument value is always prioritized over the default. Methods can also be overloaded with the use of \*args and \*\*kwargs as parameters, they are used as option parameters.

* \*args means tuple of arguments
* \*\*kwargs means keyworded list of arguments i.e, dictionary
* \* of \*args should only be in the definition but not in the body of the function.
* \*args and \*\*kwargs are merely convention, another identifier can also be used.
* If non-default variable has to used with *args then it must be before the*args in function definition.

class A:

def fun(self,a=0,b=0):

return a+b

obj = A()

print(obj.fun(1))

print(obj.fun(1,2))

print(obj.fun())

def fun(a,\*args,\*\*kwargs):

print("Non-default value:",a)

print("Tuple of optional arguments: ",args)

print("keyworded argument")

for i in kwargs.keys():

print(i)

for i in kwargs.values():

print(i)

for i in kwargs.items():

print(i)

fun("John",35,3245,34,5,6,user="root",host="localhost")

Operator overlaoding[¶](https://devu.in/tutorial/#Operator-overlaoding)

It means changing the default behaviour of an operator. When an addition defition for the operator is also added, it can be called overloaded.

Example: Tuples when added with another tuple gets concatenated

t1 = (1,2)

t2 = (3,4)

print(t1+t2)

(1,2,3,4)

But the desired output is of vector addition, i.e, (4,6)

class A:

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

def \_\_add\_\_(obj1,obj2):

return "(%i,%i)" %(obj1.a+obj2.a,obj1.b+obj2.b)

obj1 = A(1,2)

obj2 = A(3,4)

print(obj1+obj2)

Encapsulation[¶](https://devu.in/tutorial/#Encapsulation)

It means hiding attributes of a class or making them private.

* Hiding: \_\_attribute
* Unhiding: \_class\_\_attribute

class A:

\_\_x = 100

def \_\_f1(self):

return "Hello world"

obj = A()

print(obj.\_A\_\_x)

print(obj.\_A\_\_f1())

**Multi-threading**[**¶**](https://devu.in/tutorial/#Multi-threading)

In computing, a process is an instance of a computer program that is being executed. Just like multiprocessing, multithreading is a way of achieving multitasking with optimal uses of available resources. In multithreading, the concept of threads is used which can be understood as a lightweight process.

Any process has 3 basic components:

* An executable code
* The associated data
* The execution context of the program (State of process)
* A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS.

In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

Making a thread[¶](https://devu.in/tutorial/#Making-a-thread)

**Import the libraries**[**¶**](https://devu.in/tutorial/#Import-the-libraries)

In Python3 threading library defines Thread class with functions to start and run a thread. So begin with the import of Thread class from threading module

>>> from threading import Thread

**Make thread objects**[**¶**](https://devu.in/tutorial/#Make-thread-objects)

Define a class which subclasses the Thread class. The objects of this new class thus will also be a thread object:

>>> class MyThread(Thread): pass

>>> threadObject = MyThread()

**Starting thread object**[**¶**](https://devu.in/tutorial/#Starting-thread-object)

Use the inherited function start() for starting a thread object.

>>> threadObject.start()

**Run the target function/method as thread**[**¶**](https://devu.in/tutorial/#Run-the-target-function/method-as-thread)

>>> ...

def run(self):

call\_to\_function()

from threading import Thread

import time

def fun(t,d):

count = 0

while count<2:

print("\nCurrent thread is: %s" %t)

time.sleep(d)

count += 1

class MyThread(Thread):

def \_\_init\_\_(self,tname,delay):

self.tname = tname

self.delay = delay

Thread.\_\_init\_\_(self)

self.start()

def run(self):

print("%s started\n" %self.tname)

fun(self.tname,self.delay)

t1 = MyThread("Thread-1",2)

t2 = MyThread("Thread-2",5)

Output[¶](https://devu.in/tutorial/#Output)

Thread-1 started

Thread-2 started

Current thread is: Thread-1

Current thread is: Thread-2

Current thread is: Thread-1

Current thread is: Thread-2

**Socket programming**[**¶**](https://devu.in/tutorial/#Socket-programming)

Socket programming is about message/information exchange between two parties, (referred as client and server) via network sockets.

What are sockets?[¶](https://devu.in/tutorial/#What-are-sockets?)

Sockets are end-points in a communication link defined by host and port combination.

For client-server communication, sockets are configured at the two ends, namely client and server. Server initites the connection and listens for client. Upon client connection request and acceptance, the responses are sent at both ends thereby establishing a bidirectional communication.

How to create a socket object in python?[¶](https://devu.in/tutorial/#How-to-create-a-socket-object-in-python?)

To create/initialize a socket, we use the socket.socket() method defined in the Python’s socket module. It's syntax is as follows.

sock\_obj = socket.socket( socket\_family, socket\_type, protocol=0)

here,

* socket\_family: defines the family of protocols used as the transport mechanism. It can have either of the two values. Either AF\_UNIX, or AF\_INET (IP version 4 or IPv4).
* socket\_type: defines the types of communication between the two end-points. It can have following values. SOCK\_STREAM (for connection oriented protocols e.g. TCP), or SOCK\_DGRAM (for connectionless protocols e.g. UDP).
* protocol: We typically leave this field or set this field to zero.

import socket

#Instantiate an AF\_INET, STREAM socket (TCP)

sock\_obj = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

print ('Socket Initialized')

Server socket methods.[¶](https://devu.in/tutorial/#Server-socket-methods.)

* sock\_object.bind(address): This method binds the socket to address (hostname, port number pair)
* sock\_object.listen(backlog): This method is used to listen to the connections associated with the socket. The backlog parameter indicates the maximum number of queued connections.Maximum value can go up to 5 and minimum should be at least zero.
* sock\_object.accept(): This function returns (conn, address) pair where ‘conn’ is new socket object used to send and receive data on the communication channel and ‘address’ is the IP address tied to the socket on the another end of the channel.Execution of accept() method returns a socket object that is different from the socket object created using socket.socket(). This new socket object is dedicatedly used to manage the communication with the particular client with which accept happened. This mechanism also helps Server to maintain the connection with n number of clients simultaneously.

Client socket methods.[¶](https://devu.in/tutorial/#Client-socket-methods.)

* sock\_object.connect(): This method is used to connect the client to host and port and initiate the connection towards the server.

Other socket methods.[¶](https://devu.in/tutorial/#Other-socket-methods.)

* sock\_object.recv(): Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.
* sock\_object.send(): Apply this method to send messages from endpoints in case the protocol is TCP.
* sock\_object.recvfrom(): Call this method to receive messages at endpoints if the protocol used is UDP.
* sock\_object.sendto(): Invoke this method to send messages from endpoints if the protocol parameter is UDP.
* sock\_object.gethostname(): This method returns the hostname.
* sock\_object.close(): This method is used to close the socket. The remote endpoint will not receive data from this side.

Python-Server.Py[¶](https://devu.in/tutorial/#Python-Server.Py)

This server module will both send and receive data to/from the client. In Python-Server.py, we have created a server socket which remains in the wait state till it receives a request from the client. Whenever a client gets connected, the server accepts that connection. The client will then start passing messages to the server. And the server will process those messages and send the response back to the client. In the below code, we are also asking the user to input his response that he wants to pass to the client.

**Email sending**[**¶**](https://devu.in/tutorial/#Email-sending)

Python supports email sending using the *sendmail()* function. To invoke *sendmail()* function one must call by SMTPobject. *sendmail()* requires three parameters, firstly, the sender email address as string, secondly, receiver email address as string or list and finally a string message. If sender has a gmail account he must use smtp.gmail.com as the SMTP server. Additionally, gmial requires a secure tramission port TLS:587 unlike unsecured port SMTP:25. Simply login by giving your gmail username and password. To logout use *quit()*.

* Step 1: Ready your variables like sender receiver, message, username, passowrd etc.
* Step 2: Login with correct username and password and also make sure the security settings allow it.
* Step 3: Invoke sendmail() function with SMTP object
* Step 4: Send it all securely uning TLS port 587
* Step 5: Logout
* Add exception handling if necessary.

In [ ]:

# Email sending using Gmail

import smtplib

sender = 'sender\_email@gmail.com'

receiver = 'receiver\_email@gmail.com'

message = """ This is a test email. Ignore. """

username = sender

password = "your\_email\_password"

try:

SMTPobj = smtplib.SMTP("smtp.gmail.com:587")

SMTPobj.starttls()

SMTPobj.login(username,password)

SMTPobj.sendmail(sender,receiver,message)

SMTPobj.quit()

print("Email sent successfully.")

except:

print("Email sending failed.")

# Email sending with attachment

from smtplib import SMTP

from getpass import getpass

from email.mime.multipart import MIMEMultipart

from email.mime.image import MIMEImage

from email.mime.text import MIMEText

username = '\_\_\_\_\_\_\_\_@gmail.com'

password = getpass()

sender = username

receiver = ['\_\_\_\_\_\_@\_\_\_\_.\_\_\_\_','\_\_\_\_\_\_@\_\_\_\_.\_\_\_\_']

message = MIMEMultipart()

message['Subject'] = "Say your email subject here."

# message['Cc'] = '\_\_\_\_\_\_@\_\_\_\_.\_\_\_\_'

# message['Bcc'] = '\_\_\_\_\_\_@\_\_\_\_.\_\_\_\_'

# message['From'] = 'Your name '

txt = """

plain text

Red color bold html text

"""

t = MIMEText(txt,'html')

message.attach(t)

img = open('\_\_\_\_\_\_\_.\_\_\_\_\_','rb') # Add image path if required

i = MIMEImage(img.read())

message.attach(i)

try:

try:

SMTPobj = SMTP('smtp.gmail.com:587')

except:

print("SMTP server error")

SMTPobj.starttls()

try:

SMTPobj.login(username,password)

except:

print("Username/Password incorrect")

SMTPobj.sendmail(sender, receiver, message.as\_string())

SMTPobj.quit()

print("Email sending successful.")

except:

print("Email sending failed. Check internet connection")