Python String format() Method

String Methods

Example

Insert the price inside the placeholder, the price should be in fixed point, two-decimal format:

```
txt = "For only {price:.2f} dollars!"
print(txt.format(price = 49))
```

Try it Yourself »

Definition and Usage

The `format()` method formats the specified value(s) and insert them inside the string's placeholder.

The placeholder is defined using curly brackets: {}. Read more about the placeholders in the Placeholder section below.

The `format()` method returns the formatted string.

Syntax

```
string.format(value1, value2...)
```

Parameter Values

Parameter	Description
value1, value2...	Required. One or more values that should be formatted and inserted in the string. The values are either a list of values separated by commas, a key=value list, or a combination of both. The values can be of any data type.

The Placeholders

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

Example

Using different placeholder values:

```
txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age = 36)
txt2 = "My name is {0}, I'm {1}".format("John",36)
```

txt3 = "my name is {}, I'm {}".format("John", 36)

o/p = my name is John, I'm 36.

String operators : are used to manipulate the strings in multiple ways.

① + → concatenate two strings

Str = "Hello"

Str1 = "world"

eg: print(Str + Str1) → Hello world.

② * → repetation opr.

Print(Str*3) → HelloHelloHello

③ [:] → range slice opr. to access sub-strings

Print(Str[2:4]) → ll { leave value at index 4 }

④ [] → slice opr.

Print(Str[4]) → o

⑤ in → membership opr.

Print('w' in Str) → false.

⑥ Not in → membership opr.

Print('wo' not in Str1) → false

⑦ r/R → to specify raw string.

Print(r 'C://python37') → C://python37

Print As it is writte

⑧ %

Print("The string str: %s"%(Str)) → The string str: Hello

This is also known as % method.

eg: name = "Mohan"

print("Hello, %s!"%name) → Hello, Mohan.

tting Types txt3 = "My name is {}, I'm {}".format("John",36)
Formatting types:
de the placeholders you can add a formatting type to format the result:

:< Try it Left aligns the result (within the available space)

txt = "We have {:<8} chickens."
print(txt.format(49)) } o/p → we have 49 ____ chickens
8

:> Try it Right aligns the result (within the available space) If 8→0, then no effect.

o/p → we have _____49 chickens

:^ Try it Center aligns the result (within the available space)

____49____

:= Try it Places the sign to the left most position

txt = "The temperature is {:=8} degrees celsius."
print(txt.format(-5)) } O/P
The temperature is - 5deg

:+ Try it Use a plus sign to indicate if the result is positive or negative

txt = "temperature is between {:+} and {:+} degrees celsius."
print(txt.format(-3,7)) o/p → The temp is bet" -3 and +7 degrees celsius.

:- Try it Use a minus sign for negative values only

No sign for +ve value. Only sign for -ve. val.
↙ diff

: Try it Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)

✓ :, Try it Use a comma as a thousand separator

txt = "The universe is {:,} years old."
print(txt.format(13800000000)) } o/p. The universe is 13,800,000,000 years old.

:_ Try it Use a underscore as a thousand separator

✓ :b Try it Binary format

txt = "The binary version of {0} is {0:b}" } O/P
print(txt.format(5)) The binary version of 5 is 101.
↑ index.

:c Converts the value into the corresponding unicode character

:d Try it Decimal format

txt = "We have {:d} chickens." } o/p → we have 5 chickens.
print(txt.format(0b101))

:e Try it Scientific format, with a lower case e

:E Try it Scientific format, with an upper case E

:f Try it Fix point number format

General format

:G General format (using a upper case E for scientific notations)

:o Try it Octal format

```
txt = "The octal version of {0} is {0:o}"
print(txt.format(10))
```

o/p → The octal version of 10 is 12.

:x Try it Hex format, lower case

```
txt = "The Hexadecimal version of {0} is {0:x}"
print(txt.format(255))
```

o/p → The Hexadecimal version of 255 is ff.

:X Try it Hex format, upper case

:n Number format

:% Try it Percentage format

```
txt = "You scored {:%}"
print(txt.format(0.25))
```

o/p → You scored 25.000000%

For without any decimal.

```
txt = "You scored {:.0%}"
print(txt.format(0.25))
```

o/p → You scored 25%.

Another method of writing

```
Val=10
Print("decimal:{0:d}".format(val));   → decimal:10
Print("hex:{0:x}".format(val));       → a
Print("octal:{0:o}".format(val));     → 12
Print("binary:{0:b}".format(val));    → 1010
```

```
Val = 100 000000
Print("decimal:{:,}".format(val);     → decimal: 100,000,000
```

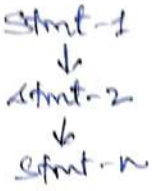
```
Print("decimal:{:.2%}".format(56/9); → decimal: 622.22%
```

(*) Types of statements:

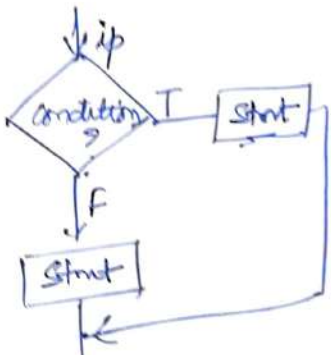
1. Empty stmt / Null operation stmt = it does nothing
   eg: Pass
2. Simple stmt : Any single executable stmt is a simple stmt
   eg: name = input("your name"), print(name)
3. Compound stmt. = group of stmts executed as a unit
   eg: block in python. , if a>b: → header
   body. { Print("a is greater")

(*) Stmt Flow control:

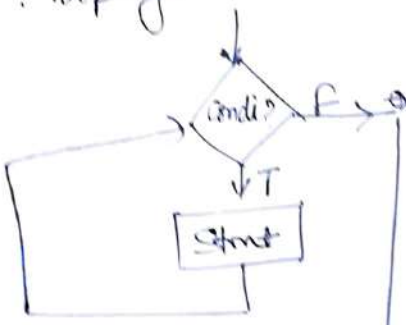
(1) Sequence :



(2) Selection :



(3) Iteration : looping.



There is no exit controlled loop in python.

(*) If statement: selection/decision construct/conditional

(1) Simple "if" syntax: if <conditional expression> : (True)
stmts

eg:
```
a = 33
b = 200
if b>a:
    Print("b is greater")
```

(*) If - Else:

```
If <Cond.Exp">:
    [Stmt]
    [Stmt]
else:
    [Stmt]
    [Stmt]
```

(colon are must.)

eg:
```
if a>=0:
    Print(a, "is zero or a positive no")
else:
    Print(a, "is a negative no")
```

(*) if- elif stmt: to check another condition in case the test condition of "if" evaluates to false.

eg:
```
a=33
b=33
if b>a:
    Print("b is greater than a")
elif a==b:
    Print("a and b are equal")
```

(*) if-elif-else stmt:

eg:
```
a=200
b=33
if b>a:
    Print("b is greater than a")
elif a==b:
    Print("a & b are equal")
else:
    Print("a is greater than b")
```

(*) Nested-if stmt

There are four cases:

case-1 → 'if' inside 'if's body.

```
If <Cond.Exp>:
    if <cond.exp2>:
        Stmts
    else:
        stmts.
elif: <cond.exp3>
    stmts.
else:
    stmts
```

case-2 if inside elif's body

```
if <cond.exp">:
    Stmts
elif <cond.Exp2>:
    if <C.E.3>:
        Stmts
    else:
        Stmts
else:
    stmts.
```

case-3 if inside else body.

```
If <C.E>:
    Stmts
elif<CE2>:
    Stmts
else:
    If <CE3>:
        Stmts
    else:
        stmts
```

case-4 if inside if's as well as else's body.

```
If (CondExp):
    if <C.E.1>:
        Stmts
    else:
        stmts
elif <CE2>:
    if <CE3>:
        stmts
    else:
        stmts
else if <CE4>:
    Stmts
    else:
        stmts
```

range() function:

Syntax: range(lower limit, upper limit), both limits should be integers.

eg: range(0,5), default step value is +1.

so it produces list [0,1,2,3,4]. This list should in starts with 0 to (A.P (n-1)).

range(5,0) → results to empty list. [] since A.P can be generated.

Range() with steps:

range(l, u, s) → step value.
lower lt, upper limit

eg: range(0,10,2) = [0,2,4,6,8]
range(5,0,-1) = [5,4,3,2,1]

in and not in operators.

eg: 3 in [1,2,3,4] → Yes.

Iteration/Loops statements:

= Repeating of statements.

Two kind of loops:
1. Counting loops eg: for loop
2. conditional loops eg: while loop.

1. For loop:

process items of any sequence. such as list, string,

```
for <variable> in <sequence>:
    statements to repeat.
```

eg:
```
for a in [1,4,7]:
    Print(a)
    Print(a*a)
```
o/p:
```
1
1
4
16
7
49.
```

or
```
for ch in 'calm':
    Print(ch)
```
o/p:
```
c
a
l
m
```

or
```
for val in range(3,18):
    Print(val)
```
o/p: 3,4,5, - - - , 17

(*) **while-loop** : is a conditional loop. It repeats stmts upto the condition remains true.

Syntax: while <logical exp"> :
    stmts
(Pass can be used here)

eg:
```
a = 5
while a > 0:
    print(a)
    a = a - 1
print("loop over")
```
→ update expression and it is must in loop; otherwise it will become infinite.

imp: 1) initialization : before entering into loop, variable must be initialised.
2) Test expression
3) Body of loop
4) update expression

(*) **Loop-else stmt** : for & while loop has else block. Here else of a loop executes only when the loop ends normally.

(*) **Jump-stmts**: < break / continue.

1) **break** → it skips over a part of the code. Also terminates the loop. Execution resumes at the stmt immediately following the body of the terminated stmt. Also used to terminate infinite loop.

eg:
```
while <test-condition>:
    Stmt 1
    If <condition>:
        break
    Stmt 2
    Stmt 3
Stmt-4
Stmt-5
```

2) **Continue**: it is somewhat opposite to break. It doesn't terminate the control at a point rather it forces the next iteration of the loop.

```
while <TC>:
    stmt-1
    If <condi>:
        continue
    Stmt 2
    Stmt 3
    Stmt-4
```

LIST

(*) **LIST:** Lists are containers that are used to store a list of values of any type. [ ] brackets used. It contains comma separated values. Every element is indexed starting with 0.

Features:

1) Python lists are mutables (we can change its elements)
2) Python will not create any fresh list if any change is made
3) It is a sequence like strings and tuples. but differ in mutability.
4) eg: [ ] → empty list. , ['a','b','c'] = list of character
[1,2,3] → list of integer , ['a', 1, 3.5, 'zero'] = mixed value
[1.1, 2, 3.3, 4] → list of mixed nos, ['one', 'two', 'four'] = list of strings

⇒ Lists and dictionaries are mutable
But, all other data types of Python are immutable.

(*) **Creating LIST:** Start and end of lists are shown by [and]. and all values are comma separated.

1. **Empty list:** [ ], It is equivalent of '0' or '' and it has truth value as false.

```
L = list()
```

2. **Long list:** list = [0, 1, 2, 3, ... 10, 11, 12, 13 ... , n]

3. **Nested lists:** A list can have an element in it, which itself is a list.

```
L = [3,4,[5,6],7]
index =  0 1  2 2 0 2 1  3
```

Here, Length of L is 4,

```
L[0] = 3
L[2] = [5,6]
L[2][0] = 5 ,   L[2][1] = 6.
```

(*) **Creating lists from Existing sequences:**

eg: L = list(<sequence>) → strings, tuples, lists

```
>>> L1 = list('hello')   string
>>> L1
['h','e','l','l','o']
>>> t = ('w','e','r') → tuple.
>>> L2 = list(t)
>>> L2
['w','e','r']
```

Shakti Raj Singh

* input integer but output string type

```
>>> l1 = list(input('Enter list elements :')) ↵
   enter list elements : 234567
>>> l1 ↵
   ['2','3','4','5','6','7']   which is string.
```

* To make a list of integers using keyboard.

```
List = eval(input("enter list"))
print("list you entered", list)
```

O/P → Enter list [6, 7, 8]
List you entered [6,7,8]

Here eval(input()) function tries to identify type by looking at the given input. This function can also be used to evaluate & return the result of an expression given as string.

Eg: eval('5+8')

O/P → 13

Or

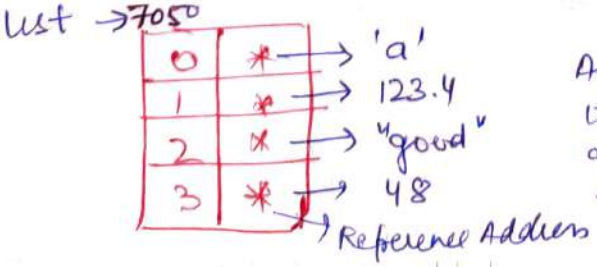
```
y = eval("3*10")
print(y)
```

O/P → 30

* Accessing LIST: List items are properly indexed in two directions - forward and backward. Values can be accessed using both indexing methods.

```
                0   1   2   3   4   → Forward indexing
list1 = ['a', 'e', 'i', 'o', 'u']
               -5  -4  -3  -2  -1   → Backward indexing
```

O/P → list1[0] = 'a' = list1[-5]
& list1[4] = 'u' = list1[-1]



As size of some list's objects are large, therefore, it store addresses of them against the index and actual value is stored somewhere else in memory.

Shaloti Roy

List and String:

Accessing list items is quite similar to string's items.

List[i] → access i^{th} item.
List[a:b] → access items from index a to b-1.

Difference of List and String:

Lists are mutable but string not.

means L[i] = <element> is valid for list but not for string.

eg: Vowels[0] = 'A'

```
>>> vowels ↵
['A', 'e', 'i', 'o', 'u']
```

Traversing List:

It means accessing and processing each elements of list.

Here FOR-loop is used to traverse.

Syntax: For <item> in <list>:
    Process each item

Also → For <index> in range(len(L)):
    Process list[index]

eg:
```
L = ['P', 'Y', 'T', 'H', 'O', 'N']
For a in L:
    print(a)
```
→ O/P →
P
Y
T
H
O
N

Comparing Lists:

Python compares two lists in lexicographical order and <, >, ==, != etc operators are used.

For eg:
```
>>> L1, L2 = [1,2,3], [1,2,3]
>>> L3 = [1, [2,3]]
>>> L1 == L2
    True
>>> L1 == L3
    False
```

corresponding indexed items are compared sequentially.

lists should be of comparable types.

```
>>> L1 < L2
    False.
>>> L1 < L3
    Error.
```

→ comparison of L1[0] < L3[0] ✓
L1[1] & L3[1] - Not comparable

[values are not comparable types.

Shakti Raj

(*) LIST operations: Joining list, Replicating list, Slicing lists.

1) Joining lists: + used, both operands should be of list Type.

```
>>> list1 = [1,3,5]
>>> list2 = [6,7,8]
>>> list1 + list2
  [1,3,5,6,7,8]
```

list1+2
list1+'a' } X

2) Repeating list: * operator used to replicate

eg: list1 = [1,3,5]

```
>>> list1 * 3
  [1,3,5,1,3,5,1,3,5]
```

3) Slicing the lists: indexes are used for slicing

slice = list[start : stop] → Not included.
└→ (last-1)

It carries items start, start+1, ..., stop-1.

eg:
```
>>> list = [10,12,14,20,22,24,30,32,34]
           0  1  2  3  4  5  6  7  8
          -9 -8 -7 -6 -5 -4 -3 -2 -1
>>> slice = list[3:-3]
>>> slice ↵
  [20,22,24]
>>> slic[1] = 28   (Modifying) Assignment operation
>>> slic ↵
  [20,28,24]
```

slic is again a list in itself.

* If slice contains limit out of boundary.

```
>>> slice[3:30]
  [20,22,24,30,32,34]      still get the o/p if limit is crossing list range.
>>> slice[-15:7]
  [10,12,14,20,22,24,30]   same o/p.
>>> slice[9:12]
  []                       Python will generate no error and gives empty sequence.
```

* Slices steps: means not consecutive items o/p.

```
>>> slice[0:10:2]   → step - take every 2nd element
  [10,14,22,30,34]
```

```
>>> slice[ : : 3]      default start & stop consider.
  [10,20,30]
```

If `>>> slice[ : : -3]`

[34, 24, 14] Reversed.

✱ Use of slices for list manipulation : / Modification

```
>>> L = ["one", "two", "three"]      (0, 1, 2)
>>> L[0:2] = [0,1]      Assigning new values to list slice.
>>> L ↵
  [0,1,"three"]
>>> L = ["one","two","three"]
>>> L[0:2] = "a"
>>> L ↵
  ["a","three"]
```

eg:

```
>>> L1 = [1,2,3]
>>> L1[2:] = "604"   ← string is also a sequence
>>> L1 ↵
  [1,2,3,'6','0','4']
```

But if we assign non-sequence values to a list slice such as a numbers, Python will give error

```
>>> L1[2:] = 345   ← is a numbers, not a sequence
```

Generate Error:

Shaikh Raj

✱ Operations on LISTs : Appending, updating, deleting etc.

1) Appending Elements to a list : Adding items to an existing sequence. Append( ) method is used to add a single item to the end of the list.

L[index] = <new value>

eg:

```
>>> list1 = [10,12,14,16]      (0, 1, 2, 3)
>>> list1[2] = 24
>>> list1 ↵
  [10,12,24,16]
```

updating elements to a list

L.append(item)

```
>>> list1 = [10,12,14]
>>> list1.append(16)
>>> list1 ↵      → [10,12,14,16]
```

2) Deleting Elements from a list:

"del" statement is used to remove an individual item, or to remove all items identified by a slice.

```
del list[index]          # to remove element at index
del list[start: stop]    # to remove elements in list slice
```

eg:

```
>>> list = [1,2,3,4,5,6,7,8]
>>> del list[3]
>>> list
[1,2,3,5,6,7,8]
>>> del list[2:4]
>>> list
[1,2,6,7,8]
>>> del list
```

All items deleted. and the list object too.

(*) pop() method is also used to remove single elements, not list slices. It also returns the value.

list.pop(index) → optional, if not mentioned then last value will be deleted.

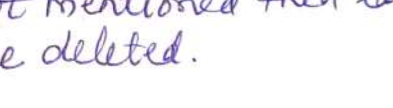
```
>>> list = [1,2,3,4,5,6]
>>> list.pop()
6
>>> list.pop(3)
4
```

Use of Pop(): when you want to store the elements being deleting for later use.

eg
```
item1 = list.pop()     = 6
item2 = list.pop(3)    = 4
```

(*) Make a copy of a list: Two methods!

1) a = [1,2,3]
   b = a   ← make copy of list



Here no new list named b will be created. only list with name 'a' has another label 'b'.

2) b = list(a)

This method is used to make a separate copy of list 'a' in memory.

Shakti Raj

(*) List Functions and methods

methods are implimented as

<listobject> . <method name> ( )

1) index method! list.index(item)

eg: L1 = [13,18,11,16,18,14]

```
>>> L1.index(18)   ← consider First occurance of item
1
>>> L1.index(19)
```

Generate error.

2) Append - method: list.append(item)

it takes exactly one item and returns no value.

```
>>> list = [1,2,3]
>>> list2 = list.append(12)
>>> list2   ← list2 doesn't return any value, hence empty
>>> list
[1,2,3,12]
```

3) Extend method: list.extend(list)

it takes exactly one element (a list type) and return no value

This method is used for adding multiple elements in the form of list.

```
>>> t1 = ['a','b','c']
>>> t2 = ['d','e']
>>> t1.extend(t2)
>>> t1 ↵
['a','b','c','d','e']
>>> t2 ↵
['d','e']
```

Special case!

```
>>> t3 = t1.extend(t2)
>>> t3
empty
```

here t3 didn't return any value, But t1 is extended with t2 list.

Append()	extend()
1) doesn't return any value	1) doesn't return any value.
2) It adds one element to a list	2) It adds multiple elements to a list in the form of list.

4) **Insert method:** list.insert(<pos>, item)

- It takes two arguments but return no value.
- Here 'pos' is the index of the element before which the second argument <item> is to be added.
- It is used to place item at any specific position in the list. unlike append() and extend()

eg:
```
>>> t1 = ['a','e','u']
>>> t1.insert(2,'i')
>>> t1
['a','e','i','u']
```

Special case:

list.insert(0,x) = insert x at index 0. (at front)

list.insert(len(t1),x) = insert x at the end of list.
= list.append(x)

If
```
t1 = ['a','e','i','u']   (-4, -3, -2, -1)
>>>t1.insert(-9,'k')
>>>t1
['k','a','e','i','u']   #here list prepended with 'k'.
```

5) **Pop-method:** list.pop(index) → optional argument.

- takes one optional argument and returns a value - the item being deleted.
- If no index is specified, then last item will be removed.
- If list is empty, then this method will raise error.

```
>>>t2=[]
>>>t2.pop()
error.
```

6) **Remove-method:** list.remove(<value>)

- takes one essential argument and doesn't return anything.
- If list is empty, then it will generate error.
- It remove the first occurence of a given item from list.

```
>>> t1 = ['a','e','i','p','q','a','q','p']
>>> t1.remove('a')
>>> t1
['e','i','p','q','a','q','p']
>>> t1.remove('p')
```

```
>>>t1
['e','i','q','a','q','p']
>>>t1.remove('k')
ERROR
```

7) **Clear method:** List.clear()

- it removes all items in the list.
- it doesn't delete the list object.

```
>>> L1 = [2,3,4,5]
>>> L1.clear()
>>> L1
[ ]
```

* del L1 deletes object also.

8) **Count-method:** List.count(item)

- it counts the 'item' in a list. and returns its value of occurance
- if item is not in list then returns 0.

```
>>> L1 = [13,18,20,10,18,23]
>>> L1.count(18)
2
>>> L1.count(28)
0
```

9) **Reverse-method:** list.reverse()

- it reverses the items of the list.
- This is done "in place" i.e. it doesn't create a new list.
- Take no argument, returns no list/anything.

```
>>> t1
['e','i','q','a','q','p']
>>> t1.reverse()
>>> t1
['p','q','a','q','i','e']
```

```
>>> t2 = [3,4,5]
>>> t3 = t2.reverse()
>>> t3   <- doesn't return anything
>>> t2
[5,4,3]
```

10) **Sort-method:** List.sort()

- sorts the items of the list, by default in increasing order.
- This is done "in place" means no new list will be created.

```
>>> t1 = ['e','i','q','a','q','p']
>>> t1.sort()
>>> t1
['a','e','i','p','q','q']
```

- it doesn't return anything.
- to sort in decreasing order >>> List.sort(reverse=True)
- Not used for complex numbers.

Shakti Raj

Question:- How are lists different from strings when both are sequences?

Ans:- 1) Lists are mutable sequences while strings are immutable.
2) In consecutive locations, strings store the individual characters while list stores the references of its elements.
3) Strings store single type of elements - all characters while lists can store elements belonging to different types.

TUPLES

(*) **Tuples:**
- Tuples are immutable (Non-modifiable)
- store values of any type in a sequence.
- it creates a fresh tuple when you make changes to an elements of a tuple.
- it is similar to string and lists but differ by its immutable property from lists only.
- it is a standard data type of Python.
- it is depicted using parenthesis i.e. ( ).

eg: ( ) - empty tuple
(1,2,3) - tuple of integers
(1, 2.5, 3.7, 9) - tuple of numbers.
('a', 'b', 'c') - tuple of character
('a', 1, 'b', 3.5, 'zero') - tuple of mixed value types.
('one', 'two', 'three') - tuple of strings.

(*) **Creating Tuples:**
- Just as lists, tuples are created by putting value is round parenthesis and separated by comma's.

```
T = ( )
T = (value, value, ...)
T = (2,4,6)
```

1) **Empty tuple:** T = tuple()
- it is equivalent to 0 or ''.

2) **Single element tuple:** This is some tricky.

```
>>> t=(1)   Python treats it as integer expression.
>>> t
  1   <- hence return integer, not tuple.
```

so to make it tuple add a comma after a single item.

```
>>> t=3,
>>> t
  (3,)
```

```
>>> t2=(4,)
>>> t2
  (4,)
```

3) Long Tuples: T = (0, 1, 2, 3, 4, 5, 6, 7, 8, ...
12, 13, 14 - - - - - - - -
31, 32, 33 - - - - - 100)

4) Nested Tuples:

t1 = (1, 2, (3, 4))

- In this, a tuple contains an element which is tuple itself.
- It has 3-elements.

(*) Creating Tuples from existing Sequences:

T = tuple(<sequence>)
↳ strings, lists, tuples.

eg: (1) >>> t1 = tuple('hello') ← created from string.
>>> t1
('h', 'e', 'l', 'l', 'o')

(2)
```
>>> L = ['w', 'e', 'l', 't', 'y']
>>> t2 = tuple(L)
>>> t2
('w', 'e', 'l', 't', 'y')
```
← t2 created from other sequence 'L'

(3) tuple generated via keyboard input.

```
t1 = tuple(input('Enter tuple elements: '))
Enter tuple elements: 234567.
```
← don't need bracket

```
>>> t1
('2', '3', '4', '5', '6', '7')
```
← return characters Not numbers.

For ~~a~~ number value return use eval().

```
t1 = eval(input("Enter tuple elements"))
Print("Tuple you entered", t1)
```

O/P → Enter tuple elements (2, 4, "a", [1, 2]) ← bracket is mandatory.
Tuple you entered (2, 4, "a", [1, 2])

eval() doesn't work in python shell. it can be used through script.

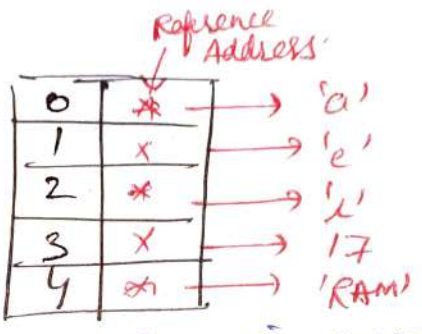
✱ Accessing Tuples:

Its items can be accessed like lists. Same index method is in tuples.

```
                -5  -4  -3  -2  -1
tuple1 = ( 'a', 'e', 'i', 'o', 'u' )
          0   1   2   3   4
```

Storage in memory

Unlike string, due to the issue of different sizes of elements, it stores reference addresses at the said indices. Actual values are somewhere else in memory.



✱ Individual Item Access:

```
>>> vowels = ('a', 'e', 'i', 'o', 'u')
>>> vowels[0]
'a'
>>> vowels[-1]
'u'
```

✱ Difference between List and Tuples.

List[i] = item ✓ it is mutable

Tuple[i] = item ✗ since it is immutable

✱ **Traversing a Tuple:** means accessing and processing each element of it. 'For' loop is generally used for it.

```
For <item> in Tuples:
    Process each item
```

eg:

```
T = ('P', 'y', 't', 'h', 'o', 'n')
For a in T:
    Print(T[a])
```

o/p →

```
P
y
t
h
o
n
```

If index is used then

```
For index in range(len(T)):
    process Tuple[index]
```

✱ Tuples operations:

1) Joining Tuples: + (concatenation operator) used.

eg:
```
>>> T1 = (1, 3, 5)
>>> T2 = (6, 7, 8)
>>> T1 + T2
(1, 3, 5, 6, 7, 8)   # join & creates new tuple also.
```

- both operands with '+' must be of tuple types. Otherwise give errors
- tuple + number
  tuple + string
  tuple + list
  tuple + complex no
  → generate errors

2) Repeat or Replicating Tuples: * used.

```
>>> T1*3
(1, 3, 5, 1, 3, 5, 1, 3, 5)
```

3) Slicing the Tuples: Seq = T[start:stop]

```
>>> T1 = (10, 12, 14, 20, 22, 24, 30, 32, 34)
    indexes: 0  1  2  3  4  5  6  7  8
    negative indexes: -9 -8 -7 -6 -5 -4 -3 -2 -1
>>> seq = T[3:-3]
>>> seq
(20, 22, 24)
```

boundary cases:

```
>>> T1 = (10, 12, 14, 20, 22, 24, 30, 32, 34)
>>> T1[3:30]
(20, 22, 24, 30, 32, 34)
>>> T1[-15:7]
(10, 12, 14, 20, 22, 24, 30)
```

Slice steps:

Seq = T[start:stop:step]

```
>>> T1[0:10:2]
(10, 14, 22, 30, 34)
```

NOTE: +, * can be used with tuples slice also.

Comparing Tuples:

- Two tuples can be compared without using any loop code.
- <, >, ==, != etc operators are used to do so.
- individual elements are compared.

eg:
```
>>> a=(2,3)
>>> b=(2,3)
>>> a==b
True
```

```
>> c=('2','3')
>> a==c
False
```

```
>> a>b
False
```

Unpacking Tuples:

packing = creating a tuple from a set of values.
unpacking = creating individual values from a tuple's elements. (Reverse of packing)

<variable 1>, <variable 2>, <variable 3>, ... = t

Variables must match the number of elements in t.

eg: If t = (1, 2, 'A', 'B'), here len(t) = 4

so to unpack,

w, x, y, z = t or (w, x, y, z) = t

If we print
print(w) → 1
print(x) → 2
print(y) → 'A'
print(z) → 'B'
} o/p.

Deleting Tuples: 'Del' statement is used.

>>> del t1[2] → error, since, it is immutable.
so individual item cannot be deleted. but complete tuple can be deleted. as

del <tuple_name>

eg:
```
>>> t1=(5,7,3,9,12)
>>> t1
(5,7,3,9,12)
>>> del t1
>>> print t1
error
```

(*) Tuples Functions and methods :

1) **len() method:** counts the elements in a tuple.

len(<tuple>) – it takes tuple name as argument & return an integer.

eg:
```
>>> employee = ('John', 1000, 24, 'Sales')
>>> len(employee)
4
```

2) **max() method:** max(<tuple>)

– takes tuple name as argument and returns the element with max. value

eg:
```
>>> t1 = (10,12,14,20,22)
>>> max(t1)
22
```

```
>>> t1 = ("karan", "zubin", "zara", "Ana")
>>> max(t1)
'Zubin'
```

NOTE → For mixed type value, this method will not work.

3) **min() method:** min(<tuple>) – return min value.

```
>>> min(t1)
10
```

```
>>> min(t1)
"Ana"
```

4) **index() method:** <tuplename>.index(<item>) – returns index no.

eg:
```
>>> t1 = [3,4,5,6.0]
>>> t1.index(5)
2
```
→ it will raise error if item doesn't exist in tuple.

5) **count() method:** <sequence name>.count(<object>) – count total occurence of elements.

eg:
```
>>> t1 = (1,1,2,3,4,5,6)
>>> t1.count(1)
2
```

6) **tuple() method:** tuple(<sequence>)

– takes an optional argument of sequence type. Returns a tuple.
– with no argument, it returns empty tuple.

a) Creating empty tuple
```
>>> tuple()
()
```

b) Creating a tuple from list.
```
>>> t = tuple([1,2,3])
>>> t
(1,2,3)
```

c) Creating a tuple from string
```
>>> t = tuple("abc")
>>> t
('a', 'b', 'c')
```

d) Creating a tuple from keys of a dictionary.
```
>>> t1 = tuple({1:"1", 2:"2"})
>>> t1
(1,2)
```

NOTE If >>> t = tuple(1) generate error. Here argument must be of sequence type like tuple(1,)

DICTIONARIES

* **Dictionaries:** Key:Value Pairs
  - Instead of using indexing method, dictionary data type uses key-value pairs.
  - They are mutable, and unordered collections.
  - It associates keys to values.
  - Curly brackets are used.

memory:

Key	Val
Key	Val
Key	Val
Key	Val

* **Creating Dictionary:**

<dict-name> = {<key>:<value>, <key>:<value> .... }

eg:
1) Teachers = {"Shakti": "Python", "Harish": "CC", "Reemi": "PHP"}
2) dict = { } # empty dictionary
3) dict = {[2,3]: "abc"} # mutable type key not allowed (List)
   ERROR.

NOTE → Dictionary are also known as associative arrays or mapping or hashes.

* **Accessing Elements of a dictionary:**
  - Key is required for accessing value.

<dict_name>[<key>]

```
>>> teachers["Shakti"]
Python
```

or, `>>> print("Shakti Sir teaches", teachers['shakti'])`

Shakti Sir teaches Python.

if, `>>> teachers["Amit"]` ↵
Error. # Not present in definition.

* **Traversing a Dictionary:**
  - Traversal of a collection means accessing and processing each element of it.
  - "For" loop is used in traversing. (mainly)

eg: `d1 = {5:"number", 'a': "string", (1,2): "tuple"}`

```
For key in d1:
    Print(key, ":", d1[key])
```

O/P →
a : string
(1,2) : tuple
5 : number

} As dict. elements are unordered so O/P may be in different order

(*) Accessing keys or values Simultaneously:

- dict.keys() → show all keys
- dict.values() → show all values.

eg: >>> teachers.keys()
dict_keys(['Shakti', 'Harish', 'Reena'])
>>> teachers.values()
dict_values(['Python', 'CC', 'PHP'])

(*) Characteristics of a Dictionary:

1) Unorder set
2) Not a sequence (like string, lists, tuple)
3) Indexed by keys, Not numbers.
4) Key must be unique.
5) Values are mutable, but not the keys.
   eg: <dict>[<keys>] = <value> # Addition is also done using it
6) Internally stored as mappings. (called hash function)

(*) Working with dictionaries:

In this we study various operations on dictionaries such as adding elements, updating and deleting elements of dictionaries.

(A) Multiple ways of creating dictionaries:

1) Initializing a dictionary: All elements are properly initialized

eg: Employee = {'name': 'John', 'salary': 10000, 'age': 24}

2) Adding Key:Value Pairs to an empty dictionary

(i) Two ways of creating empty dictionary:

① employees = { }

② employees = dict()

(ii) Add key:value pairs, one at a time.

employees['name'] = 'John'
employees['salary'] = 10000
employees['age'] = 24

3) Creating a dictionary from Name and Value pairs:

Using dict() constructor of dictionary, we can create new dict. There are multiple ways to do it.

(i) specific key:value pairs as keyword arguments to dict() function

argument, value, Assignment

```
Employee = dict(name = 'John', salary = 10000, age = 24)
```

↑ Keys are not enclosed in quotes, values are given after = sign

```
>>> Employee
{'salary': 10000, 'age': 24, 'name': 'John'}
```

(ii) Specify comma-separated key:value pairs.

```
>>> Employee = dict({'name' : 'John', 'salary': 10000, 'age': 24})
>>> Employee
{'salary': 10000, 'age': 24, 'name': 'John'}
```

(iii) Specify keys separately and corresponding values separately. Here zip() is used as follows.

```
>>> employee = dict(zip(('name', 'salary', 'age'), ('John', 10000, 24)))
>>> employee
{'salary': 10000, 'age': 24, 'name': 'John'}
```

- here keys and values are enclosed separately in parenthesis as tuples. and then given as argument to zip() function.

(iv) Specify key:value pairs separately in form of sequences.

```
>>> employee = dict([['name', 'John'], ['salary', '10000'], ['age', 24]])
>>> employee
{'salary': 10000, 'age': 24, 'name': 'John'}
```

list, list, list, list

Here, one list type agument passed to dict() constructor. This list agument, in turn contains all key:value pairs as lists.

(B) Adding Elements to dictionary:

<dictionary>[<key>] = <value>

```
>>> employee = {'name': 'John', 'salary': 1000, 'age': 24}
>>> employee['dept'] = 'sales'
>>> employee
{'salary': 10000, 'dept': 'sales', 'age': 24, 'name': 'John'}
```

⊛ Nesting dictionaries:

- We can even add dictionaries as values inside a dictionary.
- only values can be of type dictionaries. Not keys. because keys are immutable.

eg: employees = {'John': {'age': 25, 'salary': 20000}, 'Divya': {'age': 35, 'salary': 50000}}

```
For key in employees:
    Print("employee", key ':')
    Print('age:', str(employees[key]['age']))
    print('salary:', str(employees[key]['salary']))
```

O/P →
```
employee John:
    age : 25
    salary : 20000
employee Divya:
    age : 35
    salary : 50000
```

(C) Updating existing elements in a dictionary:

<dictionary>[<key>] = <value>

Only existing key's value is updated by above command. If it is not already present then it will be added as new entry.

(D) Deleting elements from a dictionary:

Two methods:

1) del - command. del <dictionary>[<key>]

```
>>> employee
{'salary': 10000, 'age': 24, 'name': 'John'}
>>> del employee['age']
>>> employee
{'salary': 10000, 'name': 'John'}
```

If `>>> del employee['RollNo']` → Error

↳ Non-existing key.

(ii) pop() method used to delete elements.

<dictionary>.pop(<key>)

- It not only delete the key:value pair for the given key but also return the corresponding value.

eg:

```
>>> employee
{'salary':10000, 'age':24, 'name':'John'}
>>> employee.pop('age')
24   ← Value shown
>>> employee
{'salary':10000, 'name':'John'}.
```

If we try to delete a 'key' which doesn't exist. then python will show error. So if a simple message is shown in this case then use —

```
>>> employee.pop('new', "Not found")
'Not found'
```

(E) Checking for existence of a key:

- 'in' and 'not in' operators are used. They can be used only for keys existence. These are not used for values if used with dict name directly.
- For values existence, reverse lookup method is used.

eg:

```
>>> employee = {'salary':10000,'age':24,'name':'John'}
>>> 'age' in employee
True
>>> 'John' in employee
False
>>> 'John' not in employee
True   ← 'John' not a key. so true.
```

for values:

```
>>> 'John' in employee.values()
True.
```

(F) Pretty Printing a dictionary: If we want change the default format of Printing then import json module in your program as—

json.dumps(<dict name>, indent = <n>)

Eg: consider employee.

```
>>> print(json.dumps(employee, indent=2))
{
  "salary" : 10000,
  "age"    : 24,
  "name"   : 'John'
}
```

indent

(4) Counting frequency of elements in a list using dictionary

left

⊛ Dictionary functions and methods:

i) Len() method: len(<dictionary>)

- It gives the length of the dictionary.
- takes dictionary name as argument and returns an integer

eg: `>>> employee={'name':'John','salary':10000,'age':24}`

```
>>> len(employee)
3   ⟶ count of key:value pairs
```

2) clear() method: <dictionary>.clear()

- it removes all items from dictionary.
- takes no argument, return no value.

eg:

```
>>> employee={'name':'John','salary':10000,'age':24}
>>> employee.clear()
>>> employee
{}
```

NOTE → del employee, will delete complete dict. as object.

(3) **get() method :** <dictionary>.get(key, [default])

- It takes 'key' as essential argument and returns corresponding value.
- If key' not present, then generate error.
- We can also use default option.

eg: >>> employee.get('name')
'John'

(4) **items() method :** <dictionary>.items()

- It takes no argument, returns a sequency of (key, value) pairs.

eg: Mylist = employee.items()
for x in Mylist:
print(x)

op → ('salary', 10000)
('age', 24)
('name', 'John')

(5) **Keys() method :** <dictionary>.keys()

- takes no argument, Returns a list sequence of keys.

eg: >>> employee.keys()
['salary', 'age', 'name']

(6) **Values() method :** <dictionary>.values()

- takes no argument, returns a list sequence of values.

eg: >>> employee.values()
[10000, 'sales', 24, 'John']

(7) **update() method :** <dictionary>.update(<other dictionary>)

- It merges key:value pairs from the new dictionary into the original dictionary, adding or replacing as needed.
- New items are added to old dict and if key already exist then it will modified by new value.

eg:
```
>>> emp1 = {'name': 'John', 'salary': 10000, 'age': 24}
>>> emp2 = {'name': 'Divya', 'salary': 54000, 'dept': 'sales'}
>>> emp1.update(emp2)
>>> emp1
{'salary': 54000, 'dept': 'sales', 'name': 'Divya', 'age': 24}
>>> emp2
{'salary': 54000, 'dept': 'sales', 'name': 'Divya'}
```

(*) **Set:** - It is an unordered collection of elements.
- it doesn't accept duplicate elements.
- It is mutable and cannot contain mutable items in it. (eg: nested lists)
- indexing is not there.
- represented using {,}.
- eg: {10, 20, 'Geek', 'zero', 't'}
- Collection of different data types.

(*) **Creating empty set:**

a = set() ✓ a={} ✗

(*) **Accessing elements:**

• a = {10, 20, "Green", "Raj", 30} ✓

since it has no index and unordered so use

print(a) ← it will print all items.

and for loop is used to access individual items.

(*) **Set creation:**

① a = {3, 4, 7, 'a'}

② a = set() # empty.

③ a = set([6, 8, 2, 5, 3,]) ← list passed. # set takes single argument so list is supplied

(*) **Deleting items:**

a.discard(2) o/p → {6, 8, 5, 3}

a.remove(3) o/p → {6, 8, 5}

If item not in set then a.discard(12) o/p → No error msg, no work

but a.remove(12) o/p → show error.

Using Pop(), we can't give index to pop(). so use as -

print(a.pop()) o/p → remove any item. (arbitrary)

use clear(), a.clear(), remove all items.

(*) **update:** for adding new item use add()

eg: a.add(567) o/p → {6, 8, 567, 2, 5, 3} → at any position

add() function is used to add single value.

(*) To add multiple Values → use update function

eg: a.update([8,4,1]) o/p → {6,8,2,3,4,5,1}

or a.update([8,4,1], {891,231}) o/p {6,8,2,3,4,51,891,231}
set.

(*) Set methods:

(1) Intersection() method.: Give the common items of two sets.

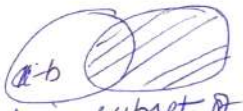
eg: I = a.intersection(b)

(2) Union() method: returns all items from set a and b. but no repetation of values.

U = a.union(b)

(3) Difference() method: returns items from a first set a and doesn't return the items in both sets.

D = a.difference(b)



(4) Issubset() method: returns true if b is subset of a, else false.

bca

S = a.issubset(b)

(5) Issuperset() method:

Su = a.issuperset(b)

eg: A = {a,e,i}
B = {i,a,e}

A ⊇ B A is superset of B
A contains all elements of B and no. of elements of A can be more or equal to B.

(*) A⊆B, A is a subset of B.
A = {1,3}, B = {1,3,9,8}
A⊆B, If set A has few or all elements equal to set B.

Function

- Function is a named unit of a group of program statements.
- This unit/subprogram can be called as many time as required in the main program.

(*) Uses of function :

1) Function divides the big program in smaller units so it makes program handling easier.
2) It avoids ambiguity.
3) It reduces program size
4) It makes program more readable and understandable.
5) It avoids rewriting same logic/code again and again.

(*) Function Creation :

syntax

```
def calcarea(x):
    A = x*x
    return A
```

fun^n name

Argument

function definition/block

body of function

statement to return computed result

```
a = int(input("enter a number"))
print(calcarea(a))
```

Not a part of function

function call inside print()

(*) Methods of passing values to functions :

eg:

```
def cube(x):
    res = x**3
    return res
```

(1) Passing literal as argument in function call

```
Cube(4)   # it would pass value as 4 to argument x
```

(2) Passing variable as argument

```
num = 10
cube(num)
```

(3) taking input while running

```
num = int(input("Enter a number"))
cube(num)
```

(4) using function call inside another statement.

```
Print(cube(3))
```

(5) Using function call inside expression

```
double of cube = 2 * Cube(3)
```

(*) Python Function Types:

1) **Built-in functions**: They are pre-defined functions in python.
   eg: len(), type(), int(), input() etc.

2) **Functions defined in modules**: These are pre-defined in a particular module and can only be used when the corresponding module is imported.
   eg: If we want to use sin() function then first we need to import "math" module.

3) **User defined functions**: They are defined/created by user for performing some special task.
   eg: cube(3)

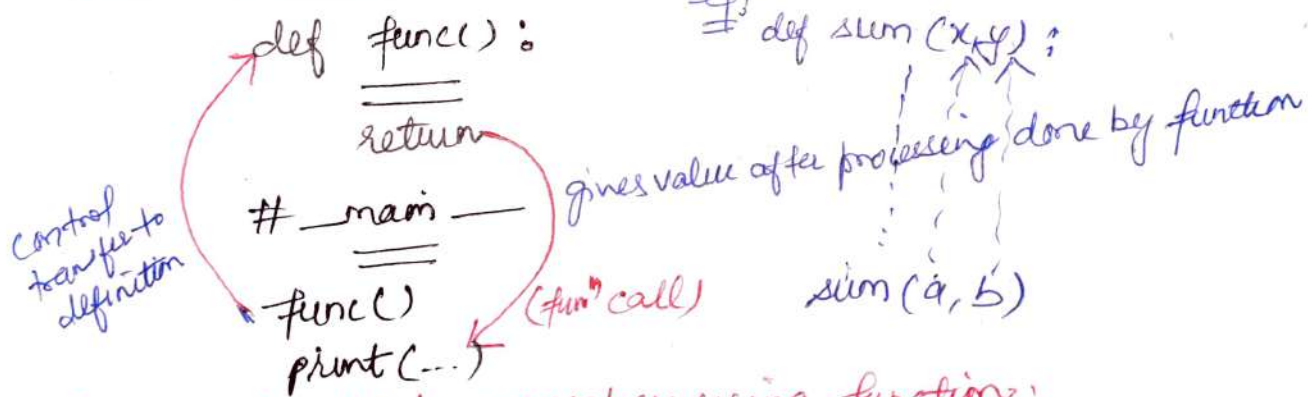
(*) Structure of a python Program:

```
def greet():
    print("Hi there!")

print("At the top-most-level")
print("Inside", __name__)
```

← Top level statement. Python will start executing program from this point downwards.

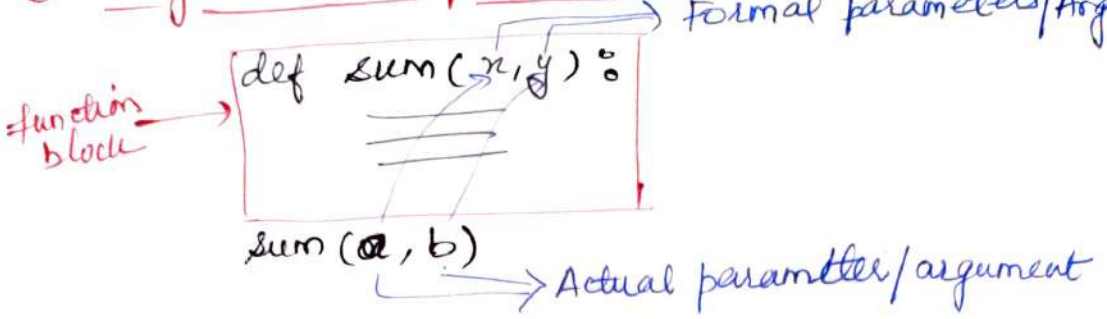
(*) Flow of execution in a function call



(*) Program to add two numbers using function:

```
def sum(x,y):
    s = x+y
    return s
num1 = float(input("Enter first no."))
num2 = float(input("Enter second no"))
sum1 = sum(num1+num2)
print("sum of two given numbers is", sum)
```

✱ Arguments and parameters:



✱ Special case:

① `def sum(x+1, b):` will generate error. } function definition

but `sum(a+1, b)` is correct. } calling of function.

② If passing values are immutable types (eg. numbers, strings, etc) then called function cannot alter the values of passed arguments. (call by value)

③ If passing values are mutable types (eg. lists, dict) then called function would be able to make changes in them. (call by reference)

call by value: the called function makes a separate copy of passed values and then work with them. So Original values remain unchanged.

Call by reference: the called function works with original values passed to it, thus any changes made, takes place in original values only.

✱ Passing parameters:

If function header has 3-parameters then function call will also pass three values.

Three types of formal arguments—

1. Positional argument (Required arguments)
2. Default argument
3. Keyword (or named) arguments

① **Positional argument:** It means values are passed to called function position-wise and Order-wise.

eg: `def check(a, b, c):`

```
check(x, y, z)      # 3 values (all variables) passed
check(2, x, z)      # 3 values (literal + var) "
```

check(2,5,7) # 3 values (all literals) passed.

- Also no of arguments in calling function should be equal to called function.

2) Default Arguments: These are default values which are already used in function definition and they need not be passed from calling function.

For eg: def interest(principal, time, rate=0.10): → called fun^n

interest(5400, 2) → calling fun^n

interest(5400, 2, 0.15) rate value needs not be supplied by this function.

- Here, rate of interest is fixed.

If value supplied then it will update the parameter in called function.

3) Keword (Named) Arguments:

In this, in calling function, Arguments are properly named matching with the formal arguments of called function. Also, if order of Actual arguments change, it will not affect the passing process. Values will passes in corresponding variables

eg: def interest(princ, time, rate):

interest(princ=2000, time=2, rate=0.10)
interest(princ=2000, rate=0.20, time=3)
interest(time=5, princ=4000, rate=0.30)

(*) multiple argument type:

interest(5000, time=5) # both positional and keyword Arg.

Rules for combining all above three types of arguments:

In function call statement.

1) positional Argument must be at first position and can be followed by any keyword
2) keyword arguments should be taken from the required arguments preferably.

3) Can not specify the value for an argument more than once.

For eg: def interest(princ, cc, time=2, rate=0.09):
return princ * time * rate

interest(princ=3000, cc=5) ✓ non-default values provided as named argument.

interest(rate=0.12, princ=5000, cc=4) ✓ keyword arguments can be used in any order and for 'time', default value is used.

interest(cc=4, rate=0.12, princ=5000) ✓ with keyword arg. we can give values in any order.

interest(5000, 3, rate=0.05) ✓ positional arg before keyword Arg.

interest(rate=0.05, 5000, 3) ✗ keyword arg. before positional

interest(5000, princ=300, cc=2) ✗ Multiple values for princ

interest(5000, principal=300, cc=2) ✗ undefined name.

interest(500, time=2, rate=005) ✗ Required argument cc is missing

In above fun". definition, princ and cc are positional and required arguments so they cannot be skipped in calling function.

Should Read

(*) Returning values from function:

Based on this, function is of two types —
1) Non-void function — return some value
2) void function — return no value.

1) Non-void function:—

function returns some computed result in terms of a value.

syntax:— return <value>

Here, value can be literal, variable, expression

eg:
```
return 5
return 5+4
return a
return a**2
return (a+8**2)/b
return a+b/c
```

2) **Void function:** A void function may or may not have a "return" statement. It does some action but doesn't return any result.

syntax: → return

eg:
```
def quote():
    print("Goodness Counts!")
    return
```
(A)

```
def greet():
    print("Hello")
```
(B)

(B) returns legal empty value of python. i.e. "None".

o/p → Goodness Counts!

op → None

eg2

code-1
```
def replicate():
    print("Hello")
print(replicate())
```
o/p
Hello
None

code2
```
def replicate1():
    return "hello"
print(replicate1())
```
hello

* **Returning Multiple Values:**

① Return statement uses multiple values/variables/expressions

eg: return <value1>, <value2>.....

② Function call statement should receive or use the returned values in one of the following ways-

a) in form of tuples

eg:
```
def square(x,y,z):
    return x*x, y*y, z*z
t = square(2,3,4)
print(t)
```
op → (4,9,16) → tuple

b) unpacked the received values of tuple

```
v1, v2, v3 = square(2,3,4)
print("The returned values are")
print(v1, v2, v3)
```

✱ Composition of arguments:

It means using an expression as part of a larger expression. In composition, arguments of a function call can be any kind of expression as given below:

1) An arithmetic expression:
   eg: greater((4+5), (3+4))

2) A logical expression.
   eg: test(a or b)

3) A function call
   eg: int(str(52)), int(float("52.5")*2)
   int(str(52) + str(10))

✱ Scope of Variables:

Part(s) of program within which a name is legal and accessible, is called scope of the variable.

Two kinds of scopes:

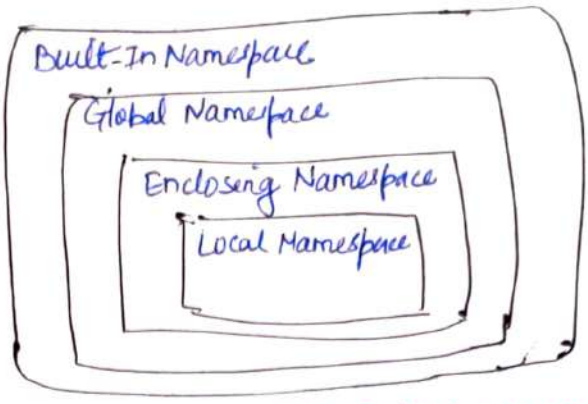
1) **Global Scope:** A variable declared in top level segment (_main_) of a program is said to have a global scope and is usable inside the whole program and all blocks contained within the program.

2) **Local Scope:** A variable declared in a function body is said to have local scope. i.e. it can be used only within this function and the other blocks contained under it.
   eg: The names/variables of formal arguments have local scope

eg:
```
x=5  <- global scope.
def func(a):
    b=a+1
    return b
y=input("Enter number")
z=y+func(x)
print(z)
```

Name Resolution: Resolving scope of a variable

Since, variables/Name are used at various levels in a program. So to identify the scope of variable or the value which it assumes to be received, is resolved using LEGB Rule-



LEGB Rule for name resolution (steps)

I) It checks within its local environment (LEGB) or local name space, If it has a variable with same name; if yes, python uses its value. If No, then go to step (ii)

II) Python now checks the enclosing environment (LEGB). If no then moves to (iii).

iii) Python now checks the Global environment (LEGB), If no, then go up

(N) Python now checks the built-in environment (LEGB) that contains all built in variables and functions of python, if it gets here then uses it value otherwise report "error".

eg: ① Here variable is a global scope, not in a local scope.

```
def calcsum(x,y):
    s = x+y
    print(num1)
    return s
num1 = int(input("Enter first no."))
num2 = int(input("Enter second no"))
print("sum is", calcsum(num1, num2))
```

→ global variable (Not declared locally within calcsum())

② Variable neither in a local scope nor in global scope.

```
def greet():
    print("Hello", name)
greet()
```

→ name is neither a local nor global, so it return error.

③ Variable in local as well as in global scope.

```
def state1():
    tiger = 15
    print(tiger)
tiger = 95
print tiger
state1() print(tiger)
```

Local

o/p → 95
15
95

(4) Use of global variable inside local scope.

To do it, use – global <Variable name>

eg:
```
def state1():
    global tiger      → it makes use of global variable
    tiger = 15
    print(tiger)
tiger = 95
print(tiger)          O/P → 95
state1()                    15
print(tiger)                15
```

Note: Once a variable is declared global in a function, you can't undo the statement. It means after a global statement, the function will always refer to the global variable and local variable cannot be created of the same name.

For good programming practice, use of global statement is always discouraged.

(*) Functions are first class objects:

First class objects in a language are handled uniformly throughout. They are treated as objects so they may be stored in data structures, passed as arguments, or used in control structures.

properties of first class functions:

1) A fun^n is an instance of object type.
2) fun^n can be stored in a variable
3) fun^n can be passed as a parameter to another function.
4) You can return the function from function
5) You can store them in data structures such as hash tables, lists etc.

(*) Map() function: It returns a list of the results after applying the given function to each item of the given iterable (list, tuple etc). syntax → map(fun, iter)

fun: It is a function to which map passes each element of given iterable (sequence)

iter: It is a iterable which is to be mapped

NOTE One or more iterable to the map() function can be passed

eg: # program to return double of n

```
def addition(n)
    return n+n

numbers = (1,2,3,4)
result = map(addition, numbers)
print(list(result))
```

o/p → [2, 4, 6, 8]

* filter() :— It filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

filter(function, sequence)
↳ tests each element of a sequence
↳ (sets, lists, tuples)

eg: # function that filters vowels.

```
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
filtered = filter(fun, sequence)
print('The filtered letter are:')
for s in filtered:
    print(s)
```

o/p → e
e

Shalett Raj

(*) **Lambda()** : Lambda function can take any number of arguments, but can only have one expression.

syntax — lambda Arguments: expression

Here expression is executed and result is returned.

eg: (1)
```
x = lambda a : a + 10
print( x(5))
```
o/p → 15

(2)
```
x = lambda a, b : a * b
print ( x(5,6))
```
o/p → 30

(*) Why do we use lambda functions ?

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

eg:
```
def myfun(n) :
    return lambda a : a * n
```
here, it takes one argument and it will be multiplied with an unknown number.
```
mydoubles = myfun(2)
print(mydoubles(11))
```
o/p → 22

(*) **Inner functions**: (Nested function) (encapsulation)

A function which is defined inside another function is known as inner function or nested function.

Nested function are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function. This process is also known as **encapsulation**.

eg:
```
def outerfun(text):
    text = text
    def innerfun():
        print(text)
    innerfun()
if __name__ == '__main__':
    outerfun('Hey!')
```
o/p → Hey!

o/p → No o/p

(*) **closures** : closures is basically keeping a record of a function within an enclosing environment. It remembers values even if they are not present in memory.

why closures :

1) closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables
2) When we have few functions in our code, closures prove to be an efficient way. But if we need to have many functions, then go for class (OOP).

eg:

```
def f1(a):
    def f2(b):
        return a+b
    return f2
a = f1(1)
b = f1(100)
print(a(2))
print(b(2))
```

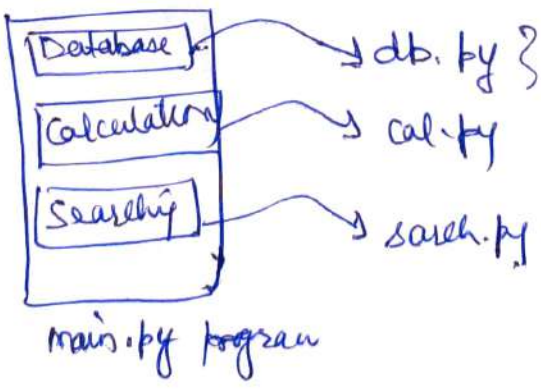
O/P → 3
102

Modules

MT

* **Module:** It is a python program file which contains a python code including python functions, class, or variables etc. This file is to be saved as ".py" extension. This is a runnable code file and can be imported in other python program.
  eg: mymodule.py
* **Library:** This is a related term with module. A library refers to a collection of modules that together caters to specific type of needs or applications.
  eg: Numpy library caters to scientific computing needs.
* **Types of Modules:** They are executed only for the first time the module name is encountered in an import statement. They are further divided into two types:
  1) User-defined Modules - ~~addcastes~~ add.py.
  2) Built-in Modules. - eg: array, math, numpy, sys,

* **When and why use module:**
  (1) When we want to make a big project, then it will be very difficult to manage all logic within one single file so if you want to separate your similar logic to a separate file, you can use module.
  (2) It will not only separate your logics but also help you to debug your code easily as you know which logic is defined in which module.
  (3) When a module is developed, it can be reused in any program that needs that module.



These can be imported in main.py as well as in other python program also.

(*) Creating a module: consider previous diagram.

How: calculation.py → module.

```
def add(a,b) :        → function
    return(a+b
def sub(a,b):
    return(a-b)
```

(*) How to use/import module:

"import" statement is used to do it.

Syntax: (1) import module_name. example: import calculation

(2) import mod_name as alias_name import calculation as cal

(3) from module_name import var1, var2, .. varn

(4) from module_name import class1, class2, .. classn

(5) from mod_name import method1, method2, tuple, list

(6) from mod_name import fun_name as alias_fun_name { Renaming a module

(7) from mod_name import * } it will import all fun's methods & objects at a time.

Note: modules can import other modules.

(*) How to access methods, functions, Variables and Classes of a module:

Using the module name, you can access the functions.

Syntax: module_name.function_name()

Eg: cal.add(10,20)
cal.sub(20,10)

Also. add = cal.add # for shortning of name.

use add(10,20)

Shalini Roy

(*) Renaming of a Module:

For renaming - import module_name as alias_name command is used. Now no need to write the complete name of module, only use its nick name defined in alias.

eg: import cal as c.

Now using alias name 'c' we can access all methods, variables and classes, fun^n's.

eg: c.add(10,20)
c.sub(30,10)

* We can also rename the member of module while accessing

eg: from cal import add as s

eg: s(10,20)

(*) import * method:

If we use: from module_name import *
this imports all names except those beginning with an underscore(_)

(*) Library and its types:

It is a collection of modules (and packages). Commonly used libraries are:

1) Python standard library: It is distributed with python that contains modules for various types of functionalities.
   eg: 1) math module: used for different types of calculations
   2) cmath module: used for maths fun^n for complex numbers
   3) random module: used for generating random-number
   4) statistics module: used for maths statistics functions
   5) Urllib module: used for URL handling fun^n so that you can access websites from within your program
2) Numpy library: it provides some advance math functionalities along with tools to create and manipulate numeric arrays.
3) Scipy library: it is used in algorithmic and mathematical tools for scientific calculations.
4) tkinter library: it is used to create userfriendly GUI interfaces for different types of applications.

5. Matplotlib library: it is used to produce quality output in variety of formats such as plots, charts, graphs etc.

(*) Module Search Path:

when a module named cal is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named cal.py in a list of directories given by the variable sys.path.
sys.path is initialized from these locations:

1) means first searches current directory.
2) If not found then searches each directory in the shell variable PYTHONPATH.
3) If not found then searches installation-dependent default path.

PYTHONPATH is a list of directory names, with the same syntax as the shell variable PATH. Means it searches the foldner named Lib where python all files are residing.

(*) Standard Module – SYS:

It provides functions and variables used to manipulate different parts of the Python runtime environment.

eg: (1) sys.exit → this causes the script to exit back to either the Python console or the command prompt.

(2) import sys.
print(sys.version) → O/P → 3.6.9 (default, Oct 8 2020, 12:12:24

(3) sys.maxsize → returns the largest integer a variable can take.

(4) sys.path → it searches path for all Python modules.

Standard Modules - math

It includes all mathematical functions like trigonometric representation function, logarithmic functions, angle conversion fun, etc. In addition, two mathematical constants are also defined in this module.

Eg: (1)
```
>>> import math
>>> math.pi
3.141592653589793.
```

(2)
```
>>> math.e  Euler's No.
2.718281828459045-
```

(3) Math Module presents 2-angle conversion functions:

1) degree( ) - converts radian to degree.
2) radian( ) - converts degree to radian

```
>>> math.radians(30)
0.5235987755982988
>>> math.degrees(math.pi/6)
29.999999999999
```

```
>>> math.sin(0.5235987755982988)
0.4999999999999999-4
```

```
>>> math.log(10)
2.302585092994046
>>> math.log10()
1.0
>>> math.e**10
22026.465794806703
>>> math.pow(2,4)
16.0
>>> math.sqrt(100)
10.0
>>> math.ceil(4.5867)
5
```

```
>>> math.floor(4.5687)
4.
```

✱ Standard module - time

It allows us to handle various operations regarding time, its conversions and representations, which finds its use in various applications in life.

The begining of time is started measuring from 1 Jan, 12.00 am, 1970 and this very time is termed as "epoch" in python.

eg: ① time() → it counts the number of seconds elapsed since the epoch.

② gmtime() → it returns a structure with 9 values each representing a time attribute in sequence. It converts seconds into time attributes (days, years, months etc) till specified seconds from epoch.

The structure attribute table is given below.

Index	Attributes	Values
0	tm_year	2008
1	tm_month	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4.	tm_min	0 to 59
5.	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6.	tm_wday	0 to 6
7.	tm_yday	1 to 366
8.	tm_isdst	-1, 0, 1 where -1 means library determines DST

Shakti Roy

eg!
```
import time
Print("seconds elapsed since the epoch are:", end=" ")
Print(time.time())
Print("Time calculated acc. to given seconds is:")
print(time.gmtime())
```

O/P :→ Seconds elapsed since the epoch are: 1470121951.953689
Time calculated acc. to given seconds is:
time.struct_time(tm_year=2016, tm_mon=8, tm_mday=2, tm_hour=7, tm_min=12, tm_sec=31, tm_wday=1, tm_yday=215, tm_isdst=0)

③ asctime("time"):→ it takes a time attributed string produced by gmtime() and return a 24 character string denoting time.

④ ctime(sec) :→ It returns a 24 character time string but takes seconds as arguments and computes time till mentioned seconds. If no argument is passed, time is calculated till present.

eg: import time
ti = time.gmtime()
print("Time calculated using asctime() is:", end="")
print(time.asctime(ti))
print("Time calculated using ctime() is:", end="")
print(time.ctime())

o/p → Time calculated using asctime() is: The Aug 2 07:47:02 2016
Time calculated using ctime() is: The Aug 2 07:47:02 2016

⑤ sleep(sec):→ This method is used to halt the program execution for the time specified in the arguments

eg: import time
print("start execution:", end="")
print(time.ctime())
time.sleep(4)
print("stop execution:", end="")
print(time.ctime())

o/p:→ start execution: The Aug 2 07:59:03 2016
stop execution: The Aug 2 07:59:07 2016

(*) Datetime Module:

It creates the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import datetime module into the python source code.

eg: import datetime
print(datetime.datetime.now())

O/P:→ 2018-12-18.16:16:45.462778

⊛ Calender Module:

It provides a calender object that contains various methods to work with the calenders.

eg: import calender
cal = calender.month(2018, 12)
print(cal)

O/P:→ December 2018
full month

② Using Dir(): It returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

eg: import json
list = dir(json)
print(list)

O/P:→ ['JSONDecoder', 'JSONEncoder', '__all__', '__author__', _
]

Shalini Rao

Exceptions

An exception is a Python object that represents an error. It is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

When a python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

⊛ Handling an exception:

If we have some suspicious code that may raise an exception, we can defend our program by placing the suspicious code in a TRY: block. After the try: block, include an EXCEPT: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:

```
Try:
    ____
    ____
Except exception1:
    Block-1
except exception2:
    Block-2
else:
    Block-3
```

⊛ Important points:

1. A single 'TRY' stmt can have multiple except statements.
2. We can provide a generic except clause, which handles any exception.
3. After except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block doesn't raise an exception.
4. The else-block is a good place for code that does not need the try:block's protection.

Eg:

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling")
except IOError:
    Print "Error: can't find file or read data"
else:
    print "written content in the file successfully"
    fh.close()
```

O/P :→ written content in the file successfully.

In this example, control is trying to open a file where you do not have write permission, so it raises an exception.

(*) Except-clause with Multiple Exceptions:

We can also use the same except statement to handle multiple exceptions.

```
Try:
    Block
except (exception1[, Exception2[, ... ExceptionN]]):
    If there is any exceptions from the above list then execute this block
    Block.
else:
    If no exception then execute this Block
```

(*) eg:

```
try:
    a=10/0;
except ArithmeticError, StandardError:
    print "Arithmetic exception"
else:
    print "Successfully Done"
```

O/P :→ Arithmetic Exception

(*) Try-finally clause:

In this case, the finally block code will must execute whether the try block raised an exception or not.

```
try:
    Block.
Finally:
    Block (definitely be executed)
```

Note: Here, Else clause can not be used.

eg:

```
try:
    fh=open("testfile", "w")
    fh.write("This is my best file")
finally:
    print "Error: can't find file or read data"
```

O/P → Can't find file or read data

✱ Argument of an Exception:

An exception can have an argument, which is a value that gives additional information about the problem.

Syntax:
```
try:
    Block
except ExceptionType, Argument:
    Print value of Argument here
```

eg:
```
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "The argument doesn't contain numbers\n", Argument)
temp_convert("xyz")
```

O/P:- The argument doesn't contain numbers
invalid literal for int() with base 10: 'xyz'

✱ Raising an Exception:

using this statement, we can raise exceptions.

Syntax:
```
raise [exception [, args [, traceback]]]
```

Here exception is the type of exception and argument is a value for the exception argument. It is optional, if not supplied, the exception argument is None.

Traceback argument is also optional and rarely used.

eg:
```
def functionName(level):
    if level < 1:
        raise "Invalid level", level
```

✱ User-defined Exceptions:

Python allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here a class is created that is subclassed from RuntimeError.

Shalini Raj Singh

(EG)

eg: Class NetworkError(RuntimeError):
    def __init__(self, arg):
        self.args = arg

try:
    raise NetworkError("Bad hostname")
except NetworkError, e:
    print e.args.

(*) Common exceptions:

A list of common exceptions that can be thrown from a normal python program is given below:

1) Zero division error = 10/0
2) Name error = It occurs when a name is not found. It may be local or global.
3) IndentationError = If incorrect indentation is given.
4) IOError = It occurs when Input Output operation fails.
5) EOFError = It occurs when end of file is reached, and yet operations are being performed.

(*) Assert statement: It is used to continue the execution if the given condition evaluates to True. If assert condition evaluates to false, then it raises the AssertionError exception with the specified error message.

Syntax: assert condition [, Error message]

eg: x=10
assert x>0, 'only positive numbers are allowed' (True)
print('x is a positive number')

o/p → x is a positive number

If x=0
assert x>0, 'only positive numbers are allowed' (False)
print('x is a positive number')

o/p → AssertionError ↵, it doesn't execute print stmt.

eg: def square(x):
    assert x>=0, 'only positive numbers are allowed'
    return x*x
try: square(-2)
except AssertionError as msg:
    print(msg)

o/p → only positive numbers are allowed.

Shalini Raj Singh

(Input and Output)(File Handling)

(*) **Needs of File Handling:** Sometimes, data becomes very large and it is not possible to display all data on console and further the memory is volatile and if data was deleted then it is again difficult to recover. So, we need to store such large data in a file and can be accessed any time. This is the need of File handling.

(*) **File:** A file is a bunch of bytes stored on some storage device.

(*) **File object:** File object is also known as file handle. A file object is a reference to a file on disk. It opens and makes it available for a number of different tasks.

File objects are very important in python as they are used to read and write data to a file on disk. All the fun that you perform on a data file are performed through file obj.

(*) **opening a file:** open() function is used.

Syntax: `File object = open(filename (path), access mode, buffering)`

Various modes are:

SNo	Access mode	(File opens in) Description	Note. (Short form) (FPSB)
01	r	Read only	File pointer (ptr) sets at beginning. This is the default mode
2.	rb	Read only (binary mode)	FPSB
3.	r+	Read & write	FPSB
4.	rb+	Read & write (binary mode)	FPSB
5.	w	write only	FPSB, it overwrites the file if previously exists or creates a new one if no file exists with the same name.

(File opens in)

SoNo	Access Mode	Description	Note
6.	wb	write only (binary)	FPSB, overwrite previous file or create new one,
7.	w+	write & read	FPSB, it is different from r+ in the sense that it overwrites the previous file if it exists whereas r+ doesn't.
8.	wb+	write & read (binary mode)	FPSB
9.	a	Append Mode	FPSE (File pointer exists at the end of the previous written file if exists any It creates new file if no file exists with the same name.
10.	ab	Append (binary Mode)	FPSE, same above.
11.	a+	Append & read	FPSE (if file exist), same above
12.	ab+	Append & read (binary)	FPSE

Shakti Raj Singh

eg:
```
f = open("file.txt","r")
if f:
    print("file is opened successfully")
```

o/p :→ <class '_io.TextIOWrapper'>
file is opened successfully.

close( ) method

It is used to close the opened file. for good practice it is necessary.

Syntax: fileobject.close( )

eg:
```
f = open("file.txt","r")
if f:
    print("file is opened successfully")
f.close()
```

✱ Read the file:

read() is used to do it.

It reads a string from the file. It can read the data in the text as well as binary format.

Syntax:

```
fileobj.read(<count>)
```

→ is the no. of bytes to be read from the file starting from the beginning of the file to the count value.

If count is not specified, then it may read the content of the file until the end.

eg:

```
f = open("file.txt", "r")
content = f.read(9)
print(type(content))
print(content)
f.close()
```

O/P → <class 'str'>
Hi, I am
↳ only 9 places

✱ Read lines of the file:

Readline() is used.

It reads the file line by line and starts from begining of the line. i.e. if we use it two times, then first two lines of the file will be displayed.

eg:

```
f = open("file.txt", "r")
content = f.readline()
print content)
f.close()
```

O/P:→ Hi, I am the file and being used as.

✱ Looping through the file:

By looping through the lines of the file, we can read the whole file.

eg:

```
f = open("file.txt", "r")
for i in f:
    print(i)
```

→ (O/P) →

Hi, I am the file and being used as
an example to read a
file in python.

write() is used.

* **Writing the file:** This mode is used to write some text to a file. File is to be opened in this mode using "a", "w".

eg: (1) `f = open("file.txt", "a")`
`f.write("Python is the new language")`
`f.close()`

o/p :→ Hi, I am the file and being used as an example to read a file in python.
Python is the new language.

(2) If "w" is used, then o/p is — Python is the new language.

* **Creating a new file:**

To create a new file "x" mode is used with open().
It creates a newfile with the given name and generate error if the same name already exists.

eg:
```
f = open("file2.txt", "x")
print(f)
If f:
    print("File created successfully")
```

o/p :→ File created successfully.

* **With - statement in files:**

This statement is used for manipulation of files purposes.
The advantage of this stmt is that it provides the guarantee to close the file regardless of how the nested block exits.
It is used in the case where break, return, exception occurs in the nested block of code. It then, automatically, closes the file. It doesn't let the file to be corrupted.

eg:
```
with open("file.txt", 'r') as f:
    content = f.read()
    print(content)
```

o/p → Python is the new language.

Shakti Raj Singh

⊛ File pointer Position: tell()

This method tells the byte number at which the file pointer exists.

eg: `f = open("file2.txt", "r")` # initially the ptr at 0
`print("The filepointer is at byte", f.tell())`
`content = f.read()`
`print("After reading, the filepointer is at:", f.tell())`

O/P:→ The filepointer is at byte 0
After reading, the filepointer is at 26.

⊛ modify file pointer position: seek()

This method is used to modify the file pointer position externally. This is requie when we want to read the file from a particular position.

Syntax: filepointer.seek(offset, from)

offset:→ refers new position of the file pointer within the file.
from:→ it indicates the reference position from where the bytes are to be moved.
- If it is set to 0, the beginig of the file is used as reference position.
- If it is set to 1, the current position of the file pointer is used as the reference position.
- If it is set to 2, the end of the file pointer is used as the reference position.

eg:
```
f=open("file2.txt","r")
print("The ptr is at byte", f.tell())
f.seek(10)
print("After reading, the file ptr is at", f.tell())
```

O/P:→ The fileptr is at 0
After reading, the fileptr is at 10.

CLASSES

Python is an object-oriented programming language. So we can easily create classes and objects.

Major principles of OOP system are -

1. object
2. class
3. Method
4. Inheritance
5. Polymorphism
6. Data Abstraction
7. Encapsulation.

1) **object**: object is an entity that has state and behaviours. It may be similar to any real-world object like mouse, pen, chair etc.

Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute

Shalabh Roy.

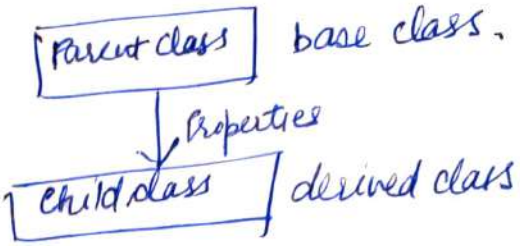
2) **class**: The class can be defined as collection of objects. It is a logical entity that has some specific attributes and methods.

eg: if we have a class - employee then it has attributes and methods i.e. an email-Id, age, salary etc

Syntax: class classname:

3) **Method**: method is a function that is associated with an objects In Python, a method is not unique to class instances. Any object type can have methods.

④ **Inheritance:** It is the most important aspect of OOP which simulates the real world concept of inheritance. In this, every class inhibits the properties and behaviours of ~~another~~ its parent class. It gives re-usability of code.



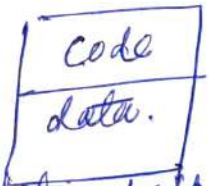
⑤ **Polymorphism:** Poly + morphs
(many) (forms/shapes)

It means one task can be performed in different ways.

for eg: ① let class → animal, and all animals speak. but they speak differently. Here, speak behavior is polymorphic in the sense and depends on the animal.

② A person at the same time can have different characteristics like a man at the same time is a father, a husband, an employee. so the same person posses different behaviour in different situations. This is called polymorphism.

⑥ **Encapsulation:** It restricts the access methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.



⑦ **Data Abstraction:** Data abstraction and encapsulation both are often used as synonyms. Data abstraction is achieved by encapsulation. Abstraction is used to hide internal details and show only functionalities.

eg: like power switch board. - it functions but hide all circuitaries inside it.

(*)

Object-oriented Programming	Procedural Programing
1. It is a problem solving approach and used where computation is done by using objects.	1) It uses a list of instructions to do computation step-by-step.
2. It makes the development and maintenance easier.	2) It is not easy to maintain the codes when the project becomes lengthy.

Shaveta Rao

OOP	PP
3) It simulates the real world entity. So real world problems can be easily solved through oops.	3) It doesn't simulate the real world It works on step by step instructions divided into small parts called functions.
4) It provides data hiding. so it is more secure. Private data cannot be accessed.	4) It doesn't provide any proper way for data binding. So it is less secure.
5) Examples - C++, Java, .Net, Python, C# etc	5) eg: C, Fortran, Pascal, VB etc

Reality Exp.

✱ Python class and objects:

Since class is a virtual entity and it can be seen as a blueprint of an object. The class came into existence when it instantiated.

eg: Let a class is a prototype of a building. A building contains all the details about floor, doors, windows etc. Now, we can make as many buildings as we want, based on these details.

Hence, the building can be seen as a class, and we can create as many objects of this class.

Hence, the object is the instance of a class. and the process of creating an object can be called as instantiation.

✱ Creating Classes: class keyword is used.

Syntax: class className:

- Each class is associated with a documentation string which can be accessed by using className.__doc__
- Attributes are the variables that belong to a class
- Attributes are always public and can be accessed using the dot (.) operator.

eg:

```
class employee:
    id=10
    name="ayush"
    def display(self):
        print(self.id, self.name)
```

self is used as a reference variable which refers to the current class object. It is always the first argument in the fun? definition. However, it is optional in function call.

⊛ Creating an instance of class:

A class can be instantiated by calling the class using the class name.

Syntax: object_name = classname(arguments)

eg:
```
class employee:
    id=10
    name="Ravi"
    def display(self):
        print("ID:%d \nName:%s"%(self.id, self.name))
emp=employee()
emp.display()
```

O/P :→
```
ID:10
Name:Ravi
```

⊛ Python Constructor:

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

They are of two types –
1) Parameterized
2) Non-parameterized.

constructor definition is executed when we create the objects of this class.

⊛ Creating the constructor in python:

the method __init__ simulates the constructor of the class. This method is called when the class is instantiated.

We can pass any numbers of arguments at the time of creating the class object, depending upon __init__ definition. The class attributes initialization is very important. Every class must have a constructor, even if it simply relies on the default constructor.

eg: To initialize the employee class attributes –

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name
    def display(self):
        print("ID: %d \nName: %s" %(self.id, self.name))
emp1 = Employee("Ravi", 101)
emp2 = Employee("Harshit", 102)
emp1.display()    # to print emp1 inf.
emp2.display()
```

O/P:-

```
ID: 101
Name: Ravi
ID : 102
Name: Harshit.
```

(2) Counting the number of objects of a class.

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1 = Student()
s2 = Student()
s3 = Student()
print("The number of students", Student.count)
```

O/P:→ The number of students : 3

* Python Non-parameterized Constructor Example:

```
class student:    # constructor-non parameterised
    def __init__(self):
        print("This is non-parameterized constructor")
    def show(self, name)
        print("Hello", name)
student = Student()
Student.show("Ravi")
```

→ O/P → This is non-parameterised constructor
Hello Ravi

⊛ Parameterized constructor example:

```
class Student:
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello", self.name)
student = Student("Ravi")
student.show()
```

O/P:→ This is parametrized constructor
Hello Ravi

⊛ Python In-built class function:

S.No.	Function	Description
1.	getattr(obj, name, default)	– It is used to access the attribute of the object.
2.	setattr(obj, name, value)	– It is used to set a particular value to the specific attribute of an object.
3.	delattr(obj, name)	– It is used to delete a specific attrib
4	hasattr(obj, name)	– It returns true if the object contains some specific attribute.

Example:

```
class Student:
    def __init__(self, name, id, age):
        self.name = name
        self.id = id
        self.age = age
s = Student("Ravi", 101, 22)   # Here creates the object of the class Student
print(getattr(s, 'name'))      # prints the attri. name of the ob -s
setattr(s, 'age', 23)          # reset the value of attri age = 23
print(getattr(s, 'age'))       # prints Ture if the student contain the attribute with name id
print(hasattr(s, 'id'))
delattr(s, 'age')              # delete the attribute age
print(s.age)                   # generate error, as attribute age has been deleted
```

Ravi ←
23 ←
True ←
Error ←

O/P→ Ravi
23
True
AttributeError: 'Student' object has no attribute 'age'.

Shakti Roy

⊛ Built-In Class Attributes:

Alongwith other attributes, a python class also contains some built-in class attributes which provide information about the class.

So. No	Attribute	Description
01	__dict__	It provides the dictionary containing the information about the class namespace.
02	__doc__	It contains a string which has the class documentation.
03.	__name__	It is used to access the class name.
04.	__module__	It is used to access the module in which this class is defined
05.	__bases__	It contains a tuple including all base classes.

example:

```
class Student:
    def __init__(self, name, id, age)
        self.name = name
        self.id = id
        self.age = age
    def display_details(self)
        print("Name: %s, ID: %d, age: %d" %(self.name, self.id)
```

```
s = student("Ravi", 101, 22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

O/P:

```
None
{'name': 'Ravi', 'id': 101, 'age': 22}
__main__
```

* **Python Inheritance:** A derived class can inherit base class by just mentioning the base in the bracket after the derived class name.

Syntax:
```
class Derivedclass(Base class):
    <class suite>
```

Also,
```
class Derivedclass(Base class1, base class2, ..... baseclassN)
    <class suite>
```

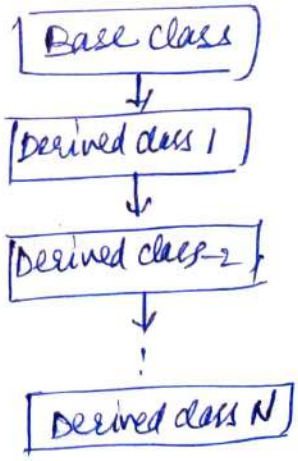
Example:

```
class Animal:
    def speak(self):
        print("Animal speaking")
    class Dog(Animal):        # child class Dog inherits the base class Animal
        def bark(self):
            Print("dog barking")
d = Dog()
d.bark()
d.speak()
```

O/P:→
```
dog barking
Animal speaking
```

* **Python Multi-level Inheritance:** This is also possible in Python. It is achieved when a derived class inherits another derived class. and there is no limit to such number of levels.

Syntax:
```
class class1:
    <class suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```



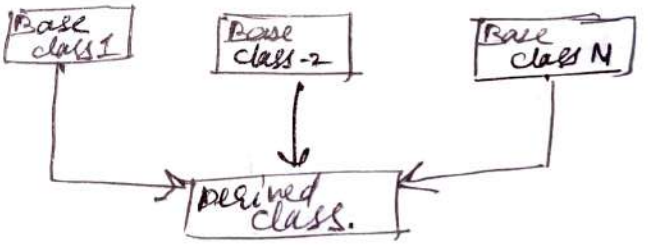
(*) Example:

```
class Animal:
    def speak(self):
        print("Animal Speaking")
class Dog(Animal):          # child class Dog inherits base class Animal.
    def bark(self):
        print("dog barking")
class DogChild(Dog):        # child class DogChild inherits base class Dog.
    def eat(self):
        print("Eating bread")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

O/P:→
```
Dog barking
Animal Speaking
Eating bread.
```

(*) Python Multiple Inheritance:

In this, Derived class inherits multiple base classes.



Syntax:

```
class Base1:
    <class-suite>
class Base2:
    <class-suite>
class BaseN:
    <class-suite>
class Derived(Base1, Base2, .. BaseN)
    <class-suite>
```

Example:

```
class calculation1:
    def sum(self,a,b):
        return a+b
class calculation2:
    def mul(self,a,b):
        return a*b
class Derived(calculation1,calculation2):
    def Divide(self,a,b):
        return a/b
d = Derived()
print(d.sum(10,20))
print(d.mul(10,20))
print(d.divide(10,20))
```

O/P :→ 30
200
0.5

⊛ issubclass(sub,sup) method:

This is used to check the relationships between the specified classes. It returns "true" if the first class is the subclass of the second class, and false, otherwise.

eg:

```
class cal1:
    def sum(self,a,b):
        return a+b
class cal2:
    def mul(self,a,b):
        return a*b
class Derived(cal1,cal2):
    def divide(self,a,b):
        return a/b
d = Derived()
print(issubclass(Derived,cal2))
print(issubclass(cal1,cal2))
```

O/P :→ True
False.

⊛ Isinstance(obj, class) method:

It is used to check the relationship between the objects and classes. It returns "true" if the first parameter, i.e. obj is the instance of the second parameter, i.e. class

eg:
```
class Cal1:
    def Sum(self,a,b):
        return a+b
class Cal2:
    def Mul(self,a,b):
        return a*b
class Derived(Cal1, Cal2):
    def Divide(self,a,b):
        return a/b
d = Derived()
print(isinstance(d, Derived))
```

O/P :→ True

⊛ Method Overriding:

In this method, we can provide some specific implementation of the parent class method in our child class. It is required where some different definition of a parent class method is needed in the child class.

eg:
```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

O/P :→ Barking

Data Abstraction in Python:

In Python, data hiding can be done by adding double underscore (__) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Eg:
```
class Emp:
    __count=0
    def __init__(self):
        emp.__count=emp.__count+1
    def display(self):
        print("The number of employee", emp.__count)
e=emp()
e1=emp()
Try:
    print(e.__count)
finally:
    e.display()
```

O/p :→ The number of employee 2
AttributeError: 'Emp' object has no attribute '__count'

Regular Expressions

RE-1

Regular expressions can be used to check if a string contains the specified search pattern. R.E. is a sequence of characters that forms a search pattern.

Expressions can include Text matching, Repetition, branching & Pattern-composition etc.

Python supports regular expressions through libraries. It supports various things like modifier, Identifiers and white space characters.

Python has a built in package called "Re" for Regular expressions.

Syntax: `import re`

* Simple Character Matches:

SNo	function	Description
01.	match	It matches the given pattern in a string, if found then gives true, else false
02.	search	It returns the match object if there is a match found in the string.
03.	findall	It returns a list that contains all the matches of a pattern in the string.
04.	split	returns a list in which the string has been split in each match.
05.	Sub	Replace one or many matches in the string.

Shaik Rag

Example:

```
import re
str = "How are you. How is everything"
matches = re.findall("How", str)
print(matches)
```

O/P :⇒ ['How', 'How']

✱ Special characters:

metacharacter/special character is a character with the specified meaning.

Meta character	Description	Example
[]	It represents the set of characters	"[a-z]"
\	represents special sequence	"\r"
.	It signals that any character is present at some specific place	"Ja.V."
^	represents the pattern present at the begining of the string	"^Java"
$	represents the pattern present at the end of the string	"point"
*	It represents 0 or more occurences of a pattern in the string	"hello*"
+	It represents 1 or more occurences of a pattern in the string	"hello+"
{}	The specified number of occurences of a pattern in the string	"Java{2}"
|	represents either this or that character is present	"Java|point"
()	Capture and group.	

Example:

(1)
```
import re
Str = "The Computer Programming"
x = re.findall("[a-m]", str)
print(x)
```

o/p:→ ['h', 'e', 'm', 'e', 'g', 'a', 'm', 'm', 'i', 'g']

(2)
```
import re
x = re.findall("^hello", str)
if(x):
    Print("yes, the string starts with 'hello'")
```

Shalini Raj

```
else:
  print("No Match")
```

O/P :→ Yes, the string starts with 'hello'.

⊛ Character Classes

A special sequence is a \ followed by one of the character in the list below and it has a special meaning-

Character	Description	Example
\A	Returns a match if the specified characters are at the begening of the string	"\AThe"
\D	Returns a match where the string "Does not" contain digits	"\D"
\s	Returns a match where the string contains a white space characters	"\s"
\S	Returns a match where the string "Does Not" contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (a-z, 0-9, _)	"\w"
\W	Returns a match where the string "Does Not" Contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Example:

```
import re
Str = "The Computer Programming"
x = re.findall("\AThe", str)   # check if the string starts with "The"
print(x)
if(x):
  print("Yes, there is a match!")
else:
  print("No Match")
```

O/P :→ ['The']
Yes, there is a match!

Quantifiers:

Quantifiers are used to specify the number of occurences to match.

A quantifier after a token, which can be a single character or group in brackets, specifies how often that preceding element is allowed to occur.

Most common quantifiers are ?, *, +.

example:

```
import re
str = "The computer programming requires logic skills"
x = re.findall("pyc*", str)   # check if the string contains "py" followed by 0 or more "c" characters.
print(x)
if (x):
    print ("Yes, there is at least one match!")
Else:
    print ("No match")
```

O/P:→ ['py']
Yes, there is at least one match

The Dot character:

Dot character represents any character (except newline char) in expression.

eg:
```
import re
str = "Hello World"
x = re.findall("he..o", str)   # search for a sequence that starts with "he" followed by two (any) characters and an "o":
print(x)
```

O/P:→ ['hello']

Greedy Matches:

*, +, ? quantifiers are all greedy. They match as much text as possible.

eg: 'a+' will match as many 'a' as possible in your string 'aaaa'.

Greedy Quantifiers

+ → one or more matches

* → zero or more matches

? → zero or one matches

{m,n} → m times at least and n times atmost.

eg: Let (bc)+ can be matched in 6-different ways as below:

abcbcbcde
abcbcbcde
abcbcbcde → left-most longest — "greedy." (starting from 1st occurence to max)
abcbcbcde
abcbcbcde
abcbcbcde

minimal match can also be done by using '?' at the end of RE i.e. '*?', this is done by "left-most shortest" or minimal match.

and '+?' will match at least one instance.

and '??' will match either 0 or 1 instances.

example:

```
import re
string = "<h1>Heading</h1><p>HTML text.</p>"
match = re.search("<.*>", string)
if match:
print ("Greedy")
print ("start:", match.start(0))
print ("end:", match.group(0))
match = re.search("<.*?>", string)
if match:
print ("\n Non-greedy")
print ("start:", match.start(0))
print ("end:", match.end(0))
print ("group:", match.group(0))
```

O/P:→

```
Greedy
start: 8
end : 33
group: <h1>Heading</h1><p>HTML text.</p>
Non-greedy
start:0
end:4
group:<h1>
```

(*) Grouping: It match whatever regular expression is inside parenthesis, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed.

eg:

Eg:

```
import re
string = "<p>HTML text.</p>"
match = re.match("<p>(.*)<%.p", string)
if match:
print ("start:", match.start(1))
print ("end:", match.end(1))
print ("group:", match.group(1))
string = "'cat':'Risky','dog':'Spot','fish':'Bubbles'"
match = re.search("'cat':'(.*?)','dog':'(.*?)','fish':
                  '(.*?)'", string)
if match:
print("groups:", match.group(1), match.group(2), match.
                                    group(3))
```

O/P:→

```
Start: 3
end : 13
group: HTML text
groups: Risky Spot Bubbles.
```

(*) Matching at Beginning or END:

^ searches string in the given expression from beginning.

Eg:

```
import re
str="hello world"
x = re.findall("^hello", str)   # check if the string starts with 'hello'.
if (x):
  print("Yes, the string starts with 'hello'")
else:
  print ("NO Match")
```

O/P:→ Yes, the string starts with 'hello'

(*) $ searches string in the given expression at the end.

example:

```
import re
x = re.findall("world$", str)   # check if the string ends with 'world'
if (x):
    print("Yes, the string ends with 'world'")
else:
    print("NO match")
```

O/p:→ Yes, the string ends with 'hello'.

(*) Match objects: It contains the information about the search and the output. If there is no match found, the None object is returned.

eg:
```
import re
str = "How are you. How many lines?"
matches = re.search("How", str)
print(type(matches))
print(matches)
```

O/p:→ <class '_sre.SRE_Match'>
<_sre.SRE_Match object; span=(0,3), match='How'>

(*) Match object Methods: Following methods are associated with Match.

span() — It returns the tuple containing the start and end position of the match.

string() — It returns a string passed into the function.

group() — Part of the string is returned where the match is found.

eg:
```
import re
str = "How are you. How is everything"
matches = re.search("How", str)
print(matches.span())
print(matches.group())
print(matches.string())
```

O/p:→ (0,3)
How
How are you. How is everything

* **Substituting:** It means replacing one character/string with another. sub() function is used for it.

Eg:
```
import re
str = "The rain in spain"
x = re.sub("\s", "9", str)   # Replaces all white spaces char with the digit "9".
print(x)
```

O/P:→ The9rain9in9spain

* **Splitting a String:** It returns a list where the string has been split at each match.

Eg:
```
import re
str = "The rain in spain"
x = re.split("\s", str)   # split the string at every white space character.
print(x)
```

* **Compiling Regular Expressions:**

If you want to use the same regular expression more than once in a script, it might be a good idea to use a regular expression object, i.e. the regular expression is compiled.

Syntax:

re.compile(pattern[, flags])

compile method returns a RE object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

However, compiled RE are not saving much time.

* **Flags:** Compilation flags let you modify some aspects of how regular expressions work. It is denoted by two names IGNORECASE and I.

(*) Substituting: it means replacing one character/string with another. sub() function is used for it.

eg:
```
import re
str = "The rain in spain"
x = re.sub("\s", "9", str)   # Replaces all white spaces char with the digit "9".
print(x)
```

O/P:→ The9rain9in9spain

(*) Splitting a String: It returns a list where the string has been split at each match.

eg:
```
import re
str = "The rain in spain"
x = re.split("\s", str)   # split the string at every white space character.
print(x)
```

(*) Compiling Regular Expressions:

If you want to use the same regular expression more than once in a script, it might be a good idea to use a regular expression object, i.e. the regular expression is compiled.

Syntax: reCompile(pattern [, flags])

compile method returns a REobject, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

However, Compiled RE are not saving much time.

(*) Flags: compilation flags let you modify some aspects of how regular expressions work. It is denoted by two names IGNORECASE and I.