1. Python Revision Tour

Key Points

➢ **Introduction :**

- Python was developed by Guido Van Rossum in 1991 when he was working with National Research Institute of Mathematics and Science in Netherland.
- Python was named from a comedy series "Monty Python's Flying Circus" telecasted on BBC.

➢ **Character Set :**

- **Character Set**-is a group of letters or signs which are specific to a language.
- Character set includes letters, sign, numbers, symbols.

  Letters: A-Z, a-z

  Digits: 0-9

  Special Symbols: _, +, -, *, /, (,), {, } . . . Etc.

  White Spaces: blank space, tab, carriage return, newline, form feed etc.

  Other characters: Python can process all characters of ASCII and UNICODE.

➢ **Token :** Token- is the **smallest unit** of any programming language. **It is also known as Lexical Unit.**

Types of Token :

- Keywords
- Identifiers
- Literals
- Punctuators
- Operators

1. **Keywords :** Keywords are those words which provides a special meaning to interpreter. These are reserved for specific functioning. These can not be used as identifiers, variable name or any other purpose.

2. **Identifiers :** These are building blocks of a program and are used to give names to different parts/blocks of a program like - variable, objects, classes, functions.

Rules to Declare an Identifier:

a) An identifier may be a **combination of letters and numbers.**

b) An identifier must **begin with an alphabet or an underscore (_).**

c) Subsequent letters may be numbers(0-9).

d) Python is **case sensitive.** Uppercase characters are distinct from lowercase characters (P and p are different for interpreter).

e) Length of an Identifier is unlimited.

f) Keywords can not be used as an identifier.

g) Space and special symbols are not permitted in an identifier name **except an underscore(_) sign.**

3. **Literals :** Literals are often called Constant Values.

Python permits following types of literals -

String literals - "Pankaj"

Numeric literals – 10, 13.5, 3+5i

Boolean literals – True or False

Special Literal - *None*

4. **Punctuators :** In Python, punctuators are used to construct the program and to make balance between instructions and statements.

Python has following Punctuators -

'	"	#	\	( )
[	]	{	}	@.
:	..	`	=	

5. **Operators :** The symbols that trigger the action / operation on data, are called Operators. The data on which operation is being carried out are called Operands.

Arithmetic Operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Floor Division
**	Exponentiation
//	Floor Division

Relational Operators

Operator	Name
==	Double Equals To
!=	Not Equals to
>	Greater Than
>=	Greater than or equals to
<	Less than
<=	Less than or equals to

Identity Operators

Operator	Name
is	Returns True, if both variables are the same object.
is not	Returns True, if both variables are not the same object.

Membership Operators

in	Returns True, if a sequence with the specified value is present in the object.
not in	Returns True, if a sequence with a specified value is not present in the object.

Logical Operators

Operator	Name
and	Returns True, if both the statements are True.
or	Returns True , if one of the statement is True.
not	Reverse the result.

Precedence of Operators: (Highest to lowest):

Operator	Description
()	Parentheses *Highest*
**	Exponentiation
+x , -x	Positive , Negative
* , / , // , %	Multiplication , Division , Floor Division , Remainder
+ , -	Addition , Subtraction
<, <= , >, >= , != , == is , is not	Comparison and Identity Operators
not x	Logical NOT
and	LOGICAL AND
or	Logical OR *Lowest*

- **Variables in Python :** Variable is a name given to a memory location. In Python, there is no need to specify the datatype of variable. Python automatically get variable datatype depending upon the value assigned to the variable.
- **Dynamic Typing :** Data type of a variable depend/change upon the value assigned to a variable on each next statement.
  X = 65 # integer type
  X = "Python" # x variable data type change to string on just next line
  Now programmer should be aware that not to write like this: Y = X / 5 # error !! String cannot be divided.
- **input ( ) :** input( ) function is used to take input from a user. Input function always returns a value of string type.
  Syntax : VarToHoldValue = input (<prompt to be displayed>)
- **print ( ) :** print( ) function is used to display a string on the console.
  Syntax: print(*objects , [sep=' ' or <separator string> end='\n' or <end string>)
  Features of print ( ) :
  - It auto-converts the items into strings.
  - It inserts spaces between items automatically because default value of sep argument is space.
  - It appends a new line character at the end of a line unless you give your own end argument.
- **Data Types :** Data type in Python specifies the type of data we are going to store in any variable and the type of operation we can perform on a variable. Data can be of many types e.g. character, integer, real, string etc.

Immutable Data Types	Mutable Data Types
The immutable types are those that can never change their value in place.	Change their value in place.
E.g. ▪Integers ▪Floating Point Numbers ▪Booleans ▪Strings ▪Tuples	E.g. ▪ List ▪ Dictionary

➢ **Conditional and Control Statements :**

- **Statements :** Statements are the instructions given to the computer to perform any kind of action.

  Types of Statements :

  a) *Empty Statement* - Empty statement is a **statement that does nothing.** Represented by a keyword ***pass***. Useful in those cases, where the syntax of a language requires the presence of a statement but where the logic of the program does not.

  b) Simple Statement - Any single executable statement is a statement in Python.

  c) *Compound Statement* - A compound statement represents a group of statements executed as a unit.

- **Statement Flow Control :** Flow of control means to determine the next statement to be executed. It can be : Sequential , Selection , Iteration.

  1. **Sequence**: Each statement will be executed in a Sequential Manner.
  2. **Selection** : The statement which is to be executed is based on the condition.

a) **if statement** : The if statements are **conditional statements** in Python and these implement Selection Constructs or Decision Constructs. The simplest form of if statement **tests a condition** and if the condition evaluates to **True**, it carries out some instructions and does nothing in case condition evaluates to **False**.

Syntax:

if < conditional expression> :
    statements to be executed

Interesting Facts:

```
if 0:
    print(                    )
if False:
    print(                    )
```
Condition will be False. No Output will get printed.

```
if 8:
    print(                    )
if -5:
    print(                    )
if True:
    print(                    )
```
Condition will be TRUE for any NON-ZERO Value.

b) **if – else statement :** This form of if statement tests a condition and if the condition evaluates to True , it carries out the statements indented below and in case, condition evaluates to False, it carries out statements indented below else.

Syntax:

if < conditional expression> :
    statements to be executed
    .....
else:
    statements to be executed
    .....

c) **if – elif :** If out of multiple statements, it is required to select one statement for processing on the basis of a condition, if-elif statement is to be used.

Syntax :

if < conditional expression> :
    statements to be executed
elif <conditional expression>:
    statements to be executed
elif <conditional expression>:
    statements to be executed
else:
    statements to be executed

d) **Nested – if :** A nested if is an if that has another if in it's if's body or in else's body.

Syntax:

if < conditional expression> :
    if < conditional expression> :
        statements
    else:
        statements
elif <conditional expression>:
    statements to be executed
else:
    statements to be executed

3. **Iteration** : The iteration statements or repetition statements allow a **set of instructions to be performed repeatedly until a certain condition is fulfilled.**

The iteration statements are also called as Loops or Looping Statements.

(a) **while loop :** A while loop is a **conditional loop** that will repeat the instructions within itself as long as the condition remains True.

Syntax :

Initialization
while < logical expression> :
    body of loop
    .....
    .....
    update counter variable

An infinite loop is a looping construct that **does not terminate** the loop and **executes the loop forever.**

(b) **for loop :** The for loop of a Python is designed to process the items of any sequence, such as a list or a string, one by one.

Syntax:

for variable in <sequence> :
    statements to repeat

- **Jump Statements :** Python offers two jump statements to be used within loops to jump out of loop iterations.

(a) **break :** The *break* statement enables a program skip over a part of the code.

A *break* statement terminates the loop it lies within. Execution resumes at the statement immediately following the body of terminated statement.

(b)continue : Instead of forcing termination, the continue statement forces the next iteration of the loop to take place, skipping any code in between. The continue statement skips the rest of the loop statements and causes the next iteration of the loop to take place.

- **Loop else** - The else clause of a Python loop executes only when the loop ends normally, not when the loop is terminating because of a break statement.
- **Nested for loop :** When a loop contains another loop inside its body, it is called Nested loop. In a nested loop, the inner loop must terminate before the outer loop.

For each value of outer loop, inner loop will execute till the condition becomes false.

➢ String Manipulation :

- A string is a sequence of characters enclosed in single, double or triple quotes.
- Indexing is used for accessing individual characters within a string.
- The first character has the index 0 and the last character has the index n-1 where n is the length of the string. The negative indexing ranges from -n to -1.
- Strings in Python are immutable, i.e., a string cannot be changed after it is created.
- String Operators : Two basic operators + and * are allowed. + is used for concatenation (joining). * is used for replication (repetition). You cannot add number and string using +. You cannot multiply string and string using * Only number*number or string*number is allowed.
- Membership operator *'in'* takes two strings and returns True if the first string appears as a substring in the second else returns False. There are many **built-in functions for working with strings in Python.**

Method	Description	Example

len ( )	Returns the length of the given string	>>> str1 = 'Hello World!'
		>>> len(str1)
		12

title()	Returns the string with first letter of every word in the string in uppercase and rest in lowercase	>>> str1 = 'hello WORLD!'
		>>> str1.title()
		'Hello World!'

lower()	Returns the string with all uppercase letters converted to lowercase	>>> str1 = 'hello WORLD!'
		>>> str1.lower()
		'hello world!'

upper()	Returns the string with all lowercase letters converted to uppercase	>>> str1 = 'hello WORLD!'
		>>> str1.upper()
		'HELLO WORLD!'

count(str, start, end)	Returns number of times substring str occurs in the given string. If we do not give start index and end index then searching starts from	>>> str1 = 'Hello World! Hello Hello'
		>>> str1.count('Hello',12,25)
		2
		>>> str1.count('Hello')

	index 0 and ends at length of the string	3

find(str,start, end)	Returns the first occurrence of index of substring str occurring in the given string. If we do not give start and end then searching starts from index 0 and ends at length of the string. If the substring is not present in the given string, then the function returns -1	>>> str1 = 'Hello World! Hello Hello'
		>>> str1.find('Hello',10,20)
		13
		>>> str1.find('Hello',15,25)
		19
		>>> str1.find('Hello')
		0
		>>> str1.find('Hee')
		-1

index(str, start, end)	Same as find() but raises an exception if the substring is not present in the given string	>>> str1 = 'Hello World! Hello Hello'
		>>> str1.index('Hello')
		0
		>>>str1.index('Hee')
		ValueError: substring not found

endswith()	Returns True if the given string ends with the supplied substring otherwise returns False	>>> str1 = 'Hello World!'
		>>>str1.endswith('World!') True
		>>> str1.endswith('!')
		True
		>>> str1.endswith('lde')
		False

startswith()	Returns True if the given string starts with the supplied substring otherwise returns False	>>> str1 = 'Hello World!'
		>>> str1.startswith('He')
		True
		>>> str1.startswith('Hee')
		False

isalnum()	Returns True if characters of the given string are either alphabets or numeric. If whitespace or special symbols are part of the given string or the string is empty it returns False	>>> str1 = 'HelloWorld'
		>>>str1.isalnum()
		True
		>>> str1 = 'HelloWorld2'
		>>>str1.isalnum()
		True
		>>> str1 = 'HelloWorld!!'
		>>> str1.isalnum()
		False

islower()	Returns True if the string is non-empty and has all lowercase alphabets, or has at least one character as lowercase alphabet and rest are non-alphabet characters	>>> str1 = 'hello world!'
		>>>str1.islower()
		True
		>>> str1 = 'hello 1234'
		>>>str1.islower()
		True
		>>> str1 = 'hello ??'
		>>> str1.islower()
		True
		>>>str1='1234'
		>>>str1.islower()
		False
		>>> str1 = 'Hello World!'
		>>> str1.islower()
		False

isupper()	Returns True if the string is non-empty and has all uppercase	>>> str1 = 'HELLO WORLD!' >>> str1.isupper()

	alphabets, or has at least one character as uppercase character and rest are non-alphabet characters	True
		>>> str1 = 'HELLO 1234'
		>>> str1.isupper()
		True
		>>> str1 = 'HELLO ??'
		>>>str1.isupper()
		True
		>>> str1 = '1234'
		>>> str1.isupper()
		False
		>>> str1 = 'Hello World!'
		>>> str1.isupper()
		False

isspace()	Returns True if the string is non-empty and title case, i.e., the first letter of every word in the string in uppercase and rest in lowercase	>>> str1 = ' \n \t \r'
		>>> str1.isspace()
		True
		>>> str1 = 'Hello \n'
		>>> str1.isspace()
		False

istitle()	Returns True if the string is non-empty and title case, i.e., the first letter of every word in the string in uppercase and rest in lowercase	>>> str1 = 'Hello World!'
		>>> str1.istitle()
		True
		>>> str1 = 'hello World!'
		>>> str1.istitle()
		False

lstrip()	Returns the string after removing the spaces only on the left of the string	>>> str1 = ' Hello World! '
		>>> str1.lstrip()
		'Hello World! '

rstrip()	Returns the string after removing the spaces only on the right of the string	>>> str1 = ' Hello World! '
		>>> str1.rstrip()
		' Hello World!'

strip()	Returns the string after removing the spaces both on the left and the right of the string	>>> str1 = ' Hello World! '
		>>> str1.strip()
		'Hello World!'

replace(oldstr, newstr)	Replaces all occurrences of old string with the new string	>>> str1 = 'Hello World!'
		>>> str1.replace('o','*')
		'Hell* W*rld!'
		>>> str1 = 'Hello World!'
		>>>str1.replace('World','Country')
		'Hello Country!'
		>>> str1 = 'Hello World! Hello'
		>>>str1.replace('Hello','Bye')
		'Bye World! Bye'

join()	Returns a string in which the characters in the string have been joined by a separator	>>> str1 = ('HelloWorld!')
		>>> str2 = '-' #separator
		>>> str2.join(str1)
		'H-e-l-l-o-W-o-r-l-d-!'

partition()	Partitions the given string at the first occurrence of the substring (separator) and returns the string partitioned into three parts. 1. Substring before the separator 2. Separator 3. Substring after the	>>> str1 = 'India is a Great Country'
		>>> str1.partition('is') ('India ', 'is', ' a Great Country')
		>>> str1.partition('are')
		('India is a Great Country','','')

	separator If the separator is not found in the string, it returns the whole string itself and two empty strings	

split()	Returns a list of words delimited by the specified substring. If no delimiter is given then words are separated by space.	>>> str1 = 'India is a Great Country'
		>>>str1.split() ['India','is','a','Great', 'Country'] >>> str1 = 'India is a Great Country'
		>>> str1.split('a')
		['Indi', ' is ', ' Gre', 't Country']

List Manipulation :

- Lists are mutable sequences in Python, i.e., we can change the elements of the list.
- Elements of a list are put in square brackets separated by comma.
- A list within a list is called a nested list. List indexing is same as that of strings and starts at 0. Two way indexing allows traversing the list in the forward as well as in the backward direction.
- Operator + concatenates one list to the end of other list.
- Operator * repeats a list by specified number of times.
- Membership operator in tells if an element is present in the list or not and not in does the opposite.
- Slicing is used to extract a part of the list.
- There are many **list manipulation functions** including: len(), list(), append(), extend(), insert(), count(), find(), remove(), pop(), reverse(), sort(), sorted(), min(), max(), sum().

Method	Description	Example
len()	Returns the length of the list passed as the argument	>>>list1 = [10,20,30,40,50] >>> len(list1) 5

list()	Creates an empty list if no argument is passed.
	Creates a list if a sequence is passed as an argument.
		>>> list1 = list()
		>>> list1
		[ ]
		>>> str1 = 'aeiou'
		>>> list1 = list(str1)
		>>> list1
		['a', 'e', 'i', 'o', 'u']

append()	Appends a single element passed as an argument at the end of the list The single element can also be a list	>>> list1 = [10,20,30,40]
		>>> list1.append(50)
		>>> list1
		[10, 20, 30, 40, 50]
		>>> list1 = [10,20,30,40]
		>>> list1.append([50,60])
		>>> list1
		[10, 20, 30, 40, [50, 60]]

extend()	Appends each element of the list passed as argument to the end of the given list	>>> list1 = [10,20,30]
		>>> list2 = [40,50]
		>>> list1.extend(list2)
		>>> list1
		[10, 20, 30, 40, 50]

insert()	Inserts an element at a particular index in the list	>>>list1 = [10,20,30,40,50]
		>>> list1.insert(2,25)
		>>> list1
		[10, 20, 25, 30, 40, 50]

		>>> list1.insert(0,5)
		>>> list1
		[5, 10, 20, 25, 30, 40, 50]

count()	Returns the number of times a given element appears in the list	>>>list1=[10,20,30,10,40,10]
		>>> list1.count(10)
		3
		>>> list1.count(90)
		0

index()	Returns index of the first occurrence of the element in the list. If the element is not present, ValueError is generated	>>>list1=[10,20,30,20,40,10] >>> list1.index(20) 1 >>>list1.index(90) ValueError: 90 is not in list

remove()	Removes the given element from the list. If the element is present multiple times, only the first occurrence is removed. If the element is not present, then ValueError is generated	>>>list1=[10,20,30,40,50,30]
		>>> list1.remove(30)
		>>> list1
		[10, 20, 40, 50, 30]
		>>>list1.remove(90)
		ValueError:list.remove(x):x not in list

pop()	Returns the element whose index is passed as parameter to this function and also removes it from the list. If no parameter is given, then it returns and removes the last element of the list	>>>list1=[10,20,30,40,50,60]
		>>> list1.pop(3)
		40
		>>> list1
		[10, 20, 30, 50, 60]
		>>>list1=[10,20,30,40,50,60]
		>>> list1.pop()
		60
		>>> list1
		[10, 20, 30, 40, 50]

reverse()	Reverses the order of elements in the given list	>>>list1=[34,66,12,89,28,99]
		>>> list1.reverse()
		>>> list1
		[99, 28, 89, 12, 66, 34]
		>>> list1 = ['Tiger' ,'Zebra' , 'Lion' , 'Cat' ,'Elephant' ,'Dog']
		>>> list1.reverse()
		>>> list1
		['Dog', 'Elephant', 'Cat', 'Lion', 'Zebra', 'Tiger']

sort()	Sorts the elements of the given list in-place	>>>list1= ['Tiger','Zebra','Lion', 'Cat', 'Elephant' ,'Dog']
		>>> list1.sort()
		>>> list1 ['Cat', 'Dog', 'Elephant', 'Lion', 'Tiger', 'Zebra']
		>>>list1=[34,66,12,89,28,99]
		>>> list1.sort(reverse = True)
		>>> list1
		[99,89,66,34,28,12]

sorted()	It takes a list as parameter and creates a new list consisting of the same elements arranged in sorted order	>>>list1=[23,45,11,67,85,56]
		>>> list2 = sorted(list1)
		>>> list1
		[23, 45, 11, 67, 85, 56]
		>>> list2

min()	Returns minimum or smallest element of the list	[11, 23, 45, 56, 67, 85] >>>list1=[34,12,63,39,92,44] >>> min(list1) 12
max()	Returns maximum or largest element of the list	>>> max(list1) 92
sum()	Returns sum of the elements of the list	>>> sum(list1) 284

Tuples :

- Tuples are immutable sequences, i.e., we cannot change the elements of a tuple once it is created.
- Elements of a tuple are put in round brackets separated by commas.
- If a sequence has comma separated elements without parentheses, it is also treated as a tuple.
- Tuples are ordered sequences as each element has a fixed position. Indexing is used to access the elements of the tuple; two way indexing holds in dictionaries as in strings and lists.
- Operator '+' adds one sequence (string, list, tuple) to the end of other.
- Operator '*' repeats a sequence (string, list, tuple) by specified number of times
- Membership operator 'in' tells if an element is present in the sequence or not and 'not in' does the opposite.
- **Tuple manipulation functions are**: len(), tuple(), count(), index(), sorted(), min(), max(),sum().

Method	Description	Example

len()	Returns the length or the number of elements of the tuple passed as the argument	>>> tuple1 = (10,20,30,40,50)
		>>> len(tuple1)
		5

tuple()	Creates an empty tuple if no argument is passed Creates a tuple if a sequence is passed as argument	>>> tuple1 = tuple()
		>>> tuple1 ( )
		>>>tuple1=tuple('aeiou')#string >>> tuple1
		('a', 'e', 'i', 'o', 'u')
		>>> tuple2 = tuple([1,2,3]) #list
		>>> tuple2
		(1, 2, 3)
		>>> tuple3 = tuple(range(5))
		>>> tuple3
		(0, 1, 2, 3, 4)

count()	Returns the number of times the given element appears in the tuple	>>>tuple1=(10,20,30,10,40,10,50)
		>>> tuple1.count(10)
		3
		>>> tuple1.count(90)
		0

index()	Returns the index of the first occurrence of the element in the given tuple	>>> tuple1 = (10,20,30,40,50)
		>>> tuple1.index(30)
		2
		>>> tuple1.index(90)
		ValueError: tuple.index(x):
		x not in tuple

sorted()	Takes elements in the tuple and returns a new sorted list. It should be noted that, sorted() does not make any change to the original tuple	>>>tuple1=("Rama","Heena","Raj", "Mohsin","Aditya") >>> sorted(tuple1) ['Aditya', 'Heena', 'Mohsin', 'Raj', 'Rama']

min()	Returns minimum or smallest element of the tuple	>>> tuple1 = (19,12,56,18,9,87,34) >>> min(tuple1) 9
max()	Returns maximum or largest element of the tuple	>>> max(tuple1) 87
sum()	Returns sum of the elements of the tuple	>>> sum(tuple1) 235

➢ **Dictionary :**

- Dictionary is a mapping (non-scalar) data type.
- It is an unordered collection of key-value pair: key value pair are put inside curly braces.
- Each key is separated from its value by a colon.
- Keys are unique and act as the index.
- Keys are of immutable type but values can be mutable.

Method	Description	Example
len()	Returns the length or number of key: value pairs of the dictionary passed as the argument	>>>dict1={'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85} >>> len(dict1) 4

dict()	Creates a dictionary from a sequence of key-value pairs	pair1=[('Mohan',95),('Ram',89), ('Suhel',92), ('Sangeeta',85)]
		>>> dict1 = dict(pair1)
		>>> dict1
		{'Mohan': 95, 'Ram': 89, 'Suhel': 92, 'Sangeeta': 85}

keys()	Returns a list of keys in the dictionary	>>>dict1={'Mohan':95, 'Ram':89,'Suhel':92, 'Sangeeta':85} >>>dict1.keys() dict_keys(['Mohan','Ram', 'Suhel', 'Sangeeta'])
values()	Returns a list of values in the dictionary	>>>dict1={'Mohan':95, 'Ram':89,'Suhel':92, 'Sangeeta':85} >>>dict1.values() dict_values([95, 89, 92, 85])
items()	*Returns a list of tuples(key – value) pair*	>>>dict1={'Mohan':95, 'Ram': 89,'Suhel':92, 'Sangeeta':85} >>> dict1.items() dict_items([('Mohan', 95), ('Ram', 89), ('Suhel', 92), ('Sangeeta', 85)])

get()	*Returns the value corresponding to the key passed as the argument If the key is not present in the dictionary it will return None*	>>> dict1 = {'Mohan':95, 'Ram':89, 'Suhel':92, 'Sangeeta':85}
		>>> dict1.get('Sangeeta')
		85
		>>> dict1.get('Sohan')
		>>>

update()	*appends the key-value pair of the dictionary passed as the argument to the key-value pair of the given dictionary*	>>>dict1={'Mohan':95, 'Ram': 89,'Suhel':92, 'Sangeeta': 85}
		>>>dict2= 'Sohan':79,'Geeta':89}
		>>> dict1.update(dict2)
		>>> dict1
		{'Mohan': 95, 'Ram': 89, 'Suhel': 92, 'Sangeeta': 85, 'Sohan': 79, 'Geeta': 89}
		>>> dict2
		{'Sohan': 79, 'Geeta': 89}

del()	Deletes the item with the given key To delete the dictionary from the memory we write: del Dict_name	>>> dict1 = {'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85}
		>>> del dict1['Ram']
		>>>dict1
		{'Mohan':95,'Suhel':92, 'Sangeeta': 85}
		>>> del dict1 ['Mohan']
		>>> dict1
		{'Suhel': 92, 'Sangeeta': 85}
		>>> del dict1
		>>> dict1
		NameError: name 'dict1' is not defined

clear()	Deletes or clear all the items of the dictionary	>>>dict1={'Mohan':95,'Ram':89, 'Suhel':92, 'Sangeeta':85}
		>>> dict1.clear()
		>>> dict1
		{}

Python Exceptions:

Many times though, a program results in an error after it is run even if it doesn't have any syntax error. Such an error is a runtime error, called an exception. A number of built-in exceptions are defined in the Python library.

Some common error types.

- **IndexError** is thrown when trying to access an item at an invalid index.

```
>>> L1=[1,2,3]
>>> L1[3]
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
L1[3]
IndexError: list index out of range
```

- **ModuleNotFoundError** is thrown when a module could not be found.

```
>>> import notamodule
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

- **ImportError** is thrown when a specified function can not be found.

```
>>> from math import cube
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
from math import cube
ImportError: cannot import name 'cube'
```

- **KeyError** is thrown when a key is not found. This exception is raised when a mapping (dictionary) key is not found in the set of existing keys.

```
>>> D1={'1':"aa", '2':"bb", '3':"cc"}
>>> D1['4']
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
D1['4']
KeyError: '4'
```

- **TypeError** is thrown when an operation or function is applied to an object of an inappropriate type.

```
>>> '2'+2
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
'2'+2
TypeError: must be str, not int
```

- **ValueError** is thrown when a function's argument is of an inappropriate type.

```
>>> int('xyz')
Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>
int('xyz')
ValueError: invalid literal for int() with base 10: 'xyz'
```

- **NameError** is thrown when an object could not be found. This exception is raised when a local or global name is not found.

```
>>> age
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
age
NameError: name 'age' is not defined
```

- **ZeroDivisionError** is thrown when the second operator in the division is zero.

```
>>> x=100/0
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
x=100/0
ZeroDivisionError: division by zero
```