

Python

Input and Output Statements --->

Input statements :

By using input() function we can read dynamic data from the keyboard. The return type of input() function is always string type.

There are three ways to read and convert (Type casting - str to int)

- WAP to read value and sum

#reading and converting

```
a = int(input('Enter 1st number:'))
b = int(input('Enter 2nd number:'))
sum = print('The sum of two numbers are:', a+b)
```

or

#reading values

```
a = input('Enter 1st number:')
b = input('Enter 2nd number:')
```

converting string values

```
x = int(a)
y = int(b)
```

```
print('The sum is :', x+y)
```

or

```
print('The sum :', int(input("Enter 1st number:")) +
      int(input('Enter 2nd number:')))
```

NOTE : While converting str --> bool type ==> if the string is empty then we will get false.
if the string is non-empty then we will always get True.

IMPORTANT : evaluate: eval() function: It will analyze the type of data represented .
eval(string) --> It will analyze the type of data represented by string and will convert it into corresponding type.
It is applicable for all data type.

Not only values but it can evaluate expression too.\

example -->

```
x = eval(input('Enter a number:'))
print(type(x))
y = eval(input('choose one : TRUE OR FALSE:'))
print(type(y))
z = eval(input('Enter expression:'))
print(type(z))
```

```

print(type(y))
z = eval(input('Enter expression:'))
print(type(z))

```

If eval() function is there then what was the need of type casting(int,float,bool) ?

when we always expect a particular data type as input or output better to go with type casting.

If you do not know the type of input internally then go with eval().

Based on provided value,convert into corresponding type automatically then we should go for eval().

How to read multiple values from the keyboard by using a single input() function ?

Now suppose we have a string '10 20' --> this string has 2 values separated by space separator. If we use split() fun the values will be splitted into two strings in a list.

```

l = '10 20'.split()
print(l)  #['10','20']

```

If the string value is separated by (,) then we have to pass , argument inside split() -->

```

'10,20': l='10,20'.split(',')
print(l)

```

list type but both the values are string.

split() is the function which can be used on the string object to split into multiple values based on separator.

```

for x in '100:200'.split(':')

```

--> means for each value of x in this list

```

int(x) for x in '100:200'.split(':')

```

--> can you typecast into int.

```

for x in '100:200'.split(':')

```

Now the string values of list will be converted into int.

syntax : l=[int(x) for x in '100:200'.split(':')] #list comprehension

--> Now this is a newly create list of those int values that we ve converted from string.

This whole concept is called List comprehension.

Now we can assign values in different variables -->

```

a,b=[int(x) for x in '100:200'.split(':')]
print(a)
print(b)

```

This is called list unpacking.

Now input() will always return string so

```
Syntax : a,b=[int(x) for x in input('Enter 2 int values:').split()]
          print('The sum:',a+b)
```

Conclusion : So We will have a list in which the list values are of string types , then we will type cast those string values into int and will create a new list. we can now use those int values.

Without typecasting -->

```
Enter values : '10,11,12'
```

```
output : ['10','12','13']
```

With typecasting into int -->

```
output : [10,12,13]
```

Command line arguments(CLAs) :

One way to provide data or input to program is input()

There is another way to provide input that is CLAs.

The arguments which we are passing from the command prompt are called CLAs.

`python test.py` --> command to execute python program.

`py test.py 10 20 30` --> input arguments passing with the command to execute program.

↑
arguments

How can you read those arguments ?

PVM will create list of those arguments with the name:argv --> predefined variable therefore we can't change the name.

```
argv=[]
```

- argv variable available in sys module
- syntax --> `from sys import argv`

Example --> Now in command prompt --> `py test.py 10 20 30`

```
from sys import argv
print(argv) #['test.py','10','20','30']
```

Here we can see

- The first element of the list will be the python file name
- The elements by default are of string type so we need to typecast to change it to some other type.

Steps to follow -->

- first we need to create a python file of the name of your choice.
- Then open the command prompt and go to the directory where you have created the python file.
- Then write the python file execution command and pass arguments(input values)

- Then write the program and use the input values in your program which is stored in a variable name argv.
- Then execute the program from command prompt.

➤ **WAP to read int values as command line arguments and print sum ?**

Inside Code editor -->

```
from sys import argv
args = argv[1:] # means consider the values from index 1 and ignore the
value in index 0(which will be python file name in string form).
print(args)
total = 0
for x in args:
    total = total + int(x) ← here we have typecast into int since by
default the values will be in string
print('The Sum:', total)
```

Command Prompt:

```
C:\Users\ANKIT ASWAL\Desktop\python tutorials\Command line arguments>
python sum.py 10 20 30
['10', '20', '30']
The Sum: 60
```

- we used args so we could consider elements from index 1 and leave the element of index 0 which is irrelevant for our program.
- for concatenation both values should be str type.
- int+str --> error

Another example -->

```
from sys import argv
print(argv[1]) #ankit
```

```
#python test.py ankit aswal --> in command prompt
The space between test.py ankit and aswal are separators.
index --> 0      1      2
```

Now if I use double quotes then the separator will be ignored.

```
#python test.py "ankit aswal" #Now ankit aswal will be treated as one CLA.
test.py "ankit aswal"
```

```
from sys import argv
print(argv[1]) #ankit aswal
```

NOTE:

1. Space is the separator between command line arguments. If our command line argument itself contains space then we should enclose within double quotes.

2. Command line arguments are always available in str form. we have to take

line argument itself contains space then we should enclose within double quotes.

2. Command line arguments are always available in str form. we have to take care about typecasting if we are expecting any other type.
3. If we are trying to access list elements with out of range index then we will get Index error.

Already when we had input function then what was the need of command line argument?

From what I know --> *For example there is a Student details.*

Whatever the data student has will be taken as input through input function. But there are data that need to be configured(data base) by programmer before student input his data through input(). This is configured by programmer not by users(student).

For example --> the name of student will be given by student itself. But where to store it will be configured by programmer by passing CLA.

Therefore we have to use input() for student data But where to store that data we need to configured it by passing CLA.

CODE -->

```
from sys import argv
name = input('Enter name:')
marks = int(input('Enter marks:'))
dbname = argv[1]
if dbname == 'oracle':
    print('Data is storing in orcale')
elif dbname == 'mongodb':
    print('Data is storing in mongodb')
else:
    print('Data is storing is mysql')
```

IN COMMAND PROMPT -->

•
C:\Users\ANKIT ASWAL\Desktop\python tutorials\Command line arguements>python student.py oracle --> input as CLA.
Enter name:Ankit
Enter marks:298
Data is storing in orcale

OUTPUT STATEMENTS -->

There is only function to print the output that is print().

Form-1: print() without any argument --> To print blank line(\n)

example --> `print('Ankit')
print()
print('Aswal')`

Form-2: print(string) -->

example--> `print('Hello') #Hello`

In the string we can use escape characters (\n, \t) also

example --> `print('Hello\nworld') #Hello
world`

In the string we can use + and * operators -->

example --> `print('Hello' + 'world') #Hello world'
print('a'*30) #aaa`

Ques. What is the difference between print('Hello'+ 'world') and print('Hello','world')

Ans. In the first one concatenation will happen #Helloworld but in the second one #Hello world, hello and world is separated by space if you provide 2 arguments.

Form-3: Any number of arguments we can pass.

example --> `a,b,c =10,20,30
print(a,b,c) #10 20 30`

Form-4: print() with sep attribute -->

By default separator is space.

If we want to use other separator other than space then we have to use sep attribute.

example -->

`a,b,c =10,20,30
print(a,b,c,sep=',') #10,20,30`

`a,b,c =10,20,30
print(a,b,c,sep=':') #10:20:30`

`a,b,c =10,20,30
print(a,b,c,sep='') #102030`

Form-5: print() with end attribute -->

Remember that By default there is a \n(new line) internally after every print() statement .

If we want to print multiple print() in a single line then we have to use end attribute.

```
example --> print('Ankit')
              print('Aswal')
              print('ManUtd') output --> #Ankit
                                         Aswal
                                         ManUtd
```

```
              print('Ankit',end='-')
              print('Aswal')
              print('ManUtd') output --> #Ankit-Aswal
                                         ManUtd
```

NOTE: By default there will always be a blank line left after all the print() statement executed. If you don't want to left that line use end attribute in the last print() statement.

example :

```
print('Ankit',end='-')
print('Aswal',end='-')
print('ManUtd',end='.') #Ankit-Aswal-ManUtd.
```

CONCLUSION : *whenever multiple values with single print() statement :sep
whenever multiple print statements : end*

If we use both end and sep together -->

```
print(10,20,30,sep='-',end='$')
print('A','B','C',sep='**',end='$$')
print('Completed')          #10-20-30$A**B**C$$Completed
```

- sep is separator between two values in a single print() while end comes between two print() function.
- NOTE: The default value of sep=space and for end='\n'.

Form-6 : print(object) -->

We can pass any python object happily -->
list,dict,tuple,string,complex,int,etc..

Form-7 : print(formatted string)

OLD STYLE FORMATTING -->

String formatting --> Adding values to string

syntax : print("formatted string" %(variable list))

format specifier -->

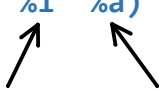
%i --> int

%d --> int

%f --> float


`%s --> string and for all the remaining type.`

```
example --> a,b,c=10,20,30
             print('a value is %i' %a)    #10
```



format specifier variable list

```
a,b,c=10,20,30
print('a value is %d and b value is %d' %(a,b))    #10 20
```



variable list

```
example -->
name='Durga'
list1=[10,20,30]
print('Hello %s, the list items are: %s' %(name,list1)) #using formatters
Output --> Hello Durga, the list items are: [10, 20, 30]
```

But If we do not use formatters, to print the same output as above is a headache.
something like that --> `print('Hello',name,',','the list items are:',list1)`
#without formatters

So formatters makes the code a bit easy.

NEW STYLE FORMATTING --> with replacement operator {}

```
name='Ankit'
salary=10000
Mother='Kapo'
```

OLD STYLING :

```
print( Hello %s,Your Salary is %d,Your Mother %s is waiting'
%(name,salary,Mother))
```

NEW STYLING :

- `print('Hello {},Your Salary is {},Your Mother {} is waiting'.format(name,salary,Mother))`

NOTE : Order is important in the above example.

- we can also take index :

```
print( Hello {0},Your Salary is {1},Your Mother {2} is
waiting'.format(name,salary,Mother))
```

NOTE : Order is not important when indexing.

```
print( 'Hello {2},Your Salary is {0},Your Mother {1} is
waiting'.format(salary,Mother,name))
```

- Taking Keywords(first word)

```
print( 'Hello {n},Your Salary is {s},Your Mother {M} is
```



```
waiting'.format(M=Mother,n=name,s=salary))
```

➤ Another way :

```
print( f'Hello {name},Your Salary is {salary},Your Mother {Mother} is  
waiting')
```

expected output :

Hello Ankit,Your Salary is 10000, Your Mother Kapo is waiting.