Functions

Program components in Python include **functions, classes, modules** and **packages.**

- **Function** is a self-contained program segment that carries out some specific task. Once defined it can be accessed many times from different parts of the program without rewriting the entire code.
  Python built-in functions are pre-defined functions
  User-defined functions are created by users.
  Anonymous functions are functions having no name
- **Classes** are blueprint of objects having common attributes and behavior.
- A **module** is a file containing definition of certain functions and classes along with variables and other constants.
- Many modules are grouped together to form a **package**.

Python interpreter comes loaded with many built-in functions which are readily available to be used in programs. Some of them are discussed below.

abs(num)

It returns the absolute value of a number. If given number is complex, it returns the magnitude.

Example
```
print(abs(-2.5))
print(abs(2+0j))
```
Output
```
2.5
2.0
```

any(iterable)

This function returns True if any of the item in an iterable is True and False otherwise. If iterable is empty, function returns False.

Example
```
lis = [1,0,"hello"]
print(any(lis))
```
Output
```
True
```

NOTE : ''An **iterable** is **any Python object capable of returning its members one at a time**, permitting it to be iterated over in a for-loop. Familiar examples of iterables include lists, tuples, and strings - any such sequence can be iterated over in a for-loop.''

all(iterable)

It returns True, only if all items in iterable are True.

Example
```
lis = [1,0,"hello"]
print(all(lis))
```
Output
```
False
```

bool(object)

The Boolean value of given object is returned by this function. It returns False only when,

- object is empty
- object is False
- object is zero
- object is None

Example

```
a=0
print(bool(a))
b=[0]
print(bool(b))
lis=[1,0,”hello”]
print(bool(lis))
```

Output

```
False
True
True
```

complex(real, imaginary)

Returns a complex number with the specified real and imaginary part.

Example

```
print(complex(3,4))
```

Output

```
(3+4j)
```

format(value[,format_spec])

The format() method returns a formatted representation of the given value controlled by the format specifier.

The format() function takes two parameters:

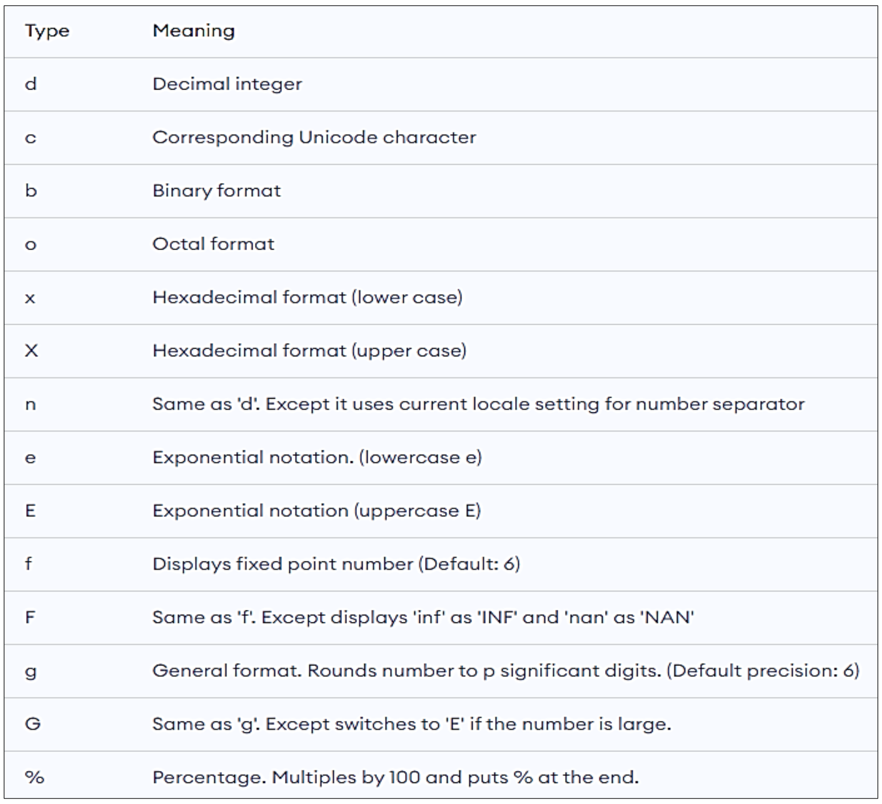
- **value** - value that needs to be formatted
- **format_spec** - The specification on how the value should be formatted.

The format specifier could be in the format:

[[fill]align][sign][#][0][width][,][.precision][type]

where, the options are

```
fill       ::=  any character
align      ::=  "<" | ">" | "=" | "^"
sign       ::=  "+" | "-" | " "
width      ::=  integer
precision  ::=  integer
type       ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```



Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
o	Octal format
x	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
n	Same as 'd'. Except it uses current locale setting for number separator
e	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiples by 100 and puts % at the end.

<u>Example</u>

```
print(format(2.2345,"10.2f"))
print(format(2,"*>10d"))
```

<u>Output</u>

```
'      2.23'
'*********2'
```

<u>bin(num)</u>

Returns binary equivalent of given integer prefixed with ‘0b’.

<u>Example</u>

```
print(bin(4))
```

<u>output</u>

```
'0b100'
```

<u>hex(num)</u>

Converts given integer into its hexadecimal equivalent prefixed with ‘0x’.

<u>Example</u>

```
print(hex(10))
```

<u>output</u>

```
'0xa'
```

<u>oct(num)</u>

Converts given integer into its octal version prefixed with ‘0o’.

<u>Example</u>

```
print(oct(8))
```

<u>output</u>

```
'0o10'
```

float(value) -> converts given value into a floating point number
int(value, base) -> converts given value into integer, 'base' is optional and its default value is 10.
str(object, encoding, errors) -> 2nd and 3rd arguments are optional, they specify the encoding used (default UTF=8) and action to be taken if decoding fails respectively.

Examples
```
print(float(3))
print(int(2.34))
s=str(230)
print(s, type(s))
```
output
```
3.0
2
230 <class 'str'>
```

len(object)
Returns number of items in an object.
Example
```
lis = [1,20,0,10]
print(len(lis))
```
output
```
4
```

min(arguments) / min(iterable)
Returns the item with lowest value.
Example
```
lis = [1,20,0,10]
print(min(lis))
```
output
```
0
```

max(arguments) / max(iterable)
Returns the item with highest value.
Example
```
lis = [1,20,0,10]
print(max(lis))
```
output
```
20
```

divmod(dividend, divisor)
The dividend is divided by the divisor and returns a pair of numbers representing the quotient and remainder.
Example
```
print(divmod(5,3))
```
output
```
(1,2)
```

pow(x, y, z)

Returns value of x to the power of y. ‘z’ is optional, and if given, power modulus z is returned.

Example

```
print(pow(2,3))
print(pow(2,3,5))
```

output

```
8
3
```

sum(iterable,start)

Sum of items in iterable is returned. Second argument ‘start’ is optional and if given, the sum found is added by ‘start’.

Example

```
lis = [1,20,0,10]
print(sum(lis))
print(sum(lis, 10))
```

output

```
31
41
```

round(number, digits)

Returns a floating point number which will be the rounded version of given argument. The decimal places to which the number is rounded can be specified by ‘digits’. It is optional and default value is 0.

Example

```
print(round(2.34561,2))
```

output

```
2.35
```

iter(object) , next(iterable, default)

These two functions are mostly used along with each other, because iter() creates an object which can be iterated one element at a time and next() returns the next item in an iterator. The ‘default’ argument is optional and if specified, denotes the default return value to return when iterable has reached to its end.

Example

```
lis = [1,2,3,4]
i = iter(lis)
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i,"End of list"))
```

output

```
1
2
3
4
End of list
```

dir() Function

dir() takes an object as an argument. It returns a list of strings which are names of members of that object. If object is a module, it will list sub-modules, functions provided by, variables, constants, etc.

help() FUNCTION

help() function is a built-in function in Python Programming Language which is used to invoke the help system. It takes an object as an argument. It gives all the detailed information about that object like if it’s a module, then it will tell you about the sub-modules, functions, variables and constants in details.

SEE NEXT PAGE

Mathematical functions

In python, there are two types of pre-defined functions.

- **Inbuilt functions**: These are functions which doesn't require any other code file such as, **Modules** or **Library Files**. These are a part of python core and are built within the Python compiler hence there is no need to import these modules/libraries in our code.
- The second type of functions require some external files(modules) in order to be used. The process of using these external files in our code is called **importing**. So, we have to import the file into our code and use the functions which are already written in that file.

Three way to import functions from a module :

1) **from math import sqrt, log**

   'math' is the name of the module and 'sqrt', 'log' are functions from math module. By using this form of import statement, only the specified functions are available to be used in our program.

2) **from math import ***

   The whole code of math module is made available to the program by this statement. All the functions in the module is made available by using *.

3) **import math**

   Using this form of import statement, all functions are made available to the program, but to use a function, its "qualified name" is to be stated. It is the module name along with the function name. For example, **math.sqrt(x)**, **math.log10(100)**. This avoids name collision if the program uses functions or variables with same name as in math module.

Numeric Functions

ceil(x) - The smallest integer greater than or equal to the x value is returned
floor(x) - The largest integer less than or equal to the x value is returned
fabs(x) - Absolute value for the x is returned
copysign(x, y) - Using the sign of y, the value for x is returned
gcd(x,y) – The greatest common divisor for x and y is returned
comb(n,k) – The combination value is found by, [n! / k!(n-k)!]
perm(n,k) – Permutation is found by using the formula [n!/(n-k)!]
fsum(iterable) - Returns an accurate floating-point sum of values in the iterable

Example :

```
import math
print(math.ceil(2.3))
print(math.floor(2.3))
print(math.fsum([1,2,3]))
```

Output :

```
3
2
6.0
```

Power and Logarithmic Functions

exp(x) - Returns e**x
log(x[, base]) - x to the base logarithm is returned
pow(x, y) - Returns x raised to the power y
sqrt(x) - Square root value for x is returned

Example :

```
import math
print(math.exp(2))
print(math.pow(2,3))
```

Output :

```
7.38905609893065
8.0
```

Trigonometric Functions

sin(x) - sine value of x in radians is determined
cos(x) - cosine value of x in radians need to be determined
tan(x) - tangent value of x in radians need to be determined
degrees(x) - radian to degree conversion
radian(x) - the degree to radian conversion

Example :

```
import math
print(math.sin(0))
print(math.tan(90))
print(math.degrees(1))
```

Output :

```
0.0
-1.995200412208242
57.29577951308232
```

Hyperbolic Functions

sinh(x)
cosh(x)
tanh(x)

Returns hyperbolic value of x

acosh(x)
asinh(x)
atanh(X)

Returns inverse hyperbolic value of x

Math Constants

A mathematical constant is a value that is represented by some special symbols. Some constants available in **math** module are :
math.pi
math.e
math.tau
Their values are respectively,
3.141592653589793
2.718281828459045
6.283185307179586

continued………..

Date-Time Functions

Date and time objects may be categorized as **“aware” or “naive”** depending on whether or not they include timezone information.

1. Aware — It contains time zone and daylight saving time information, so that it can locate itself relative to other aware objects.
2. Naive — This one is easier to work with because it doesn’t contain information about different timezones and daylight saving times. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other timezone is purely up to the program.

1) DATETIME MODULE

The **datetime** module supplies classes for manipulating dates and times. To use functions in classes provided by this module,make sure to **import datetime** module into the program.

The datetime module exports the following constants:

- datetime.**MINYEAR**
  The smallest year number allowed in a date or datetime object. MINYEAR is 1.
- datetime.**MAXYEAR**
  The largest year number allowed in a date or datetime object. MAXYEAR is 9999.

Six main classes available in datetime module to manipulate date and time include :

1) class datetime.**date**
   An idealized naive date, assuming the current Gregorian calendar.
   Attributes: year, month, and day.

2) class datetime.**time**
   An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds.
   Attributes: hour, minute, second, microsecond, and tzinfo.

3) class datetime.**datetime**
   A combination of a date and a time.
   Attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

4) class datetime.**timedelta**
   A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.

5) class datetime.**tzinfo**
   An abstract base class for time zone information . These are used by datetime and time classes to provide a customizable notion of time adjustment

6) class datetime.**timezone**
   A class that implements the tzinfo abstract base class as a fixed offset from the UTC.

Objects of date, datetime, time, and timezone types are immutable.

[1] date Objects

A date object can be created by using the constructor :

```
datetime.date(year, month, day)
```

All arguments are required. Arguments must be integers, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year

If an argument outside those ranges is given, **ValueError** is raised.

Date objects are also created using :

date.**today**()

Return the current local date.

date.**fromtimestamp**(*timestamp*)

Return the local date corresponding to the POSIX timestamp.

[Note : The Unix epoch (or Unix time or POSIX time or Unix timestamp) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT)]

date.**fromordinal**(*ordinal*)

Return the date corresponding to the given ordinal, where January 1 of year 1 has ordinal 1.

Example :

```
import datetime
import time
print(datetime.date(2021,10,1))
print(datetime.date.today())
print(datetime.date.fromtimestamp(time.time()))
print(datetime.date.fromordinal(1))
```

```
Output
2021-10-01
2021-10-19
2021-10-19
0001-01-01
```

time.time() is a function from* ***time* *module that returns time in seconds since the epoch as floating point number.*

Other methods in date class include :

date.replace(*year=self.year*, *month=self.month*, *day=self.day*)

Return a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.

date.timetuple()

Return a time.struct_time. (from time module)

date.toordinal()

Return the Gregorian ordinal of the date.

date.weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6.

date.isoweekday()
Return the day of the week as an integer, where Monday is 1 and Sunday is 7.

date.isocalendar()
Return a named tuple object with three components: year, week and weekday.

date.isoformat()
Return a string representing the date in ISO 8601 format, YYYY-MM-DD

date.__str__()
For a date *d*, str(d) is equivalent to d.isoformat()

date.ctime()
Return a string representing the date

date.strftime(*format*)
Return a string representing the date, controlled by an explicit format string. (format code list is givenafter the example)

date.__format__(*format*)
Same as date.strftime(). This makes it possible to specify a format string for a date object in formatted string literals.

Example :

```
import datetime
d = datetime.date(2021,10,19)
print(d)
newd=d.replace(month=11)
print("output 1 : ",newd)
print("output 2 : ",d.timetuple())
print("output 3 : ",d.toordinal())
print("output 4 : ",d.weekday())
```

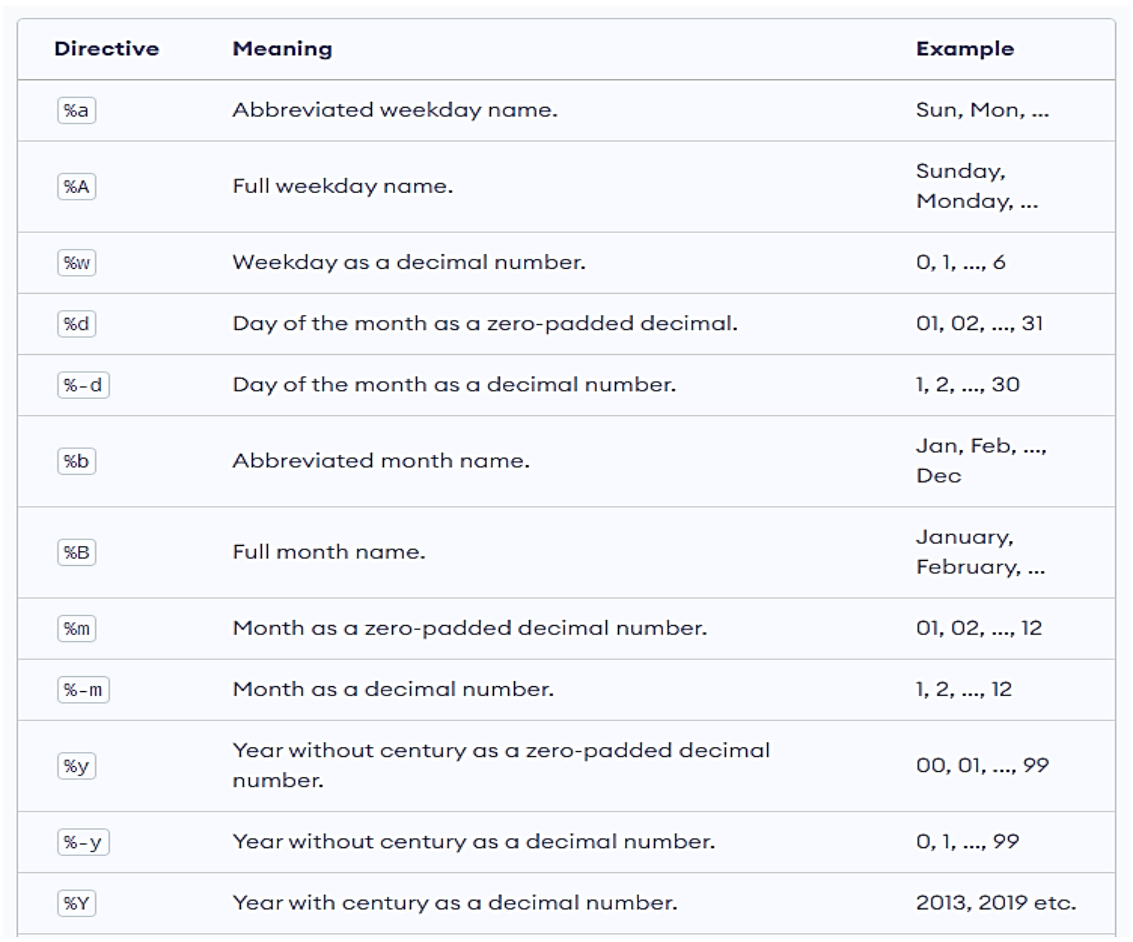
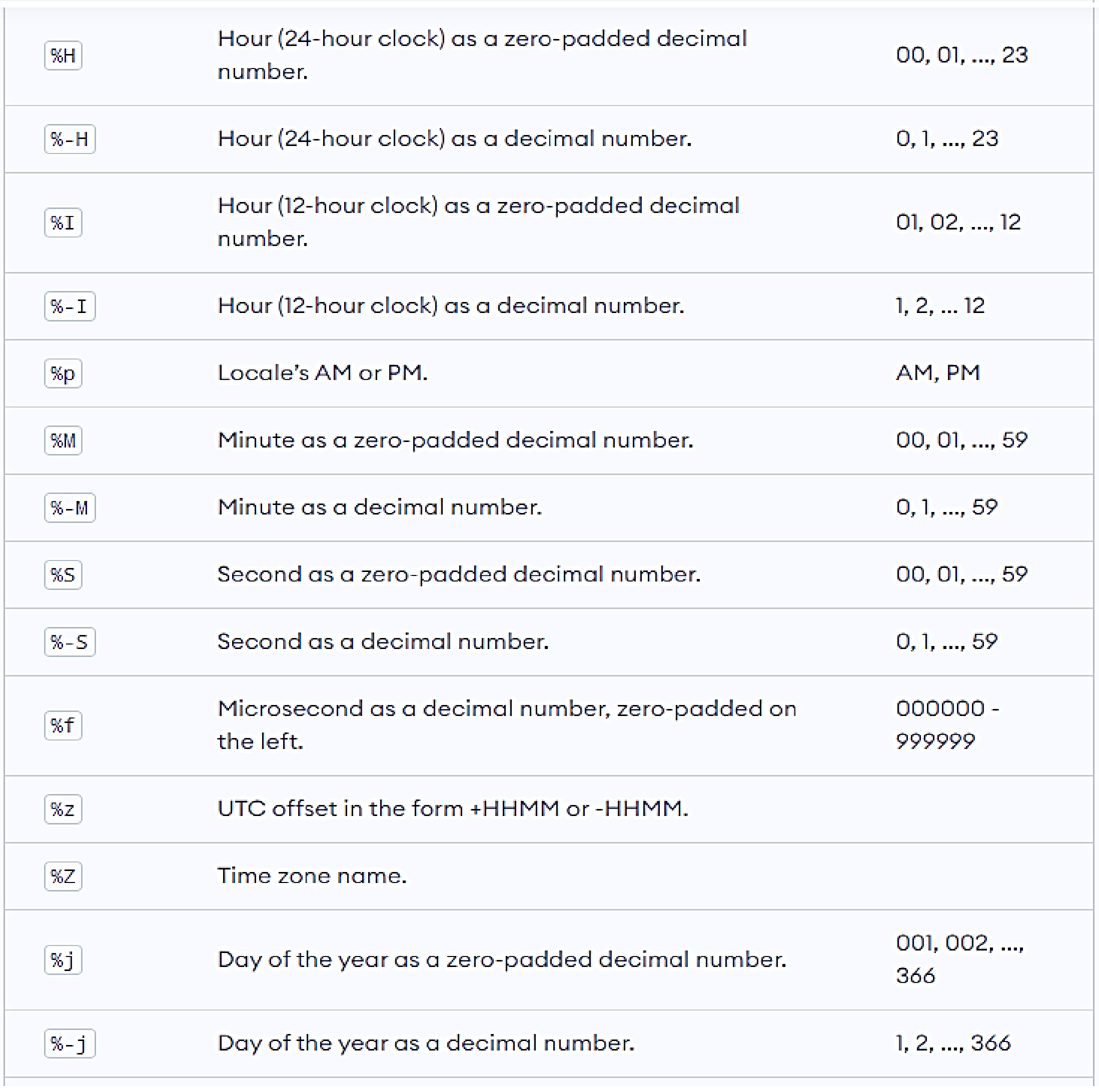
```
print("output 5 : ",d.isoweekday())
print("output 6 : ",d.isocalendar())
print("output 7 : ",d.isoformat())
print("output 8 : ",d.__str__())
print("output 9 : ",d.ctime())
print("output 10 : ",d.strftime("%m/%d/%Y, %H:%M:%S"))
formStr = "%d %B, %Y"
print("output 11 : ",d.__format__(formStr))
```

Output

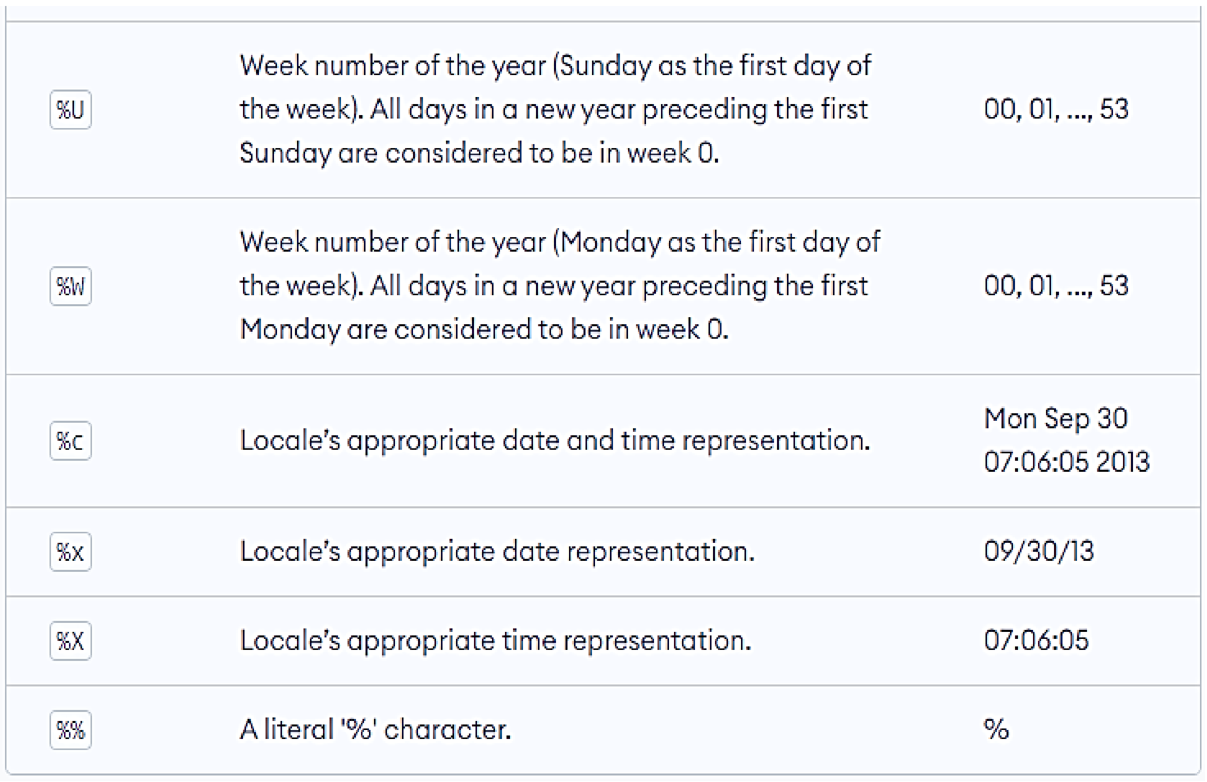
```
2021-10-19
output 1 :  2021-11-19
output 2 :  time.struct_time(tm_year=2021, tm_mon=10, tm_mday=19, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=292, tm_isdst=-1)
output 3 :  738082
output 4 :  1
output 5 :  2
output 6 :  datetime.IsoCalendarDate(year=2021, week=42, weekday=2)
output 7 :  2021-10-19
output 8 :  2021-10-19
output 9 :  Tue Oct 19 00:00:00 2021
output 10 :  10/19/2021, 00:00:00
output 11 :  19 October, 2021
```

Format Code List

The table below shows all the codes that you can pass to the strftime() method.

Directive	Meaning	Example
`%a`	Abbreviated weekday name.	Sun, Mon, ...
`%A`	Full weekday name.	Sunday, Monday, ...
`%w`	Weekday as a decimal number.	0, 1, ..., 6
`%d`	Day of the month as a zero-padded decimal.	01, 02, ..., 31
`%-d`	Day of the month as a decimal number.	1, 2, ..., 30
`%b`	Abbreviated month name.	Jan, Feb, ..., Dec
`%B`	Full month name.	January, February, ...
`%m`	Month as a zero-padded decimal number.	01, 02, ..., 12
`%-m`	Month as a decimal number.	1, 2, ..., 12
`%y`	Year without century as a zero-padded decimal number.	00, 01, ..., 99
`%-y`	Year without century as a decimal number.	0, 1, ..., 99
`%Y`	Year with century as a decimal number.	2013, 2019 etc.
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
`%-H`	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
`%-I`	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
`%p`	Locale's AM or PM.	AM, PM
`%M`	Minute as a zero-padded decimal number.	00, 01, ..., 59
`%-M`	Minute as a decimal number.	0, 1, ..., 59
`%S`	Second as a zero-padded decimal number.	00, 01, ..., 59
`%-S`	Second as a decimal number.	0, 1, ..., 59
`%f`	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
`%z`	UTC offset in the form +HHMM or -HHMM.	
`%Z`	Time zone name.	
`%j`	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
`%-j`	Day of the year as a decimal number.	1, 2, ..., 366



%U	Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, ..., 53
%W	Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, ..., 53
%c	Locale's appropriate date and time representation.	Mon Sep 30 07:06:05 2013
%x	Locale's appropriate date representation.	09/30/13
%X	Locale's appropriate time representation.	07:06:05
%%	A literal '%' character.	%

[2] time Objects

A time object represents a (local) time of day. It allows us to manipulate date without interfering date (hour, minute, second, microsecond).

A time object can be created using :

datetime.**time**(*hour*, *minute*, *second*, *microsecond*, *tzinfo*)

Methods of time class :

time.replace(hour, minute, second, microsecond, tzinfo)
Return a time with the same value, except for those attributes given new values by whichever keyword arguments are specified.

time.isoformat()
Return a string representing the time in ISO 8601 format

time.__str__()
For a time *t*, str(t) is equivalent to t.isoformat().

time.strftime(*format*)
Return a string representing the time, controlled by an explicit format string.

time.__format__(*format*)
Same as time.strftime(). This makes it possible to specify a format string for a time object in formatted string literals.

time.utcoffset()
Returns the difference between the local time and UTC time. If tzinfo is None, returns None.

time.dst()
If tzinfo is None, returns None, else returns DST(Daylight Savings Time) information.

time.tzname()
If tzinfo is None, returns None, else returns time zone name.

Example :

```
import datetime
t = datetime.time(11,10,30)
print("time :",t)
newt=t.replace(second=50)
print("output 1 : ",newt)
print("output 2 : ",t.isoformat())
print("output 3 : ",t.__str__())
print("output 4 : ",t.strftime("%H:%M %p"))
formStr = "%H:%M:%S:%f"
print("output 5 : ",t.__format__(formStr))
print("output 6 : ",t.utcoffset())
print("output 7 : ",t.dst())
print("output 8 : ",t.tzname())
```

Output :

```
time : 11:10:30
output 1 :  11:10:50
output 2 :  11:10:30
output 3 :  11:10:30
output 4 :  11:10 AM
output 5 :  11:10:30:000000
output 6 :  None
output 7 :  None
output 8 :  None
```

[3] datetime Objects

A datetime object is a single object containing all the information from a date object and a time object.

A datetime object can be created by :

datetime.**datetime**(*year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, *tzinfo*)

Other constructors and methods include :
datetime.**today**()
datetime.**now**(*tz=None*)
datetime.**utcnow**()
datetime.**fromtimestamp**(*timestamp*, *tz=None*)
datetime.**utcfromtimestamp**(*timestamp*)
datetime.**fromordinal**(*ordinal*)
datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)
datetime.**fromisoformat**(*date_string*)
datetime.**fromisocalendar**(*year*, *week*, *day*)

to create datetime objects

datetime.date()
Return date object with same year, month and day.

datetime.time()
Return time object with same hour, minute, second, microsecond

datetime.timetz()
Return time object with same hour, minute, second, microsecond, and tzinfo attributes.

datetime.replace(*year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, *tzinfo*)
Return a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified.

datetime.utcoffset()
If tzinfo is None, returns None, else returns the UTC offset from local time

datetime.dst()
If tzinfo is None, returns None, else returns DST(Daylight Savings Time) information.

datetime.tzname()
If tzinfo is None, returns None, else returns time zone name.

datetime.timetuple()
Return a time.struct_time with respective attributes.

datetime.weekday()
datetime.isoweekday()
datetime.isocalendar()
datetime.isoformat
datetime.__str__()
datetime.ctime()
datetime.strftime(*format*)
datetime.__format__(*format*)

Similar to functions defined in **date** class except **datetime** includes time attributes too.

Similar example to the one given in **date** class :

```
import datetime
d1 = datetime.datetime(2021,10,19,11,10,30)
print(d1)
print("output 1 : ",d1.weekday())
print("output 2 : ",d1.isoweekday())
print("output 3 : ",d1.isocalendar())
print("output 4 : ",d1.isoformat())
print("output 5 : ",d1.__str__())
print("output 6 : ",d1.ctime())
print("output 7 : ",d1.strftime("%m/%d/%Y, %H:%M:%S"))
formStr = "%d %B, %Y"
print("output 8 : ",d1.__format__(formStr))
```

Output

```
2021-10-19 11:10:30
output 1 :  1
output 2 :  2
output 3 :  datetime.IsoCalendarDate(year=2021, week=42, weekday=2)
output 4 :  2021-10-19T11:10:30
output 5 :  2021-10-19 11:10:30
output 6 :  Tue Oct 19 11:10:30 2021
output 7 :  10/19/2021, 11:10:30
output 8 :  19 October, 2021
```

[4] timedelta Objects

A timedelta object represents a duration, the difference between two dates or times.
To create a timedelta object :

datetime.**timedelta**(*days*, *seconds*, *microseconds*, *milliseconds*, *minutes*, *hours*, *weeks*)

timedelta.total_seconds()
Return the total number of seconds contained in the duration.

Example

```
import datetime
td = datetime.timedelta(days=1,seconds=33)
print(td)
print("Total seconds in the given duration:",td.total_seconds())
```

Output

```
1 day, 0:00:33
Total seconds in the given duration: 86433.0
```

[5] tzinfo Objects

class datetime.**tzinfo**
This is an abstract base class, meaning that objects for this class cannot be made directly. Therefore, we must define a subclass of tzinfo to capture information about a particular time zone.

Methods
tzinfo.utcoffset(*dt*)
Return offset of local time from UTC.

tzinfo.dst(*dt*)
Return the daylight saving time (DST) adjustment

tzinfo.tzname(*dt*)
Return the time zone name corresponding to the datetime object

[6] timezone Objects

The timezone class is a subclass of tzinfo, each instance of which represents a timezone defined by a fixed offset from UTC.

To create a time zone use :
datetime.timezone(*offset*, *name=None*)
The *offset* argument must be specified as a timedelta object representing the difference between the local time and UTC.

Methods
timezone.utcoffset(*dt*)
timezone.tzname(*dt*)
timezone.dst(*dt*)

strftime() and **strptime()** In Python

The **strftime** converts date object to a string date. It is a method of **date, datetime, time** classes.
Syntax : **dateobject.strftime(format)**

strptime python is used to convert string to datetime object. Method of **datetime** class only.
Syntax : **datetime.strptime(date_string, format)**

Example
```
import datetime
d=datetime.date(2021,10,1)
print("strftime : ",d.strftime("%a %d/%m/%Y"))
print("strptime : ",datetime.datetime.strptime("1/10/2021","%d/%m/%Y"))
```

Output
```
strftime :  Fri 01/10/2021
strptime :  2021-10-01 00:00:00
```

2) TIME MODULE

This module provides various time-related functions. To use time related functions from time module, make sure to **import time** module into the program.

Some functions from time module are :

time.time()

The time() function returns the number of seconds passed since epoch.

time.**ctime**([*secs*])

Convert a time expressed in seconds since the epoch to a string containing weekday, date and time.

time.sleep()

The sleep() function suspends (delays) execution of the current thread for the given number of seconds.

time.**struct_time Class**

Several functions in the time module such as gmtime(), asctime() etc. either take time.struct_time object as an argument or return it.

time.localtime()

The localtime() function takes the number of seconds passed since epoch as an argument and returns struct_time in **local time**.

time.gmtime()

The gmtime() function takes the number of seconds passed since epoch as an argument and returns struct_time in **UTC**.

time.asctime()

The asctime() function takes struct_time (or a tuple containing 9 elements corresponding to struct_time) as an argument and returns a string representing it.

Example :

```
import time
print("1) Seconds since epoch : ",time.time())
print("2) Local time by counting the seconds from epoch :",time.ctime(100))
time.sleep(5)
print("3) This is printed after 5 seconds")
print("4) Date and time info in struct_time tuple format:", time.localtime(100))
```

Output

```
1) Seconds since epoch :  1634790693.8638966
2) Local time by counting the seconds from epoch : Thu Jan  1 05:31:40 1970
3) This is printed after 5 seconds
4) Date and time info in struct_time tuple format: time.struct_time(tm_year=1970, tm_mon=1,
tm_mday=1, tm_hour=5, tm_min=31, tm_sec=40, tm_wday=3, tm_yday=1, tm_isdst=0)
```

random MODULE IN PYTHON

Python **random module** is an inbuilt module of python that is used to generate random numbers in python. There are many inbuilt functions available in the random module. Some of them are:

random.random()

It generates random floating numbers between 0.0 and 1.0. This function does not take any parameters.

random.randint(a,b)

This function takes two parameters, starting value and ending value. Parameters should be of integer type. The function will return a random number between ‘a’ and ‘b’.

random.randrange(start,stop,step)

Just like range function a list of integer value depending upon and starting and ending value and step value is considered and any random value from this list is returned.

random.choice(Sequence)

This function takes a single parameter. This parameter might be String or List or dictionary or any other sequence type, and it will return a random value from that sequence.

random.shuffle(sequence)

It takes any sequence and reorganize the order of the items.

random.sample(population,k)

Returns a ‘k’ length list of unique elements chosen from the population sequence or set.

random.seed(value)

Seed function is used to save the state of a random function, so that it can generate same random numbers on multiple executions of the code. Any value can be given to the parameter. To get the same random number use the same value given before.

Example :

```
import random
print("Output 1 : ",random.random())
print("Output 2 : ",random.randint(1,10))
print("Output 3 : ",random.randrange(40,51,2))
print("Output 4 : ",random.choice(['hi',11,'hello',22]))
lis = ['Amy','Catherine','Dennis']
random.shuffle(lis)
print("Output 5 : ",lis)
lis2 = [1,1,2,3,4,4,3,2,3,4,5,6,7,6,5,5,6,7,9,10,10,8,1]
print("Output 6 : ",random.sample(lis2,5))
random.seed(1)
print("A random number : ",random.random())
random.seed(1)
print("Same random number again : ",random.random())
```

```
Output 1 :  0.9846159116405021
Output 2 :  7
Output 3 :  40
Output 4 :  hi
Output 5 :  ['Catherine', 'Amy', 'Dennis']
Output 6 :  [6, 7, 1, 5, 4]
A random number :  0.13436424411240122
Same random number again :  0.13436424411240122
```

USER DEFINED FUNCTIONS

Functions are self-contained programs that perform some particular tasks. Once a function is created by the programmer, this function can be called anytime to perform the specific task.
There are many advantages of using functions:

a) They reduce duplication of code in a program.
b) They break the large complex problems into small parts.
c) They help in improving the clarity of code (i.e., make the code easy to understand).
d) A piece of code can be reused as many times as we want with the help of functions.

We have already seen built-in functions in python. These are some built-in functions in the Python programming language that can convert one type of data into another type. For example, the int() function can take any number value and convert it into an integer. This is called **type conversion**.

There is also another kind of type conversion in the Python language, known as implicit conversion. Implicit conversion is also known as **type coercion** and is the process through which the Python interpreter automatically converts a value of one type into a value of another type according to the requirement.

Example (type conversion/ explicit conversion)

```
a = 2.5
print("value of a : ",a,"\n type of a : ",type(a))
c = int(a)
print("value of c : ",c,"\n type of c : ",type(c))
```

Output

```
value of a :  2.5
 type of a :  <class 'float'>
value of c :  2
 type of c :  <class 'int'>
```

Example (type coercion / implicit conversion)

```
a = 2
c = 5/a
print("value of c : ",c,"\n type of c : ",type(c))
```

Output

```
value of c :  2.5
 type of c :  <class 'float'>
```

Now, we are going to discuss about **user defined functions**. Users have to define the function to use their own functions to carry out a particular task.
There are two aspects to every Python function:

- **Function definition.** The definition of a function contains the code that determines the function’s behaviour.
- **Function invocation/function call**. A function is used within a program via a function invocation.

An ordinary function definition consists of three parts:

- Name—Most Python functions have a name. The name is an identifier.
- Parameters—The parameters appear in a parenthesized comma-separated list that accepts values passed to the function.
- Body—every function definition has a block of indented statements that constitute the function's body. The body contains the code to execute when clients invoke the function.

Syntax

```
def function_name(parameters) :
        " " " function_docstring " " "
        function_statements
        return [expression]
```

- The block of the function starts with a keyword **def** after which the function name is written followed by parentheses.
- We can also give some input parameters by placing them within these parentheses.
- The block of statements always starts with a colon (:). After the colon, the statements belonging to a function is **indented with same number of spaces from left**.
- We can optionally include a **docstring** as the first thing inside a function withing triple quotes. The user can define what is the use of the function and, how the variables and statements are used for its proper working in the docstring. It is different from comments as this docstring can be accessed using __doc__ attribute. It is used to enhance readability and understandability of a function.
- After writing the code statements, the block is ended with a return statement. Whenever a return statement is executed in a function, it terminates the function and may return some value to the place from where the function was called. If the return statement is not given, the last statement of the function terminates the function and the control of program is given back to the called line.

Example

```
def sum(a,b) :
   c=a+b
   return c

var1 = int(input("Enter first number : "))
var2 = int(input("Enter second number : "))
print("Sum of var1 and var2 is : ",sum(var1,var2))
```

Output

```
Enter first number : 3
Enter second number : 6
Sum of var1 and var2 is :  9
```

COMPOSITION OF FUNCTIONS

The syntax of composition in mathematics is as follows:

f(g(x)) = f o g(x),

where, f and g are functions. This means the return value of function 'g' is passed into the function 'f' as parameters/arguments.

Just as with the mathematical functions, Python functions can also be composed. We can use any kind of expression including arithmetic operators as an argument to a function.

Example :

```
import math
x = math.exp(math.log(10.0))
print(x)
```

Here, the value of the function math.log(10.0)is calculated first and then used as the argument for the function math.exp.

PARAMETERS AND ARGUMENTS

- Arguments or parameters appearing in the function call are called **actual parameters/actual arguments**
- Arguments or parameters appearing in the function definition header are called **formal parameters/formal arguments**

Even though 'parameters' and 'arguments' are used interchangeably to refer the variables or values inside a function parenthesis whether it is in function header or function call, a standard explanation is given below.

- **Parameters** are variables inside the parenthesis in function definition header that accepts values from function call
- **Arguments** are values inside the parenthesis of function call that is sent to the function definition code.

There can be four types of formal arguments using which a function can be called which are as follows:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

1. Required arguments

While calling a function, the arguments should be passed to the function in correct positional order; and the number of arguments should match the defined number of parameters. These arguments which are passed in the same order as that of the parameters to which they have to be assigned are called **positional arguments/required arguments**.

Example

```
def display(name, age):
   print("My name is", name, "and my age is", age)

display("Arun", 25)
```

Output

```
My name is Arun and my age is 25
```

2. Keyword arguments

In keyword arguments, the arguments are recognized by the parameter’s names. This type of argument can also be out of order.

Example

```
def display(name, age):
   print("My name is", name, "and my age is", age)

display(age = 25, name = "Arun")
```

Output

```
My name is Arun and my age is 25
```

Note that arguments name and age is out of order in the function call and how the values are passed using the ‘keyword = value’ format.

3. Default arguments

In default arguments, we can assign a value to a parameter at the time of function definition. This value is considered the default value to that parameter. If we do not provide a value to the parameter at the time of calling, it will not produce an error. Instead, it will pick the default value and use it.

Default arguments must follow non-default arguments when given together.

Example

```
def display(name, age = 25):
   print("My name is", name, "and my age is", age)

display("Arun")
display("Archana", 24)
```

Output

```
My name is Arun and my age is 25
My name is Archana and my age is 24
```

In the above example, age=25 in the function definition is the default argument. The first function call has only one argument name, the second argument value is taken as the default value 25. In the second function call, there are two arguments and therefore the second argument 24 overrides the default value 25.

4. Variable-length arguments/Arbitrary Arguments

There are many cases where we are required to process a function with more number of arguments than we specified in the function definition. These types of arguments are known as variable-length arguments.

There are two types of arbitrary arguments.

1. Arbitrary positional arguments
2. Arbitrary keyword arguments

Arbitrary Positional Arguments

An asterisk (*) is used before the parameter name for arbitrary positional arguments.

Syntax : **def function_name(*parameter_name)**

We can pass any number of non-keyword arguments to this parameter.

Example

```
def test(*marks) :
   print("MARKS :",marks)
   sum=0
   for m in marks :
      sum=sum+m
   print("TOTAL MARKS :",sum)

print("TEST SERIES")
test(10,10,20)
```

Output

```
TEST SERIES
MARKS : (10, 10, 20)
TOTAL MARKS : 40
```

In the above example, 3 arguments (10,10,20) are passed to function test() and the variable 'marks' accept them as a **tuple**.

Arbitrary Keyword Arguments

A double asterisk (**) is used before the parameter name for arbitrary keyword arguments. We can pass any number of keyword arguments to this parameter.

Syntax : **def function_name(**parameter_name)**

Example

```
def test(**marks) :
   print("MARKS :",marks)
   sum=marks['sub1']+marks['sub2']+marks['sub3']
   print("TOTAL MARKS :",sum)

print("TEST SERIES")
test(sub1=10, sub2=10, sub3=20)
```

Output

```
TEST SERIES
MARKS : {'sub1': 10, 'sub2': 10, 'sub3': 20}
TOTAL MARKS : 40
```

Here, 3 arguments are accepted by the function as **dictionary** having key-value paris.

In arbitrary positional arguments, the passed values get wrapped up into a tuple. Whereas, in arbitrary keyword arguments, the passed values get wrapped up into a dictionary.

FUNCTION CALLS

- A function is called using the name with which it was defined earlier, followed by a pair of parentheses (()).
- Any input parameters or arguments are to be placed within these calling parentheses.

In Python, values are **passed to function by object reference**.

- If object is immutable(not modifiable) then a new value assigned to it inside function definition will not reflect in parameters of called function.
  int, float, complex, string, tuple, frozen set, bytes are all immutable.
- If object is mutable (modifiable) than modified value is available outside the function.
  list, dict, set, byte array are all mutable

Example

```
def func(a) :
   a=a+1
   print("Value of a inside function :",a)

print("PASSING IMMUTABLE OBJECT")
a=5
print("Value of a before calling function :",a)
func(a)
print("Value of a after calling function :",a)
```

Output

```
PASSING IMMUTABLE OBJECT
Value of a before calling function : 5
Value of a inside function : 6
Value of a after calling function : 5
```

```
def func(list) :
   list[1]=10
   print("list inside function :",list)

print("PASSING MUTABLE OBJECT")
list=[5,6]
print("list before calling function :",list)
func(list)
print("list after calling function :",list)
```

Output

```
PASSING MUTABLE OBJECT
list before calling function : [5, 6]
list inside function : [5, 10]
list after calling function : [5, 10]
```

THE return STATEMENT

The return statement is used to exit a function. A function may or may not return a value. If a function returns a value, it is passed back by the return statement. If it does not return a value, we simply write ‘return’ with no arguments or skip return statement.

We can also return multiple values from a function. The values are returned as tuple. These values may also be assigned to individual variables by tuple unpacking.

```
def prod(a,b) :
   p=a*b
   return p

print("RETURNING SINGLE VALUE")
a = 3
b = 5
c = prod(a,b)
print("The product is :",c)
```

```
RETURNING SINGLE VALUE
The product is : 15
```

```
def prodSum(a,b) :
   p=a*b
   s=a+b
   return p,s

print("RETURNING MULTIPLE VALUES")
a = 3
b = 5
c = prodSum(a,b)
print("The product and sum are :",c)
```

```
RETURNING MULTIPLE VALUES
The product and sum are : (15, 8)
```

<u>Global and Local Variables</u>

- Variables defined within functions or inside a particular block are **local variables**. The memory required to store a local variable is used only when the variable is in scope. [**Scope of a variable is the region of program over which that variable can be accessed or is 'active'.**]
  Therefore, same variable name can be used in different functions without any conflict since they are not accessible outside their local scope.

- A **global variable** lives as long as the program is running. In contrast to a local variable, a global variable is defined outside of all functions and is not local to any particular function. Any function can legally access and/or modify a global variable.

```
glob = 10

def func(a):
   s = a+glob
   print("Local variable s =",s)

print("Global variable =",glob)
a=1
func(a)
```

This program will print:
Global variable = 10
Local variable s = 11
Note that the variable 'glob' is accessible inside func() and to the block below it.

Now, consider the same code above with an additional statement :

```
glob = 10

def func(a):
   s = a+glob
   print("Local variable s =",s)

print("Global variable =",glob)
a=1
func(a)
print("Sum =",s)
```

Here, the last statement tries to print the value of variable ‘s’, which is a local variable of func(). Local variables are not accessible outside their local scope(here the scope is inside the function only). Therefore, this program will show error.

RECURSION

If a function, procedure or method calls itself, it is called **recursive**. Recursion is calling a function inside the same function.

As an example, consider a program to find factorial of a number. Factorial of any number ‘n’ is :
n! = (n)*(n-1)*(n-2)*....*1

```
def factorial(x):
   if x==0 or x==1:
      return 1
   else:
      return x*factorial(x-1)

print("Factorial of 4 is, 4! =",factorial(4))
```

Output

```
Factorial of 4 is, 4! = 24
```

Note the statement **return x*factorial(x-1)**. For the value 4, the function calls and return statements go through the stages :

Inside factorial(4),
   a) **return 4*factorial(3)**

factorial(3),
   b) **return 3*factorial(2)**

factorial(2),
   c) **return 2*factorial(1)**

factorial(1)
   d) **return 1**

- **return 1** returns value 1 to **factorial(1)** of (c)
- (c) becomes **return 2*1**, therefore value 2 is returned to **factorial(2)** in (b)
- (b) becomes **return 3*2**, therefore value 6 is returned to **factorial(4)** in (a)
- (a) becomes **return 4*6,** finally, value 24 is returned to the statement that called the function and 24 is printed as 4!.

ANONYMOUS/LAMBDA FUNCTIONS

Anonymous functions are also called lambda functions in Python because instead of defining them with the standard 'def' keyword, they are defined using '**lambda' keyword** and are defined **without any name**.

Example

```
square = lambda x: x*x

print("4*4 =",square(4))
print("5*5 =",square(5))
```

Output

```
4*4 = 16
5*5 = 25
```

lambda x: x*x is the anonymous or lambda function. **x** is the **argument**, and **x*x** is the expression or statement that gets evaluated and returned.

It is equivalent to :

```
def square(x) :
        return x*x
```

Lambda function with multiple arguments :

```
exp = lambda x,y,z : (x+y)*z

x=5
y=5
z=10
print("result =",exp(x,y,z))
```

Output

```
result = 100
```