**Variable:** name of memory location used to store values.

**Identifier:** name of object/variable etc. simply used to give name.

1. Can be used number, alphabets, underscore
2. Cannot start with number

**Standard data types**

**Immutable:** If mutable, then we can add, remove, or change the data.

1. Numbers
2. int (signed integer)
3. float (real numbers)
4. complex numbers
5. Strings
6. Tuples
7. bool

**Mutable:** We can not edit, change the values

1. List
2. Sets
3. Dictionaries

**##########################################################################**

**# List #**

**##########################################################################**

List is a fundamental, built-in data structure used to store **an ordered, mutable, heterogeneous and duplicate values. collection of items**. Key feature are:-

* **Ordered**: Lists maintain the order in which items are added. This means items can be accessed by their position (index), starting from 0 for the first element.
* **Mutable**: Lists are changeable, meaning their elements can be modified, added, or removed after the list has been created.
* **Allows Duplicates**: A list can contain multiple instances of the same item.
* Can Store Mixed Data Types: Lists are highly flexible and can store elements of different data types within the same list (e.g., integers, strings, booleans, or even other lists).
* **Defined by Square Brackets**: Lists are created by enclosing a comma-separated sequence of items within square brackets [].

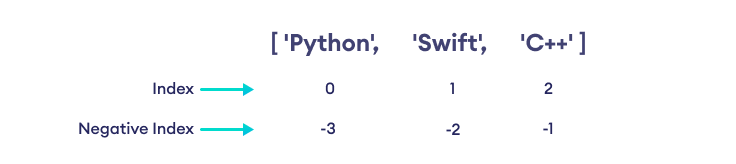
**Indexing in list**

1. **Indexing Lists (+ve index)**

* Each item in a list corresponds to an index number, which is an integer value, starting with the index number 0.

1. **Indexing Lists (-ve index)**

* In Python, a list can also have negative indices.
* The index of the last element is -1, the second last element is -2 and so on.



**Creating list**

We can create list in any of below method-

1. **Using list function** –

var\_name = list( values\_optional )

1. Using square bracket

var\_name = [ values\_optional ]

**Add Elements to a Python List**

To add element in list we have below two method-

1. **append()**
2. **insert()**
3. **extend()**
4. **+ operator**

**append()**

* This method appends/adds element to **the end of existing list and returns None**.

list\_var.append( another\_variable)

**insert()**

* The insert() method is used to **insert the given value at given index** and returns None
* If a element is already present on given index then it is moved to next position ( right shift)
* If trying to insert as index > len(list) then behaves as append()

list\_var.insert('index',value)

# create a list of prime numbers

prime\_numbers = [2, 3, 5, 7]

# insert 11 at index 4

prime\_numbers.insert(2, 'Hello')

print('List:', prime\_numbers)

#List: [2, 3, 5, 7, 11] Hello is inserted at index 2 and elements starting at index 2 are right shifted

**extend()**

* The extend() method adds all the items of the specified iterable, such as list, tuple, dictionary, or string , to the end of a list and return None.
* It *iterates through each individual element within the provided iterable. For each element encountered during this iteration, extend() appends that element to the end of the list on which the method was called.*

**list1.extend( iterable )**

Example1

languages = ['French']

languages\_tuple = ('Spanish', 'Portuguese')

# add items of the tuple to the languages list

languages.extend(languages\_tuple)

print( languages) #['French', 'Spanish', 'Portuguese']

languages\_set = {'Chinese', 'Japanese'}

# add items of the set to the languages list

languages.extend(languages\_set)

print(languages) #['French', 'Spanish', 'Portuguese', 'Chinese', 'Japanese']

Example2

languages = ['French']

languages\_tuple = [('Spanish', 'Portuguese')]

# add items of the tuple to the languages list

languages.extend(languages\_tuple)

print( languages)# ['French', ('Spanish', 'Portuguese')] **# iterate and append thru each val**

**Example3: -- Interview**

What will be output of below code.

person = ['hello']

s='morming'

person.extend(s)

print(person)

Answer

print(person)#['hello', 'm', 'o', 'r', 'm', 'i', 'n', 'g'] #it accepts iterable data

**Explanation:**

Behaviour of update method. It takes argument as iterable.

**+ operator**

It behaves same as extend().

**#Python extend() Vs append()#**

* If need to add the item/iterable data itself (rather than its elements), use the append() method.
* If need to add **each value (not nested level) of item/iterable data** then use the extend() method.

**Accessing element from list**

We can access element using indices.

* Positive Indexing
* Negative Indexing
* Slicing operator
* By iteration ( for, while loop )

**List Methods**

Here are some other common list methods.

* list.append(elem) -- Already done above
* list.insert(index, elem) -- Already done above
* list.extend(list2) --Already done above
* list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
* list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
* list.sort() -- sorts the list in place (does not return it). (The sorted() function shown later is preferred.)
* list.reverse() -- reverses the list in place (does not return it)
* list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).
* Del list[index]: delete values from given index index

**##########################################################################**

**# Set #**

**##########################################################################**

Set is an **unordered collection of unique, immutable elements**.

It's a fundamental data structure, like a mathematical set, offering efficient ways to store and manage distinct items. Key feature:

* **Unordered**
* **Unique Elements**
* **Mutable (but elements must be immutable)**
* **No Indexing or Slicing**
* **Efficient Membership Testing**
* **Heterogeneous**

**Sets support various mathematical set operations, including:**

* **Union (| or union()):** Returns a new set containing all unique elements from both sets.
* **Intersection (& or intersection()):** Returns a new set containing only the common elements between two sets.
* **Difference (- or difference()):** Returns a new set containing elements present in the first set but not in the second.
* **Symmetric Difference (^ or symmetric\_difference()):** Returns a new set containing elements that are in either set but not in both.

**Create a Set in Python**

* Using set() --- > s= set( values separated by comma)
* By placing the element is set in curly braces --- > s= { values separated by comma )

**Create an Empty Set in Python**

* Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python.
* To make a set without any elements, we use the set() function without any argument.

**Example:**

# create an empty set

empty\_set = set()

# create an empty dictionary

empty\_set2 = { }

# check data type of empty\_set

print('Data type of empty\_set:', type(empty\_set)) # set

# check data type of dictionary\_set

print('Data type of empty\_dictionary:', type(empty\_set2)) # dictionary

**Adding elements to set**

To add elements in python set we can use below two methods:

* add
* update()

1. **add()**

* The add() method adds a given element to set. If the element is already present, it does not add any element.

**set.add( single\_element )**

**Example:**

# set of vowels

vowels = {'a', 'e', 'i', 'u'}

# adding 'o'

vowels.add('o')

print('Vowels are:', vowels)#Vowels are: {'a', 'u', 'e', 'i', 'o'}

1. **update() -- Interview**

* The Python set update() method updates the set, adding items from other iterables and returns None.
* The update() method can take any number of arguments

**set\_var.update( itertable\_var1, itertable\_var2, . . . )**

**Example1:**

A = {1, 3, 5}

B = {2, 4, 6}

C = {0}

print('Original A:', A) # {1, 3, 5}

# adds items of B and C to A and updates A

A.update(B, C)

print('A after update()', A) # {1, 3, 5, 0}

A.update(9)

print('A after second update()', A) **# Error because update accepts iterable data**

**Example2: -- Interview**

person = {'hello'}

s='morming'

person.update(s)

print(person**)#{'n', 'g', 'i', 'hello', 'm', 'Hii', 'r', 'o'} # will treat s as iterable**

**Deleting elements from set**

To delete the elements from set we have below two methds:-

1. **discard( element )**
2. **remove( element )**

**remove vs discard in set**

The key difference is remove() method will raise an error if the specified item does not exist, and the discard() method will not.

**Built-in methods of set**

|  |  |
| --- | --- |
| Function | Description |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Returns True if all elements of the set are true (or if the set is empty). |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Returns True if any element of the set is true. If the set is empty, returns False. |
| [enumerate()](https://www.programiz.com/python-programming/methods/built-in/enumerate) | Returns an enumerate object. It contains the index and value for all the items of the set as a pair. |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Returns the length (the number of items) in the set. |
| [max()](https://www.programiz.com/python-programming/methods/built-in/max) | Returns the largest item in the set. |
| [min()](https://www.programiz.com/python-programming/methods/built-in/min) | Returns the smallest item in the set. |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Returns a new sorted list from elements in the set(does not sort the set itself). |
| [sum()](https://www.programiz.com/python-programming/methods/built-in/sum) | Returns the sum of all elements in the set. |

**Slicing in Python**

[**https://www.youtube.com/watch?v=ajrtAuDg3yw**](https://www.youtube.com/watch?v=ajrtAuDg3yw)

* Slicing in Python is a method for extracting specific portions of sequences from ietrable data (list, tuple,string etc) which stores data on indexes.
* Slicing is done using bracket notation that looks like this: var[start:end:step]

**start**: Index where the slice begins (inclusive). Defaults to 0 if omitted.

**stop**: Index where the slice ends (exclusive). Defaults to the sequence's length if omitted.

**step**: Determines the interval between elements. +ve means left or right, -ve means right to left. Defaults to 1 if omitted.

**Thumb rule for slicing:**

* Write the -ve and +ve index both.
* Based on step value go in left to right or right to left.

Example:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| values | 1 | hello | 3.14 | TRUE | [1,2] | {"key": "value"} | (4,5) | None | FALSE | world |
| +ve index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| -ve index | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

**Question:**

Find the output of below lines.

l = [1,"hello",3.14,True,[1, 2], {"key": "value"},(4, 5), None,False,  "world" ]

print(l[3:-2])# start at 3 ends at -2 , left to right

print(l[-4:-1:])# start at -4 ends at -1 , left to right

print(l[-4:-1:-1])# start at -4 ends at -1 , right to left

print(l[-3:-1])#start at -3, end at -1 , left to right direction

print(l[-3:1:-1])#start at -3, end at 1 , right to left

Answers

print(l[3:-2])#[True, [1, 2], {'key': 'value'}, (4, 5), None]

print(l[-4:-1:])# [(4, 5), None, False]

print(l[-4:-1:-1])# []

print(l[-3:-1])# [None, False]

print(l[-3:1:-1])# [None, (4, 5), {'key': 'value'}, [1, 2], True, 3.14]

**##########################################################################**

**# Dictionary #**

**##########################################################################**

* A Python dictionary is a built-in data type that **stores data in key-value pairs**.
* It is a **mutable, ordered collection (as of Python 3.7)** where each unique key maps to a specific value. Dictionaries are also known as associative arrays or hash maps in other programming languages.

**Key Characteristics:**

* **Key-Value Pairs**: Data is organized as key: value pairs.
* **Unique and immutable Keys:** Each key within a dictionary must be unique and immutable
* **Mutable Values:** Values can be of any data type and can be modified.
* **Ordered (Python 3.7+):** Dictionaries maintain the insertion order of items.
* Mutable: Dictionaries can be modified after creation (items can be added, removed, or updated).

**Creating a Dictionary:**

* Using curly braces --> **d={'key1' : ; val1', . . }** # key-value pair are optional
* Using dict() -- > d= dict('key1'='val1', . . ) # key and value are optional

**Add Items to a Dictionary**

* We can add key-value pair using square bracket as - **dict\_var [ key ]= value**

**Remove Dictionary Items**

* We can delete key-value pair from dictionary usinf del function.

**del dict\_var [ key\_name ]**

**Change/Update Dictionary Items**

* We can update the key-value pair using square bracket.
* We key is already present then update the values else create new key-value pair

**dict\_var [ key ] = new\_val**

**Dictionary methods**

* We have below dictionary methods which are commonly used.
* All of these are instance methods**. (need to call on dictionary variable)**

|  |  |
| --- | --- |
| [pop(key, [val ] )](https://www.programiz.com/python-programming/methods/dictionary/pop) | * Removes the item with the specified key. * If key not present then return value (val) * If key not present and val not given then error. |
| [update( dict\_var )](https://www.programiz.com/python-programming/methods/dictionary/update) | Adds or changes dictionary items. |
| [clear()](https://www.programiz.com/python-programming/methods/dictionary/clear) | Remove all the items from the dictionary. |
| [keys()](https://www.programiz.com/python-programming/methods/dictionary/keys) | Returns all the dictionary's keys. |
| [values()](https://www.programiz.com/python-programming/methods/dictionary/values) | Returns all the dictionary's values. |
| [get()](https://www.programiz.com/python-programming/methods/dictionary/get) | Returns the value of the specified key. |
| [popitem()](https://www.programiz.com/python-programming/methods/dictionary/popitem) | Returns the last inserted key and value as a tuple. |
| [copy()](https://www.programiz.com/python-programming/methods/dictionary/copy) | Returns a copy of the dictionary. |
| get(key[, val]) | * If key is present in dictionary the return it's value * If **key is not present then return value (val ).** * If key not present and val not given the None. |
|  |  |

**##########################################################################**

**# Tuple #**

**##########################################################################**

A tuple in Python is an ordered and **immutable collection of items.** Feature: -

1. **Ordered** *(stored on index, index start with 0 )*
2. **Immutable**
3. **Allows duplicate values**
4. **Heterogeneous**

**Note**:

1. As tuple is immutable then we can not make any change in it. If we do then create a new tuple.
2. If creating a tuple with **single element then need to have comma** at end in tuple value.
3. Tuples can **contain elements of various data types, including other tuples, lists, dictionaries and even functions**.

**Example:**

# An empty tuple

empty\_tuple = ()

# A tuple with various data types

my\_tuple = ("apple", 3, True, 5.0)

# A tuple with a single item (requires a comma)

single\_item\_tuple = ("banana",)

Question:

What will be output of below line-

t1=('hello')

t2=('hello',)

print(type(t1))# string – single element not with comms

print(type(t2)) # tuple

**Python Tuple Basic Operations**

* Accessing of Python Tuples
* Concatenation of Tuples
* Slicing of Tuple – Already above
* Deleting a Tuple

**Accessing of Python Tuples**

We can access the elements of a tuple by using indexing and slicing,

**Concatenation of Tuples**

Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

**Slicing of Tuple**

Slicing a tuple means creating a new tuple from a subset of elements of the original tuple.

The slicing syntax is tuple:- [start : stop : step].

**Deleting a Tuple**

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using del keyword.

**del variable\_name**

tup = (0, 1, 2, 3, 4)

del tup

print(tup)# error becuase tup is deleted

**#########################################################################**

**# OPERATOR IN PYTHON #**

**#########################################################################**

Operator is a symbol that performs certain operations.

Python provides the following set of operators

1. Arithmetic Operators
2. Relational Operators or Comparison Operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Special operators

**1. Arithmetic Operators:**

* + ==> Addition
* - ==> Subtraction
* \* ==> Multiplication
* / ==> Division operator
* % ===> Modulo operator
* // ==> Floor Division operator
* \*\* ==>Exponent operator or power operator

**Note:**

* Division **/ operator always performs floating point arithmetic**. Hence it will always returns
* float value.
* Floor division **(//) can perform both floating point and integral arithmetic**. Floating point is if at **least one argument is float type** .

**Ternary Operator**

**x = on\_true if condition else on\_false** < ----- Syntax

**eval() function**

eval Function take a String and evaluate the Result.

**#########################################################################**

**# Command Line Argument #**

**#########################################################################**

**###############**

**# 1. Using sys.argv #**

**###############**

* In **python's sys module we have argv variable which gives a list containing the command-line argument**
* sys.argv[0] always contains the name of the script being executed.
* Subsequent elements (sys.argv[1], sys.argv[2], etc.) contain the arguments passed to the script.
* Each argument while calling python script is treated as one parameter by argv.

Example: **my\_file.py --> called as -----** python my\_file.py Alice

import sys

if len(sys.argv) > 1:

    script\_name=sys.argv[0]

    name = sys.argv[1]

    print(f"Python file name, {script\_name}!")

    print(f"Hello, {name}!")

else:

    print("Hello, world!")

**Output**

PS D:\Learnings\LeetCode> python my\_file.py Alice

Python file name, my\_file.py!

Hello, Alice!

**######################**

**# 2. Using argparse module #**

**######################**

Before starting with argparse module lets start with positional argument and flag argument.

**Positional Argument**

* **Definition**: An argument that is identified by its position in the command line, not by a flag.

**Flag Argument**

* **Definition**: An argument that is identified by a flag like --name or --age.

**Example: Let say we are calling below code-**

python my\_file.py 101 --name alice --age 30

* 101 is positional argument ( passed directly without a flag—it’s matched by position)
* 'name' and 'age' are flag

**Note:**

* All positional argument is mandatory.
* All flag argument is optional ( if not marked as required=True in add\_argument function )
* Flag argument values are always assigned to it's name variable
* Positional arguments are assigned based on the order /position of assigning the values to variable

**# argparse module #**

* The 'argparse' module in Python helps create a program in a command-line-environment in a way that appears not only easy to code but also improves interaction.
* The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.
* To use argparse module for argument parsing we should follow below steps-
  + **Creating parser ( ArgumentParser objects)**
  + **Adding argument**

1. **ArgumentParser objects**

**class argparse.ArgumentParser**(prog=None, usage=None, description=None, epilog=None, parents=[], formatter\_class=argparse.HelpFormatter, prefix\_chars='-', fromfile\_prefix\_chars=None, argument\_default=None, conflict\_handler='error', add\_help=True, allow\_abbrev=True, exit\_on\_error=True, \*, suggest\_on\_error=False, color=True)

* [prog](https://docs.python.org/3/library/argparse.html#prog) - The name of the program (default: generated from the \_\_main\_\_ module attributes and sys.argv[0])
* [usage](https://docs.python.org/3/library/argparse.html#usage) - The string describing the program usage (default: generated from arguments added to parser)
* [description](https://docs.python.org/3/library/argparse.html#description) - Text to display before the argument help (by default, no text)
* [epilog](https://docs.python.org/3/library/argparse.html#epilog) - Text to display after the argument help (by default, no text)
* [parents](https://docs.python.org/3/library/argparse.html#parents) - A list of [ArgumentParser](https://docs.python.org/3/library/argparse.html" \l "argparse.ArgumentParser" \o "argparse.ArgumentParser) objects whose arguments should also be included
* [formatter\_class](https://docs.python.org/3/library/argparse.html#formatter-class) - A class for customizing the help output
* [prefix\_chars](https://docs.python.org/3/library/argparse.html#prefix-chars) - The set of characters that prefix optional arguments (default: ‘-‘)
* [fromfile\_prefix\_chars](https://docs.python.org/3/library/argparse.html#fromfile-prefix-chars) - The set of characters that prefix files from which additional arguments should be read (default: None)
* [argument\_default](https://docs.python.org/3/library/argparse.html#argument-default) - The global default value for arguments (default: None)
* [conflict\_handler](https://docs.python.org/3/library/argparse.html#conflict-handler) - The strategy for resolving conflicting optionals (usually unnecessary)
* [add\_help](https://docs.python.org/3/library/argparse.html#add-help) - Add a -h/--help option to the parser (default: True)
* [allow\_abbrev](https://docs.python.org/3/library/argparse.html#allow-abbrev) - Allows long options to be abbreviated if the abbreviation is unambiguous (default: True)
* [exit\_on\_error](https://docs.python.org/3/library/argparse.html#exit-on-error) - Determines whether or not ArgumentParser exits with error info when an error occurs. (default: True)
* [suggest\_on\_error](https://docs.python.org/3/library/argparse.html#suggest-on-error) - Enables suggestions for mistyped argument choices and subparser names (default: False)
* [color](https://docs.python.org/3/library/argparse.html#color) - Allow color output (default: True)

1. **Adding Arguments**

**ArgumentParserObject.add\_argument**(name or flags..., \*[, action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest][, deprecated])

* [name or flags](https://docs.python.org/3/library/argparse.html#name-or-flags) - Either a name or a list of option strings
  + ( variable **passed as name-value pair** while calling function).
* [action](https://docs.python.org/3/library/argparse.html#action) - The basic type of action to be taken when this argument is encountered at the command line.
* [nargs](https://docs.python.org/3/library/argparse.html#nargs) - The number of command-line arguments that should be consumed.
* [const](https://docs.python.org/3/library/argparse.html#const) - A constant value required by some [action](https://docs.python.org/3/library/argparse.html#action) and [nargs](https://docs.python.org/3/library/argparse.html#nargs) selections.
* [default](https://docs.python.org/3/library/argparse.html#default) - The value produced if the argument is absent from the command line and if it is absent from the namespace object.
* [type](https://docs.python.org/3/library/argparse.html#type) - The type to which the command-line argument should be converted.
* [choices](https://docs.python.org/3/library/argparse.html#choices) - A sequence of the allowable values for the argument.
* [**required**](https://docs.python.org/3/library/argparse.html#required) - Whether or not the command-line option may be omitted (optionals only).
* [help](https://docs.python.org/3/library/argparse.html#help) - A brief description of what the argument does.
* [metavar](https://docs.python.org/3/library/argparse.html#metavar) - A name for the argument in usage messages.
* [dest](https://docs.python.org/3/library/argparse.html#dest) - The name of the attribute to be added to the object returned by [parse\_args()](https://docs.python.org/3/library/argparse.html" \l "argparse.ArgumentParser.parse_args" \o "argparse.ArgumentParser.parse_args).
* [deprecated](https://docs.python.org/3/library/argparse.html#deprecated) - Whether or not use of the argument is deprecated.

**Example: my\_file.py**

import argparse

#

parser = argparse.ArgumentParser()

parser.add\_argument('first\_name')#mandatory, positional arg

parser.add\_argument('last\_name')#mandatory, positional arg

parser.add\_argument('--age',required=False)#optional b/c required=False

parser.add\_argument('--emp\_id',required=True)#mandatory b/c required=False

args = parser.parse\_args()

print(args.first\_name)

print(args.last\_name)

print(args.age)

print(args.emp\_id)

**What will be output when script is called like below:-**

**Example1:**

python my\_file.py John Doe --emp\_id=1001

**Output**

John

Doe

None

1001

**Example2:** Explanation – positional in order of assigned and flag by name- value pair ( check above Note)

python my\_file.py John Doe --emp\_id=1001

**Output**

John

Doe

None

1001