Python Programming

Unit 2: Collections & Functions

Strings

Python treats strings as contiguous series of characters delimited by single, double or even triple quotes. Python has a built-in string class named "str" that has many useful features. We can simultaneously declare and define a string by creating a variable of string type. This can be done in several ways which are as follows:

name = "India" graduate = 'N' country = name nationality = str("Indian")

Indexing: Individual characters in a string are accessed using the subscript ([ ]) operator. The expression in brackets is called an index. The index specifies a member of an ordered set and in this case it specifies the character we want to access from the given set of characters in the string.

The index of the first character is 0 and that of the last character is n-1 where n is the number of characters in the string. If you try to exceed the bounds (below 0 or above n-1), then an error is raised.

Strings

Traversing a String: **A string can be traversed by accessing character(s) from one index to another. For example, the following program uses indexing to traverse a string from first character to the last.**

Example:

```
message = "Hello!"
index = 0
for i in message:
print("message[", index, "] = ", i)
index += 1
```

OUTPUT

```
message[ 0 ] =  H
message[ 1 ] =  e
message[ 2 ] =  l
message[ 3 ] =  l
message[ 4 ] =  o
message[ 5 ] =  !
```

Concatenating, Appending and Multiplying Strings

Examples:

```
str1 = "Hello "
str2 = "World"
str3 = str1 + str2
print("The concatenated string is : ", str3)
```

OUTPUT

```
The concatenated string is :  Hello World
```

```
str = "Hello"
print(str * 3)
```

OUTPUT

```
Hello Hello Hello
```

```
str = "Hello, "
name = input("\n Enter your name : ")
str += name
str += ". Welcome to Python Programming."
print(str)
```

OUTPUT

```
Enter your name : Arnav
Hello, Arnav. Welcome to Python Programming.
```

Strings are Immutable

Python strings are immutable which means that once created they cannot be changed. Whenever you try to modify an existing string variable, a new string is created.

Example:

```
str1 = "Hello"
print("Str1 is : ", str1)
print("ID of str1 is : ", id(str1))

str2 = "World"
print("Str2 is : ", str2)

print("ID of str1 is : ", id(str2))
str1 += str2
print("Str1 after concatenation is : ", str1)
print("ID of str1 is : ", id(str1))
str3 = str1
print("str3 = ", str3)
print("ID of str3 is : ", id(str3))
```

OUTPUT

```
Str1 is :  Hello
ID of str1 is :  45093344
Str2 is :  World
ID of str1 is :  45093312
Str1 after concatenation is :  HelloWorld
ID of str1 is :  43861792
str3 =  HelloWorld
ID of str3 is :  43861792
```

String Formatting Operator

The % operator takes a format string on the left (that has %d, %s, etc) and the corresponding values in a tuple (will be discussed in subsequent chapter) on the right. The format operator, % allow users to construct strings, replacing parts of the strings with the data stored in variables. The syntax for the string formatting operation is:

"<Format>" % (<Values>)

Example:

```
name = "Aarish"
age = 8
print("Name = %s and Age = %d" %(name, age))
print("Name = %s and Age = %d" %("Anika", 6))
```

OUTPUT

```
Name = Aarish and Age = 8
Name = Anika and Age = 6
```

Built-in String Methods and Functions

Function	Usage	Example

`capitalize()`	This function is used to capitalize first letter of the string.	`str = "hello"`
		`print(str.capitalize())`
		OUTPUT
		`Hello`

`center(width, fillchar)`	Returns a string with the original string centered to a total of `width` columns and filled with `fillchar` in columns that do not have characters.	`str = "hello"`
		`print(str.center(10, '*'))`
		OUTPUT
		`**hello***`

`count(str, beg, end)`	Counts number of times str occurs in a string. You can specify beg as 0 and end as the length of the message to search the entire string or use any other value to just search a part of the string.	`str = "he"`
		`message = "helloworldhellohello"`
		`print(message. count (str,0, len (message)))`
		OUTPUT
		`3`

`endswith (suffix, beg, end)`	Checks if string ends with suffix; returns `True` if so and `False` otherwise. You can either set `beg = 0` and `end` equal to the length of the message to search entire string or use any other value to search a part of it.	`message = "She is my best friend"`
		`print(message.endswith("end", 0,len(message)))`
		OUTPUT
		`True`

Built-in String Methods and Functions

`find(str, beg, end)`	Checks if `str` is present in string. If found it returns the position at which `str` occurs in string, otherwise returns `-1`. You can either set `beg = 0` and `end` equal to the length of the message to search entire string or use any other value to search a part of it.	`message = "She is my best friend"`
		`print(message. find("my",0, len (message)))`
		OUTPUT
		`7`

`index(str, beg, end)`	Same as find but raises an exception if `str` is not found.	`message = "She is my best friend"`
		`print(message.index("mine", 0, len(message)))`
		OUTPUT
		`ValueError: substring not found`

`rfind(str, beg, end)`	Same as find but starts searching from the end.	`str = "Is this your bag?"`
		`print(str.rfind("is", 0, len(str)))`
		OUTPUT
		`5`

`rindex(str, beg, end)`	Same as `rindex` but start searching from the end and raises an exception if `str` is not found.	`str = "Is this your bag?"`
		`print(str.rindex("you", 0, len(str)))`
		OUTPUT
		`8`

Built-in String Methods and Functions

`isalnum()`	Returns True if string has at least 1 character and every character is either a number or an alphabet and False otherwise.	`message = "JamesBond007"`
		`print(message.isalnum())`
		OUTPUT
		`True`

`isalpha()`	Returns True if string has at least 1 character and every character is an alphabet and False otherwise.	`message = "JamesBond007"`
		`print(message.isalpha())`
		OUTPUT
		`False`

`isdigit()`	Returns True if string contains only digits and False otherwise.	`message = "007"`
		`print(message.isdigit())`
		OUTPUT
		`True`

`islower()`	Returns True if string has at least 1 character and every character is a lowercase alphabet and False otherwise.	`message = "Hello"`
		`print(message.islower())`
		OUTPUT
		`False`

`isspace()`	Returns True if string contains only whitespace characters and False otherwise.	`message = "  "`
		`print(message.isspace())`
		OUTPUT
		`True`

`isupper()`	Returns True if string has at least 1 character and every character is an upper case alphabet and False otherwise.	`message = "HELLO"`
		`print(message.isupper())`
		OUTPUT
		`True`

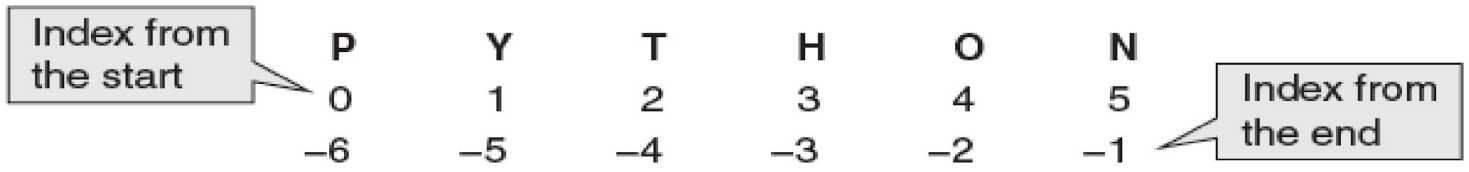
`len(string)`	Returns the length of the string.	`str = "Hello"`
		`print(len(str))`
		OUTPUT
		`5`

`ljust(width[, fillchar])`	Returns a string left-justified to a total of `width` columns. Columns without characters are padded with the character specified in the `fillchar` argument.	`str = "Hello"`
		`print(str.ljust(10, '*'))`
		OUTPUT
		`Hello*****`

`rjust(width[, fillchar])`	Returns a string right-justified to a total of `width` columns. Columns without characters are padded with the character specified in the `fillchar` argument.	`str = "Hello"`
		`print(str.rjust(10, '*'))`
		OUTPUT
		`*****Hello`

Slice Operation

A substring of a string is called a *slice*. The slice operation is used to refer to sub-parts of sequences and strings. You can take subset of string from original string by using [ ] operator also known as *slicing operator*.



Examples:

```
str = "PYTHON"
print("str[1:5] = ", str[1:5])   #characters starting at index 1 and extending up
to but not including index 5
print("str[:6] = ", str[:6])     # defaults to the start of the string
print("str[1:] = ", str[1:])      # defaults to the end of the string
print("str[:] = ", str[:])        # defaults to the entire string
print("str[1:20] = ", str[1:20])  # an index that is too big is truncated down to
length of the string
```

OUTPUT

```
str[1:5] =  YTHO
str[:6] =  PYTHON
str[1:] =  YTHON
str[:] =  PYTHON
str[1:20] =  YTHON
```

Programming Tip: Python does not have any separate data type for characters. They are represented as a single character string.

Specifying Stride while Slicing Strings

In the slice operation, you can specify a third argument as the *stride*, which refers to the number of characters to move forward after the first character is retrieved from the string. By default the value of stride is 1, so in all the above examples where he had not specified the stride, it used the value of 1 which means that every character between two index numbers is retrieved.

Examples:

```
str = "Welcome to the world of Python"
print("str[2:10] = ", str[2:10])    # default stride is 1
print("str[2:10:1] = ", str[2:10:1])   # same as stride = 1
print("str[2:10:2] = ", str[2:10:2])   # skips every alternate character
print("str[2:13:4] = ", str[2:13:4])   # skips every fourth character
```

OUTPUT

```
str[2:10] = lcome to
str[2:10:1] = lcome to
str[2:10:2] = loet
str[2:13:4] = le
```

ord() and chr() Functions

ord() function returns the ASCII code of the character and chr() function returns character represented by a ASCII number.

Examples:

```
ch = 'R'
print(ord(ch))
```

OUTPUT

```
82
```

```
print(chr(82))
```

OUTPUT

```
R
```

```
print(chr(112))
```

OUTPUT

```
p
```

```
print(ord('p'))
```

OUTPUT

```
112
```

in and not in Operators

in and not in operators can be used with strings to determine whether a string is present in another string. Therefore, the in and not in operator are also known as membership operators.

Examples:

```
str1 = "Welcome to the world of Python
!!!"
str2 = "the"
if str2 in str1:
    print("Found")
else:
    print("Not Found")
```

OUTPUT

```
Found
```

```
str1 = "This is a very good book"
str2 = "best"
if str2 not in str1:
    print("The book is very good but it
may not be the best one.")
else:
    print ("It is the best book.")
```

OUTPUT

```
The book is very good but it may not be
the best one.
```

Comparing Strings

Operator	Description	Example
==	If two strings are equal, it returns True.	>>> "AbC" == "AbC" True

!= or <>	If two strings are not equal, it returns True.	>>> "AbC" != "Abc"
		True
		>>> "abc" <> "ABC"
		True

>	If the first string is greater than the second, it returns True.	>>> "abc" > "Abc" True
<	If the second string is greater than the first, it returns True.	>>> "abC" < "abc" True
>=	If the first string is greater than or equal to the second, it returns True.	>>> "aBC" >= "ABC" True
<=	If the second string is greater than or equal to the first, it returns True.	>>> "ABc" <= "ABc" True

Iterating String

String is a sequence type (sequence of characters). You can iterate through the string using for loop.

Examples:

```
str = "Welcome to Python"
for i in str:
print(i, end=' ')
```

OUTPUT

```
W e l c o m e  t o  P y t h o n
```

```
message = " Welcome to Python "
index = 0
while index < len(message):
    letter = message[index]
    print(letter, end=' ')
    index += 1
```

OUTPUT

```
W e l c o m e  t o  P y t h o n
```

The String Module

The string module consist of a number of useful constants, classes and functions (some of which are deprecated). These functions are used to manipulate strings.

Examples:

```
str = "Welcome to the world of Python"
print("Uppercase - ", str.upper())
print("Lowercase - ", str.lower())
print("Split - ", str.split())
print("Join - ", '-'.join(str.split()))
print("Replace - ", str.replace("Python", "Java"))
print("Count of o - ", str.count('o') )
print("Find of - ", str.find("of"))
```

OUTPUT

```
Uppercase -  WELCOME TO THE WORLD OF PYTHON
Lowercase -  welcome to the world of python
Split -  ['Welcome', 'to', 'the', 'world', 'of', 'Python']
Join -  Welcome-to-the-world-of-Python
Replace -  Welcome to the world of Java
Count of o -  5
Find of -  21
```

Programming Tip: A method is called by appending its name to the variable name using the period as a delimiter.

Working with Constants in String Module

You can use the constants defined in the string module along with the find function to classify characters. For example, if find(lowercase, ch) returns a value except -1, then it means that ch must be a lowercase character. An alternate way to do the same job is to use the in operator or even the comparison operation.

Examples:

```
# First Way
import string
print(string.find(string.
lowercase, 'g') != -1)
```

OUTPUT

```
True
```

```
# Second Way
import string
print('g' in string.
lowercase)
```

OUTPUT

```
True
```

```
# Third Way
import string
ch = 'g'
print('a' <= ch <= 'z')
```

OUTPUT

```
True
```

Lists

List **is a versatile data type available in Python. It is a sequence in which elements are written as a list of comma-separated values (items) between square brackets. The key feature of a list is that it can have elements that belong to different data types. The syntax of defining a list can be given as,**

List_variable = [val1, val2,...]

Examples:

```
>>> list_A = [1,2,3,4,5]
>>> print(list_A)
[1, 2, 3, 4, 5]
```

```
>>> list_B = ['A', 'b', 'C', 'd', 'E']
>>> print(list_B)
['A', 'b', 'C', 'd', 'E']
```

```
>>> list_C = ["Good", "Going"]
>>> print(list_C)
['Good', 'Going']
```

```
>>> list_D = [1, 'a', "bcd"]
>>> print(list_D)
[1, 'a', 'bcd']
```

Access Values in Lists

Similar to strings, lists can also be sliced and concatenated. To access values in lists, square brackets are used to slice along with the index or indices to get value stored at that index. The syntax for the *slice operation* is given as, seq = List[start:stop:step]

Example:

```
num_list = [1,2,3,4,5,6,7,8,9,10]
print("num_list is : ", num_list)
print("First element in the list is ", num_list[0])
print("num_list[2:5] = ", num_list[2:5])
print("num_list[::2] = ", num_list[::2])
print("num_list[1::3] = ", num_list[1::3])
```

OUTPUT

```
num_list is :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
First element in the list is  1
num_list[2:5] =  [3, 4, 5]
num_list[::2] =  [1, 3, 5, 7, 9]
num_list[1::3] =  [2, 5, 8]
```

Updating Values in Lists

Once created, one or more elements of a list can be easily updated by giving the slice on the left-hand side of the assignment operator. You can also append new values in the list and remove existing value(s) from the list using the *append() method* and *del statement* respectively.

Example:

```
num_list = [1,2,3,4,5,6,7,8,9,10]
print("List is : ", num_list)
num_list[5] = 100
print("List after udpation is : ", num_list)
num_list.append(200)
print("List after appending a value is ", num_list)
del num_list[3]
print("List after deleting a value is ", num_list)
```

Programming Tip: append() and insert() methods are list methods. They cannot be called on other values such as strings or integers.

OUTPUT

```
List is :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List after udpation is :  [1, 2, 3, 4, 5, 100, 7, 8, 9, 10]
List after appending a value is  [1, 2, 3, 4, 5, 100, 7, 8, 9, 10, 200]
List after deleting a value is  [1, 2, 3, 5, 100, 7, 8, 9, 10, 200]
```

Nested Lists

Nested list **means a list within another list. We have already said that a list has elements of different data types which can include even a list.**

Example:

```
list1 = [1, 'a', "abc", [2,3,4,5], 8.9]
i=0
while i<(len(list1)):
    print("List1[",i,"] = ",list1[i])
    i+=1
```

OUTPUT

```
List1[0] =  1
List1[1] =  a
List1[2] =  abc
List1[3] =  [2, 3, 4, 5]
List1[4] =  8.9
```

Cloning Lists

If you want to modify a list and also keep a copy of the original list, then you should create a separate copy of the list (not just the reference). This process is called *cloning*. The slice operation is used to clone a list.

Example:

```
list1 = [1,2,3,4,5,6,7,8,9,10]
list2 = list1                  #copies a list using reference
print("List1 = ", list1)
print("List2 = ", list2)     #both lists point to the same list
list3 = list1[2:6]
print("List3 = ", list3)     #list is a clone of list1
```

OUTPUT

```
List1 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List2 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List3 =  [3, 4, 5, 6]
```

Basic List Operations

Method	Description	Syntax	Example	Output
append()	Appends an element to the list. In insert(), if the index is 0, then element is inserted as the first element and if we write, list.insert(len(list), obj), then it inserts obj as the last element in the list. That is, if index= len(list) then insert() method behaves exactly same as append() method.	list. append(obj)	num_list = [6,3,7,0,1,2,4,9] num_list. append(10) print (num_list)	[6,3,7, 0,1,2, 4,9,10]
count()	Counts the number of times an element appears in the list.	list.count (obj)	print(num_list. count(4))	1
index()	Returns the lowest index of obj in the list. Gives a ValueError if obj is not present in the list.	list. index(obj)	>>> num_list = [6,3,7,0,3,7,6,0] >>> print(num_ list.index(7))	2
insert()	Inserts obj at the specified index in the list.	list. insert(index, obj)	>>> num_list = [6,3,7,0,3,7,6,0] >>> num_list. insert(3, 100) >>> print(num_list)	[6,3,7, 100,0, 3,7,6, 0]
pop()	Removes the element at the specified index from the list. Index is an optional parameter. If no index is specified, then removes the last object (or element) from the list.	list. pop([index])	num_list = [6,3,7,0,1,2,4,9] print(num_list. pop()) print(num_list)	9 [6, 3, 7, 0, 1, 2, 4]

List Methods

remove()	Removes or deletes obj from the list. ValueError is generated if obj is not present in the list. If multiple copies of obj exists in the list, then the first value is deleted.	list.
		remove(obj)
			>>> num_list =
			[6,3,7,0,1,2,4,9]
			>>> num_list.
			remove(0)
			>>>
			print(num_list)
				[6,3,7,
				1,2,4,
				9]

reverse()	Reverse the elements in the list.	list.
		reverse()
			>>> num_list =
			[6,3,7,0,1,2,4,9]
			>>> num_list.
			reverse()
			>>>print(num_list)
				[9, 4,
				2, 1, 7,
				3, 6]

sort()	Sorts the elements in the list.	list.sort()	>>> num_list =
			[6,3,7,0,1,2,4,9]
			>>> num_list.
			sort()
			>>>
			print(num_list)
				[9, 4,
				2, 1, 0,
				7, 3,
				6]

extend()	Adds the elements in a list to the end of another list. Using + or += on a list is similar to using extend().	list1.
		extend(list2)
			>>> num_list1 =
			[1,2,3,4,5]
			>>> num_list2 =
			[6,7,8,9,10]
			>>> num_list1.
			extend(num_list2)
			>>>print(num_
			list1)
				[1, 2,
				3, 4, 5,
				6, 7, 8,
				9, 10]

List Comprehensions

Python also supports computed lists called *list comprehensions* having the following syntax.

List = [expression for variable in sequence]

Where, the expression is evaluated once, for every item in the sequence.

List comprehensions help programmers to create lists in a concise way. This is mainly beneficial to make new lists where each element is the obtained by applying some operations to each member of another sequence or iterable. List comprehension is also used to create a subsequence of those elements that satisfy a certain condition.

Example:

```
cubes = []  # an empty list
for i in range(11):
    cubes.append(i**3)
print("Cubes of numbers from 1-10 : ", cubes)
```

OUTPUT

```
Cubes of numbers from 1-10 :  [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Looping in Lists

Python's for and in constructs are extremely useful especially when working with lists. The for var in list statement is an easy way to access each element in a list (or any other sequence). For example, in the following code, the for loop is used to access each item in the list.

```
for i in list:
    print(i)
```

Example:

```
num_list = [1,2,3,4,5,6,7,8,9,10]
sum = 0
for i in num_list:
    sum += i
print("Sum of elements in the list = ", sum)
print("Average of elements in the list = ",
float(sum/float(len(num_list))))
```

OUTPUT

```
Sum of elements in the list =  55
Average of elements in the list =  5.5
```

Using the enumerate() and range() Functions

enumerate() function is used when you want to print both index as well as an item in the list. The function returns an enumerate object which contains the index and value of all the items of the list as a tuple.

The **range()** function is used when you need to print index.

Examples:

```
num_list = [1,2,3,4,5]
for index, i in enumerate(num_list):
    print(i, " is at index : ", index)
```

OUTPUT

```
1  is a t index :  0
2  is a t index :  1
3  is a t index :  2
4  is a t index :  3
5  is a t index :  4
```

```
num_list = [1,2,3,4,5]
for i in range(len(num_list)):
    print("index : ", i)
```

OUTPUT

```
index :  0
index :  1
index :  2
index :  3
index :  4
```

Using an Iterator

You can create an iterator using the built-in iter() function. The iterator is used to loop over the elements of the list. For this, the iterator fetches the value and then automatically points to the next element in the list when it is used with the next() method.

Example:

```
num_list = [1,2,3,4,5]
it = iter(num_list)
for i in range(len(num_list)):
    print("Element at index ", i, " is : ", next(it))
```

OUTPUT

```
Element at index  0  is :  1
Element at index  1  is :  2
Element at index  2  is :  3
Element at index  3  is :  4
Element at index  4  is :  5
```

filter() Function

The filter() function constructs a list from those elements of the list for which a function returns True. The syntax of the filter() function is given as, filter(function, sequence)

As per the syntax, the filter() function returns a sequence that contains items from the sequence for which the function is True. If sequence is a string, Unicode, or a tuple, then the result will be of the same type; otherwise, it is always a list.

Example:

```
def check(x):
    if (x % 2 == 0 or x % 4 == 0):
        return 1
# call check() for every value between 2 to 21
evens = list(filter(check, range(2, 22)))
print(evens)
```

OUTPUT

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Programming Tip: Do not add or remove elements from the list during iteration.

map() Function

The map() function applies a particular function to every element of a list. Its syntax is same as the filter function

```
map(function, sequence)
```

After applying the specified function on the sequence, the map() function returns the modified list. The map() function calls function(item) for each item in the sequence and returns a list of the return values.

Example: Program that adds 2 to every value in the list

```
def add_2(x):
    x += 2
    return x
num_list = [1,2,3,4,5,6,7]
print("Original List is : ", num_list)
new_list = list(map(add_2, num_list))
print("Modified List is : ", new_list)
```

OUTPUT

```
Original List is :  [1, 2, 3, 4, 5, 6, 7]
Modified List is :  [3, 4, 5, 6, 7, 8, 9]
```

reduce() Function

The reduce() function with syntax as given below returns a single value generated by calling the function on the first two items of the sequence, then on the result and the next item, and so on.

```
reduce(function, sequence)
```

Example: Program to calculate the sum of values in a list using the reduce() function

```
import functools #functools is a module that contains the function reduce()
def add(x,y):
    return x+y
num_list = [1,2,3,4,5]
print("Sum of values in list = ")
print(functools.reduce(add, num_list))
```

OUTPUT

```
Sum of values in list =  15
```

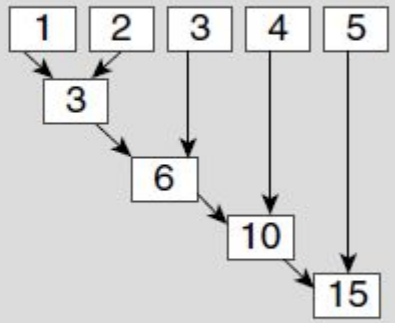


Figure 8.4 reduce() function

Tuple

Like lists, tuple is another data structure supported by Python. It is very similar to lists but differs in two things.

- First, a tuple is a sequence of *immutable* objects. This means that while you can change the value of one or more items in a list, you cannot change the values in a tuple.
- Second, tuples use parentheses to define its elements whereas lists use square brackets.

Creating Tuple

Creating a tuple is very simple and almost similar to creating a list. For creating a tuple, generally you need to just put the different comma-separated values within a parentheses as shown below.

Tup1 = (val 1, val 2,...)

where val (or values) can be an integer, a floating number, a character, or a string.

Utility of Tuples

In real-world applications, tuples are extremely useful for representing records or structures as we call in other programming languages. These structures store related information about a subject together.

The information belongs to different data types.

For example, a tuple that stores information about a student can have elements like roll_no, name, course, total marks, avg, etc. Some built-in functions return a tuple. For example, the divmod() function returns two values—quotient as well as the remainder after performing the divide operation.

Examples:

```
quo, rem = divmod(100,3)
print("Quotient = ",quo)
print("Remainder = ", rem)
```

OUTPUT

```
Quotient =  33
Remainder =  1
```

```
Tup1 = (1,2,3,4,5)
Tup2 = (6,7,8,9,10)
Tup3 = Tup1 + Tup2
print(Tup3)
```

OUTPUT

```
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Accessing Values in a Tuple

Like other sequences (strings and lists) covered so far, indices in a tuple also starts at 0. You can even perform operations like slice, concatenate, etc. on a tuple. For example, to access values in tuple, slice operation is used along with the index or indices to obtain value stored at that index

Example:

```
Tup1 = (1,2,3,4,5,6,7,8,9,10)
print("Tup[3:6] = ", Tup1[3:6])
print("Tup[:8] = ", Tup1[:4])
print("Tup[4:] = ", Tup1[4:])
print("Tup[:] = ", Tup1[:])
```

OUTPUT

```
Tup[3:6] =  (4, 5, 6)
Tup[:8] =  (1, 2, 3, 4)
Tup[4:] =  (5, 6, 7, 8, 9, 10)
Tup[:] =  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Deleting Elements in Tuple

Since tuple is an immutable data structure, you cannot delete value(s) from it. Of course, you can create a new tuple that has all elements in your tuple except the ones you don't want (those you wanted to be deleted).

Examples:

```
Tup1 = (1,2,3,4,5)
del Tup1[3]
print(Tup1)
```

OUTPUT

```
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 2, in <module>
    del Tup1[3]
TypeError: 'tuple' object doesn't support item deletion
```

```
Tup1 = (1,2,3,4,5)
del Tup1
print(Tup1)
```

OUTPUT

```
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 3, in <module>
    print Tup1
NameError: name 'Tup1' is not defined
```

Basic Tuple Operations

Operation	Expression	Output
Length	`len((1,2,3,4,5,6))`	`6`
Concatenation	`(1,2,3) + (4,5,6)`	`(1, 2, 3, 4, 5, 6)`
Repetition	`('Good..')*3`	`'Good..Good..Good..'`
Membership	`5 in (1,2,3,4,5,6,7,8,9)`	`True`
Iteration	`for i in (1,2,3,4,5,6,7,8,9,10):` `  print(i,end=' ')`	`1,2,3,4,5,6,7,8,9,10`

Comparison (Use >, <, ==)	`Tup1 = (1,2,3,4,5)`
	`Tup2 = (1,2,3,4,5)`
	`print(Tup1>Tup2)`
		`False`

Maximum	`max(1,0,3,8,2,9)`	`9`
Minimum	`min(1,0,3,8,2,9)`	`0`
Convert to tuple (converts a sequence into a tuple)	`tuple("Hello")` `tuple([1,2,3,4,5])`	`('H', 'e', 'l', 'l', 'o')` `(1, 2, 3, 4, 5)`

Tuple Assignment

Tuple assignment is a very powerful feature in Python. It allows a tuple of variables on the left side of the assignment operator to be assigned values from a tuple given on the right side of the assignment operator.

Each value is assigned to its respective variable. In case, an expression is specified on the right side of the assignment operator, first that expression is evaluated and then assignment is done.

Example:

```
# an unnamed tuple of values assigned to values of another unnamed tuple
(val1, val2, val3) = (1,2,3)
print(val1, val2, val3)
Tup1 = (100, 200, 300)
(val1, val2, val3) = Tup1    # tuple assigned to another tuple
print(val1, val2, val3)
# expressions are evaluated before assignment
(val1, val2, val3)= (2+4, 5/3 + 4, 9%6)
print(val1, val2, val3)
```

OUTPUT

```
1 2 3
100 200 300
6 5.666667 3
```

Tuples for Returning Multiple Values and Nested Tuples

Examples:

```
def max_min(vals):
    x = max(vals)
    y = min(vals)
    return (x,y)
vals = (99, 98, 90, 97, 89, 86, 93, 82)
(max_marks, min_marks) = max_min(vals)
print("Highest Marks = ", max_marks)
print("Lowest Marks = ", min_marks)
```

Programming Tip: You can't delete elements from a tuple. Methods like remove() or pop() do not work with a tuple.

OUTPUT

```
Highest Marks =  99
Lowest Marks =  82
```

```
Toppers = (("Arav", "BSc",92.0), ("Chaitanya", "BCA", 99.0),
("Dhruvika", "Btech", 97))
for i in Toppers:
    print(i)
```

OUTPUT

```
('Arav', 'BSc', 92.0)
('Chaitanya', 'BCA', 99.0)
('Dhruvika', 'Btech', 97)
```

Checking the Index: index() method

The index of an element in the tuple can be obtained by using the index() method. If the element being searched is not present in the list, then error is generated. The syntax of index() is given as, list.index(obj) where, obj is the object to be found out.

Examples:

```
Tup = (1,2,3,4,5,6,7,8)
print(Tup.index(4))
```

OUTPUT

```
3
```

```
students = ("Bhavya", "Era", "Falguni","Huma")
index = students.index("Falguni")
print("Falguni is present at location : ", index)
index = students.index("Isha")
print("Isha is present at location : ", index)
```

OUTPUT

```
Falguni is present at location :  2
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 4, in <module>
    index = students.index("Isha")
ValueError: tuple.index(x): x not in tuple
```

count()Method and List Comprehension and Tuples

Examples:

```
tup = "abcdxxxabcdxxxabcdxxx"
print("x appears ", tup.count('x'), " times in ", tup)

OUTPUT

x appears  9  times in  abcdxxxabcdxxxabcdxxx
```

```
def double(T):
    return ([i*2 for i in T])
Tup = 1,2,3,4,5
print("Original Tuple : ", Tup)
print("Double Values: ",double(Tup))

OUTPUT

Original Tuple :  (1, 2, 3, 4, 5)
Double Values:  [2, 4, 6, 8, 10]
```

Variable-length Argument Tuples

Many built-in functions like max(), min(), sum(), etc. use variable-length arguments since these functions themselves do not know how many arguments will be passed to them. It allows a function to accept a variable (different) number of arguments. This is especially useful in defining functions that are applicable to a large variety of arguments. For example, if you have a function that displays all the parameters passed to it, then even the function does not know how many values it will be passed. In such cases, we use a variable-length argument that begins with a '*' symbol. Any argument that starts with a '*' symbol is known as gather and specifies a variable-length argument.

Example:

```
def display(*args):
    print(args)
Tup = (1,2,3,4,5,6)
display(Tup)

OUTPUT
((1, 2, 3, 4, 5, 6),)
```

The zip() Function

The zip() is a built-in function that takes two or more sequences and "zips" them into a list of tuples. The tuple thus, formed has one element from each sequence.

Example: Program to show the use of zip() function

```
Tup = (1,2,3,4,5)
List1 = ['a','b','c','d','e']
print(list((zip(Tup, List1))))
```

OUTPUT

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Advantages of Tuple over List

- Tuples are used to store values of different data types. Lists can however, store data of similar data types.
- Since tuples are immutable, iterating through tuples is faster than iterating over a list. This means that a tuple performs better than a list.
- Tuples can be used as key for a dictionary but lists cannot be used as keys.
- Tuples are best suited for storing data that is write-protected.
- Tuples can be used in place of lists where the number of values is known and small.
- If you are passing a tuple as an argument to a function, then the potential for unexpected behavior due to aliasing gets reduced.
- Multiple values from a function can be returned using a tuple.
- Tuples are used to format strings.

Sets

Sets **is another data structure supported by Python. Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically, a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.**

A set is created by placing all the elements inside curly brackets {}, separated by comma or by using the built-in function set(). The syntax of creating a set can be given as,

```
set_variable = {val1, val2, ...}
```

Example: To create a set, you can write,

```
>>> s = {1,2.0,"abc"}
>>> print( s)
set([1, 2.0,'abc'])
```

Set Operations

Operation	Description	Code	Output

`s.update(t)`	Adds elements of set t in the set s provided that all duplicates are avoided	`s = set([1,2,3,4,5])`
		`t = set([6,7,8])`
		`s.update(t)`
		`print(s)`
			`(1,2,3,4,5,6,7,8)`

`s.add(x)`	Adds element x to the set s provided that all duplicates are avoided	`s = set([1,2,3,4,5])`
		`s.add(6)`
		`print(s)`
			`(1,2,3,4,5,6)`

`s.remove(x)`	Removes element x from set s. Returns KeyError if x is not present	`s = set([1,2,3,4,5])`
		`s.remove(3)`
		`print(s)`
			`(1,2,4,5)`

`s.discard(x)`	Same as `remove()` but does not give an error if x is not present in the set	`s = set([1,2,3,4,5])`
		`s.discard(3)`
		`print(s)`
			`(1,2,4,5)`

`s.pop()`	Removes and returns any arbitrary element from s. KeyError is raised if s is empty	`s = set([1,2,3,4,5])`
		`s.pop()`
		`print(s)`
			`(2,3,4,5)`

`s.clear()`	Removes all elements from the set	`s = set([1,2,3,4,5])`
		`s.clear()`
		`print(s)`
			`set()`

Set Operations

`len(s)`	Returns the length of set	`s = set([1,2,3,4,5])` `print(len(s))`	`5`
`x in s`	Returns True is x is present in set s and False otherwise	`s = set([1,2,3,4,5])` `print(3 in s)`	`True`
`x not in s`	Returns True is x is not present in set s and False otherwise	`s = set([1,2,3,4,5])` `print(6 not in s)`	`True`

`s.issubset(t)` or `s<=t`	Returns True if every element in set s is present in set t and False otherwise	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`print(s<=t)`
			`True`

`s.issuperset(t)` or `s>=t`	Returns True if every element in t is present in set s and False otherwise	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`print(s.issuperset(t))`
			`False`

`s.union(t)` or `s|t`	Returns a set s that has elements from both sets s and t	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`print(s|t)`
			`(1,2,3,4, 5,6,7,8, 9,10)`

`s.intersection(t)` or `s&t`	Returns a new set that has elements which are common to both the sets s and t	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`z = s&t`
		`print(z)`
			`(1,2,3, 4,5)`

`s.intersection_ update(t)`	Returns a set that has elements which are common to both the sets s and t	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`s.intersection_update(t)`
		`print(s)`
			`(1,2,10)`

Set Operations

`s.difference(t) or s-t`	Returns a new set that has elements in set s but not in t	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`z = s-t`
		`print(z)`
			(12)

`s.difference_ update(t)`	Removes all elements of another set from this set	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`s.difference_update(t)`
		`print(s)`
			(12)

`s.symmetric_ difference(t) or s^t`	Returns a new set with elements either in s or in t but not both	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`z = s^t`
		`print(z)`
			(3,4,5,6, 7,8,9,12)

`s.copy()`	Returns a copy of set s	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9, 10])`
		`print(s.copy())`
			(1,2,12,10)

`s.isisjoint(t)`	Returns True if two sets have a null intersection	`s = set([1,2, 3])`
		`t = set([4,5,6])`
		`print(s.isdisjoint(t))`
			True

`all(s)`	Returns True if all elements in the set are True and False otherwise	`s = set([0,1,2,3,4])` `print(all(s))`	False
`any(s)`	Returns True if any of the elements in the set is True. Returns False if the set is empty	`s = set([0,1,2,3,4])` `print(any(s))`	True

Set Operations

`enumerate(s)`	Returns an enumerate object which contains index as well as value of all the items of set as a pair	`s = set(['a','b','c','d'])`
		`for i in enumerate(s):`
		`  print(i,end= ' ')`
			`(0, 'a') (1, 'c') (2, 'b') (3, 'd')`

`max(s)`	Returns the maximum value in a set	`s = set([0,1,2,3,4,5])` `print(max(s))`	`5`
`min(s)`	Returns the minimum value in a set	`s = set([0,1,2,3,4,5])` `print(min(s))`	`0`
`sum(s)`	Returns the sum of elements in the set	`s = set([0,1,2,3,4,5])` `print(sum(s))`	`15`
`sorted(s)`	Return a new sorted list from elements in the set. It does not sorts the set as sets are immutable.	`s = set([5,4,3,2,1,0])` `print(sorted(s))`	`[0,1,2,3, 4,5]`

`s == t and s != t`	s == t returns True if the two set are equivalent and False otherwise. s!=t returns True if both sets are not equivalent and False otherwise	`s = set(['a','b','c'])`
		`t = set("abc")`
		`z = set(tuple('abc'))`
		`print(s == t)`
		`print(s!=z)`
			`True`
			`False`

Dictionaries

Dictionary **is a data structure in which we store values as a pair of key and value. Each key is separated from its value by a colon (:), and consecutive items are separated by commas. The entire items in a dictionary are enclosed in curly brackets({}). The syntax for defining a dictionary is**

dictionary_name = {key_1: value_1, key_2: value_2, key_3: value_3}

If there are many keys and values in dictionaries, then we can also write just one key-value pair on a line to make the code easier to read and understand. This is shown below.

dictionary_name = {key_1: value_1, key_2: value_2, key_3: value_3, ...}

Example:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print(Dict)

OUTPUT
{'Roll_No': '16/001', 'Name': 'Arav', 'Course': 'BTech'}
```

Accessing Values

Example:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
```

OUTPUT

```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
```

Adding and Modifying an Item in a Dictionary

Example:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
Dict['Marks'] = 95      # new entry
print("Dict[MARKS] = ", Dict['Marks'])
```

Programming Tip: Trying to index a key that isn't part of the dictionary returns a KeyError.

OUTPUT

```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
Dict[MARKS] =  95
```

Modifying an Entry

Example:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
Dict['Marks'] = 95      # new entry
print("Dict[MARKS] = ", Dict['Marks'])

Dict['Course'] = 'BCA'
print("Dict[COURSE] = ", Dict['Course']) #entry updated
```

OUTPUT

```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
Dict[MARKS] =  95
Dict[COURSE] =  BCA
```

Deleting Items

You can delete one or more items using the del keyword. To delete or remove all the items in just one statement, use the clear() function. Finally, to remove an entire dictionary from the memory, we can gain use the del statement as del Dict_name. The syntax to use the del statement can be given as,

del dictionary_variable[key]

Example:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Name is : ", Dict.pop('Name'))    # returns Name)
print("Dictionary after popping Name is : ", Dict)
print("Marks is :", Dict.pop('Marks', -1))   # returns default value
print("Dictionary after popping Marks is : ", Dict)
print("Randomly popping any item : ",Dict.popitem())
print("Dictionary after random popping is : ", Dict)
print("Aggregate is :", Dict.pop('Aggr'))    # generates error
print("Dictionary after popping Aggregate is : ", Dict)
```

OUTPUT

```
Name is :  Arav
Dictionary after popping Name is :  {'Course': 'BTech', 'Roll_No': '16/001'}
Marks is : -1
Dictionary after popping Marks is :  {'Course': 'BTech', 'Roll_No': '16/001'}
Randomly popping any item :  ('Course', 'BTech')
Dictionary after random popping is :  {'Roll_No': '16/001'}
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 8, in <module>
    print("Aggregate is :", Dict.pop('Aggr'))
KeyError: 'Aggr'
```

Sorting Items and Looping over Items in a Dictinonary

Examples:

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print(sorted(Dict.keys()))
```

OUTPUT

```
['Course', 'Name', 'Roll_No']
```

```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("KEYS : ", end = ' ')
for key in Dict:
    print(key, end = ' ')   # accessing only keys
print("\nVALUES : ", end = ' ')
for val in Dict.values():
    print(val, end = ' ')   # accessing only values
print("\nDICTIONARY : ", end = ' ')
for key, val in Dict.items():
    print(key, val, "\t", end = ' ')  # accessing keys and values
```

OUTPUT

```
KEYS :  Roll_No Course Name
VALUES :  16/001 BTech Arav
DICTIONARY :  Roll_No 16/001  Course BTech    Name Arav
```

Nested Dictionaries

Example:

```
Students = {'Shiv' : {'CS':90, 'DS':89, 'CSA':92},
            'Sadhvi' : {'CS':91, 'DS':87, 'CSA':94},
            'Krish' : {'CS':93, 'DS':92, 'CSA':88}}
for key, val in Students.items():
    print(key, val)
```

OUTPUT

```
Sadhvi {'CS': 91, 'CSA': 94, 'DS': 87}
Krish {'CS': 93, 'CSA': 88, 'DS': 92}
Shiv {'CS': 90, 'CSA': 92, 'DS': 89}
```

Built-in Dictionary Functions and Methods

Operation	Description	Example	Output
`len(Dict)`	Returns the length of dictionary. That is, number of items (key-value pairs)	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}` `print(len(Dict1))`	`3`
`str(Dict)`	Returns a string representation of the dictionary	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}` `print(str(Dict1))`	`{'Name': 'Arav', 'Roll_No': '16/001', 'Course': 'BTech'}`

`Dict.clear()`	Deletes all entries in the dictionary	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}`
		`Dict1.clear()`
		`print(Dict1)`
			`{}`

`Dict.copy()`	Returns a shallow copy of the dictionary, i.e., the dictionary returned will not have a duplicate copy of `Dict` but will have the same reference	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}`
		`Dict2 = Dict1.copy()`
		`print("Dict2 : ", Dict2)`
		`Dict2['Name'] = 'Saesha'`
		`print("Dict1 after modification : ", Dict1)`
			`Dict2 : {'Course': 'BTech', 'Name': 'Arav', 'Roll_No': '16/001'}`
			`Dict1 after modification: {'Course': 'BTech',`

Built-in Dictionary Functions and Methods

		`print("Dict2 after modification : ",Dict2)`	`'Name': 'Arav', 'Roll_No': '16/001'} Dict2 after modification: {'Course': 'BTech', 'Name': 'Saesha', 'Roll_No': '16/001'}`
`Dict. fromkeys(seq[,val])`	Create a new dictionary with keys from seq and values set to val. If no val is specified then, None is assigned as default value	`Subjects = ['CSA', 'C++', 'DS', 'OS'] Marks = dict. fromkeys(Subjects,-1) print(Marks)`	`{'OS': -1, 'DS': -1, 'CSA': -1, 'C++': -1}`
`Dict.get(key)`	Returns the value for the key passed as argument. If the key is not present in dictionary, it will return the default value. If no default value is specified then it will return None	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print( Dict1.get('Name'))`	`Arav`
`Dict.has_key(key)`	Returns True if the key is present in the dictionary and False otherwise	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print('Marks' in Dict1)`	`False`
`Dict.items()`	Returns a list of tuples (key-value pair)	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print(Dict1.items())`	`[('Course', 'BTech'), ('Name', 'Arav'), ('Roll_No', '16/001')]`
`Dict.keys()`	Returns a list of keys in the dictionary	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print(Dict1.keys())`	`['Course', 'Name', 'Roll_No']`
`Dict. setdefault(key, value)`	Sets a default value for a key that is not present in the dictionary	`Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}`	`Arav has got marks = 0`

Built-in Dictionary Functions and Methods

		Dict1. setdefault('Marks',0) print(Dict1['Name'], "has got marks = ", Dict1. get('Marks'))	
Dict1.update(Dict2)	Adds the key-value pairs of Dict2 to the key-value pairs of Dict1	Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} Dict2 = {'Marks' : 90, 'Grade' : 'O'} Dict1.update(Dict2) print(Dict1)	{'Grade':'O', 'Course': 'BTech', 'Name': 'Arav', 'Roll_No': '16/001', 'Marks': 90}
Dict.values()	Returns a list of values in dictionary	Dict1 = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print(Dict1.values())	['BTech', 'Arav', '16/001']
Dict.iteritems()	Used to iterate through items in the dictionary	Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} for i,j in Dict. iteritems(): print(i, j)	Course BTech Name Arav Roll_No 16/001
in and not in	Checks whether a given key is present in dictionary or not	Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'} print('Name' in Dict) print('Marks' in Dict)	True False

Difference between a List and a Dictionary

First, a list is an ordered set of items. But, a dictionary is a data structure that is used for matching one item (key) with another (value).

- Second, in lists, you can use indexing to access a particular item. But, these indexes should be a number. In dictionaries, you can use any type (immutable) of value as an index. For example, when we write Dict['Name'], Name acts as an index but it is not a number but a string.
- Third, lists are used to look up a value whereas a dictionary is used to take one value and look up another value. For this reason, dictionary is also known as a ***lookup table***.

Fourth, the key-value pair may not be displayed in the order in which it was specified while defining the dictionary. This is because Python uses complex algorithms (called hashing) to provide fast access to the items stored in the dictionary. This also makes dictionary preferable to use over a list of tuples.

String Formatting with Dictionaries

Python also allows you to use string formatting feature with dictionaries. So you can use %s, %d, %f, etc. to represent string, integer, floating point number, or any other data.

Example: Program that uses string formatting feature to print the key-value pairs stored in the dictionary

```
Dict = {'Sneha' : 'BTech', 'Mayank' : 'BCA'}
for key, val in Dict.items():
    print("%s is studying %s" % (key,val))
```

OUTPUT

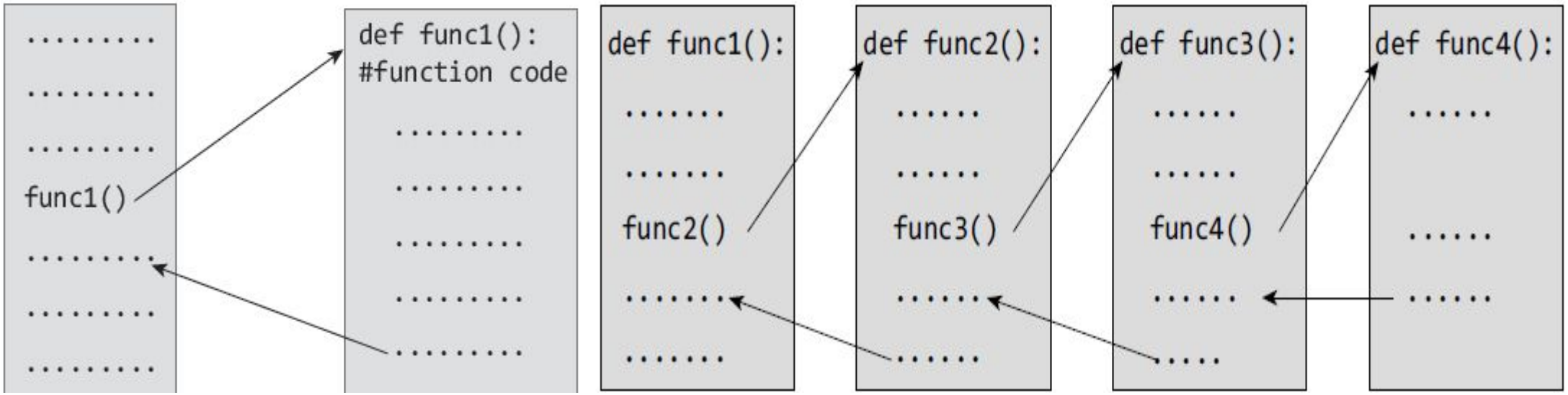
```
Sneha is studying BTech
Mayank is studying BCA
```

When to use which Data Structure?

- Use lists to store a collection of data that does not need random access.
- Use lists if the data has to be modified frequently.
- Use a set if you want to ensure that every element in the data structure must be unique.
- Use tuples when you want that your data should not be altered.

Functions

Python enables its programmers to break up a program into segments commonly known as functions, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task.



Need for Functions

Each function to be written and tested separately.

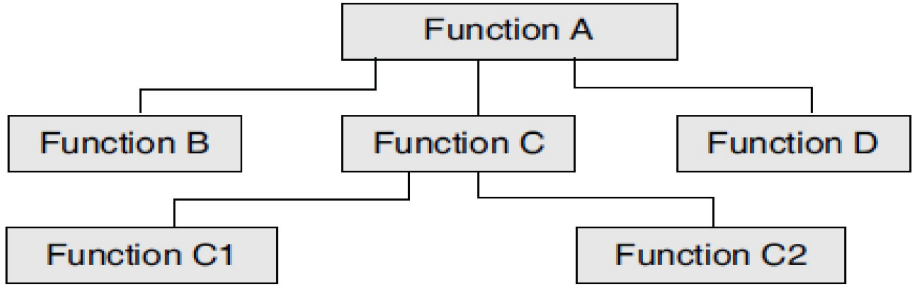
- Understanding, coding and testing multiple separate functions is far easier.

Without the use of any function, then there will be countless lines in the code and maintaining it will be a big mess.

- Programmers use functions without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he has to write.

Different programmers working on that project can divide the workload by writing different functions.

- Like Python libraries, programmers can also make their functions and use them from different point in the main program or any other program that needs its functionalities.



Function Declaration and Definition

- A function, f that uses another function g, is known as the ***calling function*** and g is known as the ***called function***.
- The inputs that the function takes are known as ***arguments/parameters***.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- ***Function declaration*** is a declaration statement that identifies a function with its name, a list of arguments that it accepts and the type of data it returns.
- ***Function definition*** consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Function Definition

Function blocks starts with the keyword def.

- **The keyword is followed by the function name and parentheses (( )).**
- **After the parentheses a colon (:) is placed.**
- **Parameters or arguments that the function accept are placed within parentheses.**
- **The first statement of a function can be an optional statement - the *docstring* describe what the function does.**
- **The code block within the function is properly indented to form the block code.**
- **A function may have a return[expression] statement. That is, the return statement is optional.**
- **You can assign the function name to a variable. Doing this will allow you to call same function using the name of that variable.**

Example:

```
def diff(x,y):          # function to subtract two numbers
        return x-y
a = 20
b = 10
operation = diff        # function name assigned to a variable
print(operation(a,b))   # function called using variable name
```

OUTPUT

```
10
```

Function Call

The function call statement invokes the function. When a function is invoked the program control jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Function Parameters

A function can take parameters which are nothing but some values that are passed to it so that the function can manipulate them to produce the desired result. These parameters are normal variables with a small difference that the values of these variables are defined (initialized) when we call the function and are then passed to the function.

Function name and the number and type of arguments in the function call must be same as that given in the function definition.

If the data type of the argument passed does not matches with that expected in function then an error is generated.

Example:

```
def function_name(variable1, variable2,..)
        documentation string
        statement block
        return [expression]
```

Function Header

Function Body

Examples

```
def func():
        for i in range(4):
                print("Hello World")
func()      #function call
```

OUTPUT

```
Hello World
Hello World
Hello World
Hello World
```

```
def func(i):
        print("Hello World", i)
func(5+2*3)
```

OUTPUT

```
Hello World 11
```

```
def total(a,b):    # function accepting parameters
   result = a+b
   print("Sum of ", a, " and ", b, " = ", result)

a = int(input("Enter the first number : "))
b = int(input("Enter the second number : "))
total(a,b) #function call with two arguments
```

OUTPUT

```
Enter the first number : 10
Enter the second number : 20
Sum of 10 and 20 = 30
```

```
def func(i):                # function definition header accepts a variable with name i
print("Hello World", i)
j = 10
func(j)                     # Function is called using variable j
```

OUTPUT

```
Hello World 10
```

Local and Global Variables

A variable which is defined within a function is ***local*** to that function. A local variable can be accessed from the point of its definition until the end of the function in which it is defined. It exists as long as the function is executing. Function parameters behave like local variables in the function. Moreover, whenever we use the assignment operator (=) inside a function, a new local variable is created.

Global variables are those variables which are defined in the main body of the program file. They are visible throughout the program file. As a good programming habit, you must try to avoid the use of global variables because they may get altered by mistake and then result in erroneous output.

Local and Global Variables

Example:

```
num1 = 10    # global variable
print("Global variable num1 = ", num1)
def func(num2):           # num2 is function parameter
        print("In Function - Local Variable num2 = ",num2)
        num3 = 30         #num3 is a local variable
        print("In Function - Local Variable num3 = ",num3)
func(20)       #20 is passed as an argument to the function
print("num1 again = ", num1)        #global variable is being accessed
#Error- local variable can't be used outside the function in which it is defined
print("num3 outside function = ", num3)
```

OUTPUT

```
Global variable num1 = 10
In Function - Local Variable num2 = 20
In Function - Local Variable num3 = 30
num1 again =  10
num3 outside function =
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 12, in <module>
    print("num3 outside function = ", num3)
NameError: name 'num3' is not defined
```

Programming Tip: Variables can only be used after the point of their declaration

Using the Global Statement

To define a variable defined inside a function as global, you must use the global statement. This declares the local or the inner variable of the function to have module scope.

Example:

```
var = "Good"
def show():
        global var1
        var1 = "Morning"
        print("In Function var is - ", var)
show()
print("Outside function, var1 is - ", var1)      #accessible as it is global
variable
print("var is - ", var)
```

OUTPUT

```
In Function var is -  Good
Outside function, var1 is -  Morning
var is -  Good
```

Programming Tip: All variables have the scope of the block.

Resolution of names

Scope **defines the visibility of a name within a block. If a local variable is defined in a block, its scope is that particular block. If it is defined in a function, then its scope is all blocks within that function.**

When a variable name is used in a code block, it is resolved using the nearest enclosing scope. If no variable of that name is found, then a NameError is raised. In the code given below, str is a global string because it has been defined before calling the function.

Example:

```
def func():
        print(str)
str = "Hello World !!!"
func()
```

OUTPUT

```
Hello World !!!
```

The Return Statement

The syntax of return statement is,

return [expression]

The expression is written in brackets because it is optional. If the expression is present, it is evaluated and the resultant value is returned to the calling function. However, if no expression is specified then the function will return None.

The return statement is used for two things.

- **Return a value to the caller**
- **To end and exit a function and go back to its caller**

Example:

```
def cube(x):
   return (x*x*x)
num = 10
result = cube(num)
print('Cube of ', num, ' = ', result)
```

OUTPUT

```
Cube of 10 = 1000
```

Required Arguments

In the *required arguments*, the arguments are passed to a function in correct positional order. Also, the number of arguments in the function call should exactly match with the number of arguments specified in the function definition

Examples:

```
def display():
   print "Hello"
display("Hi")
```

OUTPUT

```
TypeError: display() takes
no arguments (1 given)
```

```
def display(str):
   print str
display()
```

OUTPUT

```
TypeError: display() takes
exactly 1 argument (0
given)
```

```
def display(str):
   print str
str ="Hello"
display(str)
```

OUTPUT

```
Hello
```

Keyword Arguments

When we call a function with some values, the values are assigned to the arguments based on their position. Python also allow functions to be called using keyword arguments in which the order (or position) of the arguments can be changed. The values are not assigned to arguments according to their position but based on their name (or keyword).

Keyword arguments are beneficial in two cases.

- **First, if you skip arguments.**
- **Second, if in the function call you change the order of parameters.**

Example:

```
def display(str, int_x, float_y):
   print("The string is : ",str)
   print("The integer value is : ", int_x)
   print("The floating point value is : ", float_y)
display(float_y = 56789.045, str = "Hello", int_x = 1234)
```

OUTPUT

```
The string is:  Hello
The integer value is:  1234
The floating point value is:  56789.045
```

Variable-length Arguments

In some situations, it is not known in advance how many arguments will be passed to a function. In such cases, Python allows programmers to make function calls with arbitrary (or any) number of arguments.

When we use arbitrary arguments or variable length arguments, then the function definition use an asterisk (*) before the parameter name. The syntax for a function using variable arguments can be given as,

```
def functionname([arg1, arg2,.... ] *var_args_tuple ):
    function statements
    return [expression]
```

Example:

```
def func(name, *fav_subjects):
    print("\n", name, " likes to read ")
    for subject in fav_subjects:
        print(subject)
func("Goransh", "Mathematics", "Android Programming")
func("Richa", "C", "Data Structures", "Design and Analysis of Algorithms")
func("Krish")
```

OUTPUT

```
Goransh  likes to read  Mathematics Android Programming
Richa likes to read  C Data Structures Design and Analysis of Algorithms
Krish  likes to read
```

Default Arguments

Python allows users to specify function arguments that can have default values. This means that a function can be called with fewer arguments than it is defined to have. That is, if the function accepts three parameters, but function call provides only two arguments, then the third parameter will be assigned the default (already specified) value.

The default value to an argument is provided by using the assignment operator (=). Users can specify a default value for one or more arguments.

Example:

```
def display(name, course = "BTech"):
    print("Name : " + name)
    print("Course : ", course)
display(course = "BCA", name = "Arav") # Keyword Arguments
display(name = "Reyansh")              # Default Argument for course
```

OUTPUT

```
Name : Arav
Course :  BCA
Name : Reyansh
Course :  BTech
```

Lambda Functions Or Anonymous Functions

Lambda or anonymous **functions are so called because they are not declared as other functions using the def keyword. Rather, they are created using the lambda keyword. Lambda functions are throw-away functions, i.e. they are just needed where they have been created and can be used anywhere a function is required. The lambda feature was added to Python due to the demand from LISP programmers.**

Lambda functions contain only a single line. Its syntax can be given as,

```
lambda arguments: expression
```

Example:

```
sum = lambda x, y: x + y
print("Sum = ", sum(3, 5))
```

OUTPUT

```
Sum = 8
```

Documentation Strings

Docstrings (documentation strings) serve the same purpose as that of comments, as they are designed to explain code.

However, they are more specific and have a proper syntax.

```
def functionname(parameters):
   "function_docstring"
   function statements
   return [expression]
```

Example:

```
def func():
    """The program just prints a message.
    It will display Hello World !!!   """
    print("Hello World !!!")
print(func.__doc__)
```

OUTPUT

```
The program just prints a message.
    It will display Hello World !!!
```

Recursive Functions

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases, which are as follows:

- ***base case***, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- ***recursive case,*** in which first the problem at hand is divided into simpler sub parts.

Recursion utilized divide and conquer technique of problem solving.

Example:

```
def fact(n):
      if(n==1 or n==0):
            return 1
      else:
            return n*fact(n-1)
n = int(input("Enter the value of n : "))
print("The factorial of",n,"is",fact(n))
```

OUTPUT

```
Enter the value of n : 6
The factorial of 6 is 720
```