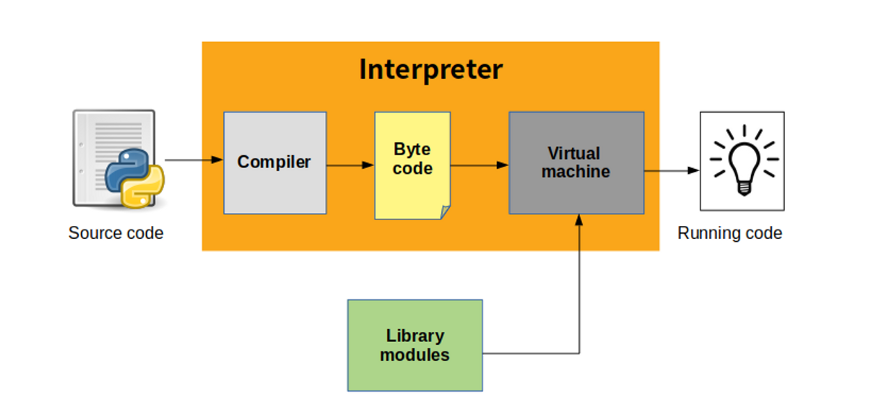
# Python features

Simple | Dynamic | Robust | High Level | Interpreted | Scripting | General purpose | Object oriented | Popular | Powerful | platform independent | garbage-collected

# Founder

It was created by Guido van Rossum, and released in 1991

# Internal working of Python



Code with error, halts execution (exit)

PVM

Syntax checker &

Translator

* **Step 1:** The python compiler reads a python source code/ instruction. Then it verifies that the instruction is well-formatted, i.e., it checks the syntax of each line. If it encounters an error, it immediately halts the translation and shows an error message.
* **Step 2:**If there is no error, i.e., if the python instruction or source code is well-formatted then the compiler translates it into its equivalent form in an intermediate language called “Byte code”/ “Intermediate code”.
* **Step 3:**Byte code is then sent to the Python Virtual Machine (PVM) which is the python interpreter. PVM converts the python byte code into machine-executable code. If an error occurs during this interpretation, then the conversion is halted with an error message.

# Difference between python 2.x and 3.x

* ***Print***- Keyword (2.x) print … & Function (3.x) print(…)
* ***Input***- input (3.x), raw\_input (2.x)

x = input() # saranj

print(x, type(x)) # saranj <class ‘str’>

y = input() # 1

print(y, type(y)) # 1 <class ‘str’> #by default input type is str no matter what is given to it

input(2.x)

x = input() # saranj

print(x, type(x)) # saranj <class ‘str’>

y = input() # 1

print(y, type(y)) # 1 <class ‘int’>

* ***Integer division-***Python 2 treats numbers that you type without any digits after the decimal point as integers, which can lead to some unexpected results during division.

print 7 / 5

print -7 / 5

'''

Output in Python 2.x

1

-2

Output in Python 3.x :

1.4

-1.4

'''

* ***Unicode****-* In Python 2, an implicit str type is ASCII.

But in Python 3.x implicit str type is Unicode.

* ***Xrange****-* Does not supported in 3.x

xrange(1, 5) # 1, 2, 3, 4, 1, 2, 3, 4

* ***Error handling***- There is a small change in error handling in both versions. In python 3.x, ‘as’ keyword is required.

# Comments

* Single line Comment - #
* String - ‘…’, “ … ” and Multiline String - ‘’’ ….’’’, “”” ….””” without assigning to a variable can be used as single line comment and multiline Comments.
* Comments can be used to explain Python code, make the code more readable and it can be used to prevent execution when testing code.

# Variables

Variables are containers for storing data values.

*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 alpha-numeric characters and underscores (A-z, 0-9, and \_ )
* Variable names are case-sensitive (age, Age and AGE are three different variables)

**Legal variable names:**

myvar = "John" / my\_var = "John" / \_my\_var = "John" / myVar = "John" / MYVAR = "John" / myvar2 = "John"

**Illegal variable names:**

2myvar = "John" / my-var = "John" / my var = "John"

**Camel Case -** myVariableName = "John"

**Pascal Case -** MyVariableName = "John"

**Snake Case -** my\_variable\_name = "John"

Scope

*Global variables* can be used by everyone, both inside of functions and outside.

*Local variables* can only be used inside the function.

i = 200 #created a global variable

print(i) # 200

for i in range(3): #creates a global scope, altered the global variable as the name are same

    print(i) # 0, 1, 2 (for each loop)

print(i) # 2 #now the value of global variable is 2

No, there is no language support for creating block scope. (If, If Else), {}

The following constructs create scope:

Module | class | function (incl. lambda) | generator expression | comprehensions (dict, set, list(in Python 3.x))

x = 'hero'

def myfunc():

  global x // global x = “fantastic” Invalid syntax

  x = "fantastic"

print(x) # hero //initial global value

myfunc()

print(x) # fantastic //newly altered global value from function

x = "awesome"

print(x) # awesome //again newly altered global value

# Package, module and library

**Module**: The module is a simple Python file that contains collections of functions and global variables and with having a .py extension file. It is an executable file and to organize all the modules we have the concept called Package in Python.

**Package**: The package is a simple directory having collections of modules. This directory contains Python modules and also having \_\_init\_\_.py file by which the interpreter interprets it as a Package. The package is simply a namespace. The package also contains sub-packages inside it. *(More reference: https://www.geeksforgeeks.org/python-packages/)*

**Library**: The library is having a collection of related functionalities of codes that allows you to perform many tasks without writing your code. It is a reusable chunk of code that we can use by importing it in our program, we can just use it by importing that library and calling the method of that library with period(.).

# Casting

* Conversion one data type to another
* to specify the data type of a variable.
* Performed using constructor functions - int(), str(), float(), list() etc

a = list(10) # error: ‘int’ object in not iterable

b = print(list({'a': 1, 'b': 2})) #[‘a’, ‘b’]

c = list('mango' ) # [‘m’, ‘a’, ‘n’, ‘g’, ‘o’]

# Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

example of an illegal character is a double quote inside a string that is surrounded by double quotes.

\’ (Single Quote), \\ ( Backslash), \n (New Line), \t (Tab), \’’ (Double Quote)

# False Values

# (), [], {}, "", the number 0, and the value None. And the value False evaluates to False.

# Build-in function

* The id() function (unique identifier)returns a unique id for the specified object.
* All objects in Python have its own unique id.
* The id is assigned to the object when it is created.
* The id is the object's memory address and will be different for each time you run the program. (except for some object that has a constant unique id, like integers from -5 to 256)

x = 'saranj'

print(type(x)) #<class 'str'> • Get the data type of any object

print(id(x)) #25823232

print(isinstance(x, str)) #True #determine if an object is of a certain data type

print(chr(65)) # A

print(ord('a')) # 97

print(bin(5)) # 0b101 decimal to binary

print(oct(10)) # 0o12 decimal to octal

print(hex(10)) # 0xa decimal to hexadecimal

'''print(int(101, 2)) #TypeError

can't convert non-string with explicit base'''

print(int('101', 2)) # 5 binary to decimal

print(int('12', 8)) # 10 octal to decimal

print(int('a', 16)) # 10 hexadecimal to decimal

# Datatypes

|  |  |  |  |
| --- | --- | --- | --- |
| Text Type: |  |  | str |
| Numeric Types: |  |  | int, float, complex |
| Sequence Types: |  |  | list, tuple, range |
| Mapping Type: |  |  | dict |
| Set Types: |  |  | set, frozenset |
| Boolean Type: |  |  | bool |
| Binary Types: |  |  | bytes, bytearray, memoryview |

x = 'hero'

print(id(x)) # 18066240

print(memoryview(bytes(id(x)))) # <memory at 0x010B4E00>

# Numeric

* Scientific numbers with an "e" to indicate the power of 10 are belong to Float (\_.\_) class.
* Complex numbers are written with a "j" as the imaginary part.
* Cannot convert complex numbers into another number type.

y, x = 0j, 1 + 0j

print(type(y), type(x)) # <class ‘complex’> <class ‘complex’>

# Strings

* Strings in Python are arrays of bytes representing Unicode characters.
* Python does not have a character data type, a single character is simply a string with a length of 1.
* we can combine strings and numbers by using the format() method!
* While using multistring, the line breaks are inserted at the same position as in the code.
* The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are, and creates a string.

# Operators

* **Arithmetic operators** : + (Addition), - (Subtraction) , \* (Multiplication), / (Division), % (Modulus) , \*\* (Exponentiation) , // (Floor division)
* **Assignment operators** : = , +=, -=, \*=, /= , %=, \*\*= , //=, &= , |= , ^=, >>=, <<=
* **Comparison operators** : ==, !=, > , < , >= , <=
* **Logical operators** : and, or, not
* **Identity operators** : is, is not
* **Membership operators :** in, not in
* **Bitwise operators :** & (AND), | (OR), ^ (XOR), ~ (NOT), << (Zero fill left shift), >> (Signed right shift)

*Right shift is calculated by /2 & Left shift is calculated by \*2*

x = 100

print(x>>3) # 100 / (2\*2\*2) = 12

print(x>>4) # 100 / (2\*2\*2\*2) = 6

print(x<<3) # 100 \* (2\*2\*2) = 800

print(x<<4) # 100 \* (2\*2\*2\*2) = 1600

500/100 => 5.0 (float)

500//100 => 5 (int) (floor division)

‘s’\*3 => ‘sss’

‘s’\*’3’ => TypeError: can't multiply sequence by non-int of type 'str'

‘s’ + 3 => TypeError: can't concatenate by int with type 'str'

‘s’ + ‘3’ => s3

# Python Collections (Arrays)

* **List** is a collection which is **ordered** and **changeable**. Allows **duplicate** members.
* **Tuple** is a collection which is **ordered** and **unchangeable**. Allows **duplicate** members.(**immutable**)
* **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.**
* *\*As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.*
* Set items are unchangeable, but you can remove items and add new items.

# Slicing

Included Excluded

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| ‘Mango’ | ‘Banana’ | ‘kiwi’ | ‘cherry’ |
| -4 | -3 | -2 | -1 |

l = ['Mango','banana', 'kiwi', 'cherry']

print(l[0: 2]) # ['Mango', 'banana']

print(l[3:2]) # []

print(l[-2: -3]) # []

print(l[-3: -2]) # ['banana']

print(l[3:]) # ['cherry']

print(l[2:2]) # [] \*

print(l[:0]) # []

print(l[:1]) # ['mango']

print(l[::-2]) # ['cherry', 'banana']

print(l[::-1]) # ['cherry', 'kiwi', 'banana', 'mango']

print(l[::2]) # ['mango', 'kiwi']

print(l[:-1]) # [‘Mango’, ‘banana’, ‘kiwi’]

print(l[:-2]) # [‘Mango’, ‘banana’]

print(l[-2:]) # [‘kiwi’, ‘cherry’]

# Operation between mutable and immutable

Accessing or unpacking is possible in immutable as well as in mutable.

But modifying is not possible with immutable.

Immutable datatypes used same storage location to store the similar data for memory utilization and to reduce memory fragmentation.

* Some of the **mutable** data types in Python are **list, dictionary, set** and **user-defined classes**.
* Some of the **immutable** data types are **int, float, decimal, bool, string, tuple, and range**.

a = (1, 2, 3)

b = (1, 2, 3)

print(a is b) # True

print(a == b) # True

print(id(a), id(b)) # 23403616 23403616

a = 'saranj'

b = 'saranj'

print(a is b) # True

print(a == b) # True

print(id(a), id(b)) # 30714688 30714688

Integers are also immutable, above operation is same for integers(i.e., immutable data type).

a = [1, 2, 3]

b = [1, 2, 3]

print(a is b) # False

print(a == b) # True

print(id(a), id(b)) # 30426616 30427776

# Range

range(start, stop, step)

start: Optional. An integer number specifying at which position to start. Default is 0 (Included)

stop: An integer number specifying at which position to end. (Excluded)

step: Optional. An integer number specifying the incrementation. Default is 1

# List

The extend() method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

List objects have a sort() method that will sort the list alphanumerically, ascending, by default. By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a *reference* to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy(). (A new duplicate list is created at different location.) Another way to make a copy is to use the built-in method list()

|  |  |
| --- | --- |
| Method | Description |
| [append()](https://www.w3schools.com/python/ref_list_append.asp) | Adds an element at the end of the list |
| [clear()](https://www.w3schools.com/python/ref_list_clear.asp) | Removes all the elements from the list |
| [copy()](https://www.w3schools.com/python/ref_list_copy.asp) | Returns a copy of the list |
| [count()](https://www.w3schools.com/python/ref_list_count.asp) | Returns the number of elements with the specified value |
| [extend()](https://www.w3schools.com/python/ref_list_extend.asp) | Add the elements of a list (or any iterable), to the end of the current list |
| [index()](https://www.w3schools.com/python/ref_list_index.asp) | Returns the index of the first element with the specified value |
| [insert()](https://www.w3schools.com/python/ref_list_insert.asp) | Adds an element at the specified position |
| [pop()](https://www.w3schools.com/python/ref_list_pop.asp) | Removes the element at the specified position |
| [remove()](https://www.w3schools.com/python/ref_list_remove.asp) | Removes the item with the specified value |
| [reverse()](https://www.w3schools.com/python/ref_list_reverse.asp) | Reverses the order of the list |
| [sort()](https://www.w3schools.com/python/ref_list_sort.asp) | Sorts the list |

Insert(>length of given list, val) = always insert at last position

#list comprehension

x = [1, 2, 3, 4, 5, 6, 7]

a = [i for i in x if i%2 == 0]

print(a) # [2, 4, 6]

y = ['mango', 'banana', 'kiwi', 'orange']

b = [i.upper() for i in y] #[‘MANGO’, ‘BANANA’, ‘KIWI’,’ ORANGE’]

c = ['B' if i == 'banana' else 'Fruit' for i in y] #[‘Fruit’, ‘B’, ‘Fruit’, ‘Fruit’]

d = ['B' if i == 'banana' 'k' elif i == 'kiwi' else 'Fruit' for i in y] # error elif not supported in list comprehension

# Tuple

**Updating a Tuple**

* **Convert into a list**: Just like the workaround for*changing*a tuple, you can convert it into a list, add/remove your item(s), and convert it back into a tuple.
* **Add tuple to a tuple**: You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple

x = (1, 2, 3)

y = (4, 5, 6)

x += y

print(x) #(1, 2, 3, 4, 5, 6)

**Note:** When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple, ie it will consider as respective datatype.

(‘s’,) => tuple | (‘s’, 10 )=> tuple | (‘s’) => str | (1) => int

Use tuple when returning collection from function as there would be no change during the transition.

**Unpacking a tuple**

## Using Asterisk\*

If the number of variables is less than the number of values, you can add an \* to the variable name and the values will be assigned to the variable as a list.

# Set

* Once a set is created, you cannot change its items, but you can add new items.
* If the item to remove does not exist, discard() will NOT raise an error whereas remove() will raise an error.
* Sets are unordered, so when using the pop() method, you do not know which item that gets removed.
* Both union() and update() will exclude any duplicate items.

x = {1, 2, 3, 4, 5}

y = {5, 6, 7, 8, 9}

print(x | y) #union {1, 2, 3, 4, 5, 6, 7, 8, 9}

print(x & y) #intersection {5}

print(x - y) #difference {1, 2, 3, 4}

# Dictionary

|  |  |
| --- | --- |
| Method | Description |
| [clear()](https://www.w3schools.com/python/ref_dictionary_clear.asp) | Removes all the elements from the dictionary |
| [copy()](https://www.w3schools.com/python/ref_dictionary_copy.asp) | Returns a copy of the dictionary |
| [fromkeys()](https://www.w3schools.com/python/ref_dictionary_fromkeys.asp) | Returns a dictionary with the specified keys and value |
| [get()](https://www.w3schools.com/python/ref_dictionary_get.asp) | Returns the value of the specified key |
| [items()](https://www.w3schools.com/python/ref_dictionary_items.asp) | Returns a list containing a tuple for each key value pair |
| [keys()](https://www.w3schools.com/python/ref_dictionary_keys.asp) | Returns a list containing the dictionary's keys |
| [pop()](https://www.w3schools.com/python/ref_dictionary_pop.asp) | Removes the element with the specified key |
| [popitem()](https://www.w3schools.com/python/ref_dictionary_popitem.asp) | Removes the last inserted key-value pair |
| [setdefault()](https://www.w3schools.com/python/ref_dictionary_setdefault.asp) | Returns the value of the specified key. If the key does not exist: insert the key,  with the specified value |
| [update()](https://www.w3schools.com/python/ref_dictionary_update.asp) | Updates the dictionary with the specified key-value pairs |
| [values()](https://www.w3schools.com/python/ref_dictionary_values.asp) | Returns a list of all the values in the dictionary |

d = {1:'a', 2:'b'}

d0 = d # d0 pointing to the same memory address of d i.e, id(d) is id(d0)

d.popitem()

print(d0, d) #{1: 'a'} {1: 'a'}

# Ternary operator/ Conditional expressions

a = 2  
b = 330  
print("A") if a > b else print("B") # B

a = 330  
b = 330  
print("A") if a > b else print("=") if a == b else print("B") # =

# For loop

Python’s **for** loop are as a **foreach**

So, it does not matter if you modify **item** inside the loop as it will be given its new value at the start of the next iteration

>>> **for** i in range(5):

... **if** i == 2:

... i = 4

... **print**(i)

...

0

1

4

3

4

**Solution:** Use while loop.

for x in range(6):

print(x, end=” ”)

else:

print("Finally finished!")

#0 1 2 3 4 5

# Finally finished!

The else block will NOT be executed if the loop is stopped by a break statement.

# Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments but can only have one expression.

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

*def myfunc(n):  
  return lambda a : a \* n  
  
mydoubler = myfunc(2)  
mytripler = myfunc(3)  
  
print(mydoubler(11)) # 22  
print(mytripler(11)) # 33*

λ(x) => function(black box) => λ(x). x + 1

x => x+1

5 => 5+1 = 6

λ(x).λ(y) => λ(x). λ(y). x + y

x, y => x+y

l = (lambda x, y: x+y)(5, 10)

print(l) #15

l1 = (lambda x: x+1)(5)

print(l1) #6

# Use lambda functions when an anonymous function is required for a short period of time.

### Function

## 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])#Linus  
  
my\_function("Emil", "Tobias", "Linus")

## Arbitrary Keyword Arguments, \*\*kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: \*\* before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly:

def my\_function(\*\*kid):  
  print("His last name is " + kid["lname"])# Refsnes  
  
my\_function(fname = "Tobias", lname = "Refsnes")

# JSON

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

json.loads()

PYTHON OBJECT

JSON STRING

json.dumps()

# RegEx Functions

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

The re module offers a set of functions that allows us to search a string for a match:

|  |  |
| --- | --- |
| Function | Description |
| [findall](https://www.w3schools.com/python/python_regex.asp#findall) | Returns a list containing all matches |
| [search](https://www.w3schools.com/python/python_regex.asp#search) | Returns a [Match object](https://www.w3schools.com/python/python_regex.asp#matchobject) if there is a match anywhere in the string |
| [split](https://www.w3schools.com/python/python_regex.asp#split) | Returns a list where the string has been split at each match |
| [sub](https://www.w3schools.com/python/python_regex.asp#sub) | Replaces one or many matches with a string |

PIP

PIP is a package manager for Python packages, or modules

# Exception Handling

*try*:

# The try block lets you test a block of code for errors.

*except*:

# optional block

# Handling of exception (if required)

*else*:

# execute if no exception

*finally*:

# The finally block lets you execute code, regardless of the result of the try- and except blocks.

## Raising Exception

The [raise statement](https://www.geeksforgeeks.org/python-raising-an-exception-to-another-exception/) allows the programmer to force a specific exception to occur. The exception should be an instance or an exception class (a class that derives from Exception).

|  |
| --- |
| *try*:  *raise* NameError("Hi there")  # Raise Error  *except* NameError:  *print* ("An exception")  *raise*  # To determine whether the exception was raised or not |

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

Traceback (most recent call last):

File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>

raise NameError("Hi there") # Raise Error

NameError: Hi there

# None

This is a special constant used to denote a null value or a void. It’s important to remember, 0, any empty container(e.g empty list) does not compute to None.

It is an object of its datatype – NoneType. It is not possible to create multiple None objects and can assign them to variables.

# Return Keywords – Return, Yield

* [**return**:](https://www.geeksforgeeks.org/python-return-statement/) This keyword is used to return from the function.
* **Return** sends a specified value back to its caller whereas **Yield** can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don’t want to store the entire sequence in memory.

*def* fun():

    s = 0

    for i in range(10):

        s += 1

    return s

print(fun()) #10

* [**yield**:](https://www.geeksforgeeks.org/python-yield-keyword/) This keyword is used like return statement but is used to return a generator.
* The yield statement suspends function’s execution and sends a value back to the caller but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

*def* fun():

    s = 0

    for i in range(10):

        s += 1

    yield s

print(fun()) #<generator object fun at 0x000001C212529A10>

for i in fun():

    print(i) #1, 2, 3, 4, 5 … 10

### Decorator

[**Decorators**](https://www.geeksforgeeks.org/function-decorators-in-python-set-1-introduction/) are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

### First Class Objects

### In Python, functions are [**first class objects**](https://www.geeksforgeeks.org/first-class-functions-python/) that mean that functions in Python can be used or passed as arguments.

### **Properties of first class functions:**

* A function is an instance of the Object type.
* You can store the function in a variable.
* You can pass the function as a parameter to another function.
* You can return the function from a function.
* You can store them in data structures such as hash tables, lists, …

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

**Syntax for Decorator:**

@gfg\_decorator

def hello\_decorator():

print("Gfg")

'''Above code is equivalent to -

def hello\_decorator():

print("Gfg")

hello\_decorator = gfg\_decorator(hello\_decorator)'''

## File Handling

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 - Opens a file for appending, creates the 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

Note- write (w) delete the whole content of the file and then start writing.

In addition, you can specify if the file should be handled as binary or text mode (Optional)

"t" - Text - Default value. Text mode

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

Shebang Line | Pound bang | hash bang

The #! syntax used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems. Most Linux shell and [perl](https://bash.cyberciti.biz/guide/Perl" \o "Perl) / [python](https://bash.cyberciti.biz/guide/Python) script starts with the following line:

*#!/bin/bash*

*#!/usr/bin/python3*

It is nothing but the absolute path to the [Bash](https://bash.cyberciti.biz/guide/Bash) [interpreter](https://bash.cyberciti.biz/wiki/index.php?title=Interpreter&action=edit&redlink=1).

It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /bin/bash.

All scripts under Linux execute using the interpreter specified on a first line.

Almost all bash scripts often begin with #!/bin/bash (assuming that Bash has been installed in /bin)

This ensures that Bash will be used to interpret the script, even if it is executed under another shell.

### Map, Filter, Reduce

import functools as ft

x = [1,2,3,4,5,6,7,8,9]

a = list(map(lambda i : i\*10, x))

print(a) #[10, 20, 30, 40, 50, 60, 70, 80, 90]

b = list(filter(lambda i: i%2 == 0, x))

print(b) #[2, 4, 6, 8]

c = ft.reduce(lambda i, j: i+j, x)

print(c) #45

# Logging

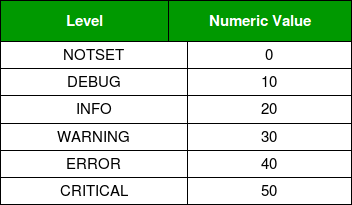
Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging, and running. If you don’t have any logging record and your program crashes, there are very few chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging, you can leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.

**Why Printing is not a good option?**

Some developers use the concept of printing the statements to validate if the statements are executed correctly or some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but for complex scripts, the printing approach will fail.  
Python has a built-in module logging which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

**Levels of Log Message**

* Debug : These are used to give Detailed information, typically of interest only when diagnosing problems.
* Info : These are used to confirm that things are working as expected
* Warning : These are used an indication that something unexpected happened, or is indicative of some problem in the near future
* Error : This tells that due to a more serious problem, the software has not been able to perform some function
* Critical : This tells serious error, indicating that the program itself may be unable to continue running



Debug and info won’t come at console