Python Input-output and variables

Taking Input

Most of the problems require you to take inputs of different kinds so here are some of the important techniques

Taking single input

If we want to take only one input then we only need to use the input() function and then we can convert it to the type we want.
input() function can be used in two alternative way

```
# Take input in next line of the prompt
   a) print( “What is your name? ” )
            name = input()

# Take input in the line of prompt
   b) n = input(“What is your name? ”)
```

After reading input as per requirement we can change it to another type

```
For strings use                  n = input([‘’])
For integers use                 n = int(input([‘’]))
For floating-point numbers use   n=float(input([‘’]))
```

Taking a known number of inputs

If we want to take a fixed number of space-separated or comma separated inputs (for eg- dimensions of a matrix) then we use the following technique
For strings

```
     m,n = input().split(‘ ’) or
     m,n = input().split()        #  space-separated input

     m,n = input().split(‘,’)     # comma-separated input
```

This will assign one of the two space-separated strings to both m and n respectively.
For integers and floats

```
m,n = map(int,input().split()) or
m,n = map(float,input().split())
```

As we know that `input().split()` returns an iterable so what we do is convert each of the objects to integers using `map()` which applies a function (`int()` in our case) to every object of the iterable.

If we have two arguments but we don't want two waste a variable for one of them then we can use `_` in its place for eg-

We want to take only first and the third integer in the variable m and n, in the example given below then we would use the following piece of code-

```
1 2 3
m,_,n = map(int,input().split())
```

Taking a variable number of inputs

For taking a variable number of space-separated inputs we usually assign them to a list but we could also use set or tuple according to our requirement.

For strings -

```
I love python
l = list(input().split())
```

The value of l will be l=['I','love','python']

For integers or floats -

```
1 2 3 4 5 6 7
l = list(map(int,input().split()))
```

The value of l will be l = [1,2,3,4,5,6,7]

If we want sets or tuples then use tuple() or set() instead of list().

Providing Output

Outputting the data wouldn't be that difficult, the only possible cases are

Outputting a text

To provide a text line in the output print() function is used with a ' ' or " ".

```
print('Hello World')
```

Output will be : Hello World

Outputting variables on different lines

Almost 90% of problems would require you to output this way. Its Very simple consider the following example -

```
result = [1,2,3,4,5]
for i in result:
print(i)
```

The **output** will be-

```
1
2
3
4
5
```

Outputting variables in the same line

If you want to output the result in the **same line** then you will have to use the end attribute of print()

```
result = [1,2,3,4,5]
for i in result:
print(i, end=' ')
```

The **output** will be **1 2 3 4 5**

Outputting Text and variables together

To provide output containing text and variable together print() function is used. Within print(), text is kept within ‘ ‘ or “ “ and variables without quotation and each one is separated by comma [,].

```
a=10
b=20
print('value of a is = ',a,' and value of b is ',b)
```

Output of the code will be: **value of a is = 10 and value of b is 20**

Advanced outputting techniques

Some problems like Google’s codejam and kickstart require you to mention case number with the output for that we use a **C like technique to print the output**.
For example -

```
result = [15,23,32]
for i in range(len(result)):
  print("Case #{}: {}".format(i+1, result[i]))
```

This would give the following output:

```
Case #1: 15
Case #2: 23
Case #3: 32
```

What's happening here is the {} is acting as a placeholder for a variable that is provided in format() just like %d in C. Here are some more examples-

```
print("I love {}".format("python"))
```

```
Output: I love python
```

```
l=['python','is','the','best','language']
print("I love python because {}".format(' '.join(l)))
```

Output-

```
I love python
I love python because python is the best language
```

Creating variables and assigning values

To create a variable in Python, all you need to do is specify the variable name, and then assign a value to it.

<variable name> = <value>

Python uses = to assign values to variables.

There's no need to declare a variable in advance or to assign a data type to it. Assigning a value to a variable itself declares and initializes the variable with that value. There's no way to declare a variable without assigning it an initial value.

Rules for variable naming:

1. Variables names must start with a letter or an underscore.

```
x = True   # valid
_y = True  # valid
9x = False # invalid as it starts with numeral
$y = False # invalid as it starts with symbol
```

2. The remainder of your variable name may consist of letters, numbers and underscores.

```
has_0_in_it = "Still Valid"
```

3. Names are case sensitive.

```
x = 9
y = X*5
```

=>NameError: name 'X' is not defined

Even though there's no need to specify a data type when declaring a variable in Python, while allocating the necessary area in memory for the variable, the Python interpreter automatically picks the most suitable built-in type for it:

a = 2 print(type(a)) #Output: <type 'int'>	pi = 3.14 print(type(pi)) #Output: <type 'float'>	c = 'A' print(type(c)) #Output: <type 'str'>	q = True print(type(q)) #Output: <type 'bool'>	x = None print(type(x)) #Output: <type 'NoneType'>

When you use = to do an assignment operation, what's on the left of = is a name for the object on the right. That is:

```
a_name = an_object
```

"a_name" is now a name for the reference to the object "an_object"

For ex: pi = 3.14 , then pi is a name (not the name, since an object can have multiple names) for the object 3.14 .

You can assign multiple values to multiple variables in one line. Note that there must be the same number of arguments on the right and left sides of the = operator:

a, b, c = 1, 2, 3 print(a, b, c) # Output: 1 2 3	a, b, c = 1,2 Error: Traceback ValueError: not enough values to unpack (expected 3, got 2)	a, b = 1, 2, 8 Error: Traceback ValueError: too many values to unpack (expected 2)	a = b = c = 1 print(a, b, c) # Output: 1 1 1

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them. Python has Six standard data types-

(i) Numbers
(ii)String
(iii)List
(iv)Tuple
(v)Dictionary
(vi) Set

Number

Number data types store numeric values. Number objects are created when we assign a value to them. For example-

var1 = 1
var2 = 10

We can also delete the reference to a number object by using the del statement. The syntax of the del statement is -

del var1

Python supports three different numerical types -

int (signed integers)
float (floating point real values)
complex (complex numbers)

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are real numbers and j is the imaginary unit.

Example: int 10 100
float 0.0 14.27
Complex 6+13j 14j 45.j

String

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at **0 in the beginning** of the string and working their way from **n-1 to the end**.
The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example-

```
#!/usr/bin/python3
str = 'Hello World!'
print (str)            # Prints complete string
print (str[0])         # Prints first character of the string
print (str[2:5])       #Prints characters starting from 3rd to 5th
print (str[2:])        #Prints string starting from 3rd character
print (str * 2)        # Prints string two times
print (str + "TEST")   # Prints concatenated string
```

Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the **differences** between them is that all the **items belonging to a list can be of different data types**. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at **0 in the beginning** of the list and working their way to **end n-1**. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example-

```
#!/usr/bin/python3
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list)               # Prints complete list
print (list[0])            # Prints first element of the list
print (list[1:3])          # Prints elements starting from 2nd to 3rd
print (list[2:])           #Prints elements starting from 3rd element
print (tinylist * 2)       # Prints list two times
print (list + tinylist)    # Prints concatenated lists
list[1:3]=['AAAA', 'BBBB']  # Modify the value of the list
```

```
print (list[1:3])          #Prints elements starting from 2nd to 3rd
```

Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
The main **difference between lists and tuples** is- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while **tuples** are enclosed in parentheses (( )) and **cannot be updated**. Tuples can be thought of as read-only lists. For example-

```
#!/usr/bin/python3
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2)
tinytuple = (123, 'john')
print (tuple)               # Prints complete tuple
print (tuple[0])            # Prints first element of the tuple
print (tuple[1:3])          #Prints elements starting from 2nd till 3rd
print (tuple[2:])           # Prints elements starting from 3rd element
print (tinytuple * 2)        # Prints tuple two times
print (tuple + tinytuple)   # Prints concatenated tuple
tuple[1]='AAAA'         #Error:'tuple' does not support item assignment
```

Dictionary

Python's dictionaries are a kind of hash-table type. They work like associative arrays or hashes found in Perl and **consist of key-value pairs**. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). Values within the dictionary can be assigned either by dict_name=value or dict_name={keys:value,keys:value,...} pair. For example-

```
#!/usr/bin/python3
dict = {}
```

```
dict['one'] = "This is one"
dict[2] = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])               # Prints value for 'one' key
print (dict[2])                   # Prints value for 2 key
print (tinydict)                  # Prints complete dictionary
print (tinydict.keys())           # Prints all the keys
print (tinydict.values())         # Prints all the values
```

Set

Sets are unordered collections of unique objects, there are two types of set:

1. Sets - They are mutable and new elements can be added once sets are defined

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)    # duplicates will be removed
>>> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)
# unique letters in a
>>> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
>>> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Frozen Sets - They are immutable and new elements cannot be added after they are defined.

```
b = frozenset('asdfagsa')
print(b)
>>> frozenset({'f', 'g', 'd', 'a', 's'})
cities = frozenset(["Frankfurt", "Basel","Freiburg"])
print(cities)
>>> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

Python Basic Operators

Operators are the constructs, which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called the operator.

Python language supports the following types of operators-

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Assume variable **a** holds the value **10** and variable **b** holds the value **21**, then-

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	a + b = 31
- (Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* (Multiplication)	Multiplies values on either side of the operator	a * b = 210
/ (Division)	Divides left hand operand by right hand operand	b / a = 2.1
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** (Exponent)	** Exponent Performs exponential (power) calculation on operators	a**b =10 to the power 21
// (Floor Division)	The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Python Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators. Assume variable **a** holds the value **10** and variable **b** holds the value **20**,

Operator	Description	Example
==	== If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal,then the condition becomes true.	(a!= b) is true.
>	If the value of the left operand is greater than the value of the right operand, then the condition becomes true.	(a > b) is not true.
<	If the value of the left operand is less than the value of the right operand, then the condition becomes true.	(a < b) is true.
>=	If the value of the left operand is greater than or equal to the value of the right operand, then the condition becomes true.	(a >= b) is not true.
<=	If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20,

Operator	Description	Example
= (Assigns)	values from right side operands to left side operand	c = a + b assigns value of a + b into c

+= (Add AND)	Add AND It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= (Subtract AND)	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= (Multiply AND)	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= (Divide AND)	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= (Modulus AND)	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= (Exponent AND)	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= (Floor Division)	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if **p = 60**; and **q = 13**; Pyhton's **built-in function bin()** can be used to obtain binary representation of an integer number. Now in binary format they will be as follows-

a = 0011 1100 (bin(p))

b = 0000 1101 (bin(q))

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Operator	Description (a = 0011 1100 b = 0000 1101)	Example
& Binary AND)	Operator copies a bit to the result, if it exists in both operands (a & b)	(a & b =12) (0000 1100)
| Binary OR)	It copies a bit, if it exists in either operand.	(a | b) = 61 (0011 1101)
^ (Binary XOR)	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (0011 0001)
~ (Binary 1's Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (1100 0011 in 2's complement form due to a signed binary number.
<< (Binary Left Shift)	The left operand's value is moved left by the number of bits specified by the right operand.	a <<2 = 240 (1111 0000)
>> (Binary Right Shift)	The left operand's value is moved right by the number of bits specified by the right operand.	a >>1 = 15 (0000 1111)

Python Logical Operators

The following logical operators are supported by Python language. Assume variable a holds True and variable b holds False then-

Operator	Description (a = 0011 1100 b = 0000 1101)	Example
and (Logical AND)	If both the operands are true then condition becomes true.	(a and b) is False.
or (Logical OR)	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not (Logical NOT)	Used to reverse the logical state of its operand.	Not(a and b) is True.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists,or tuples. There are two membership operators as explained below-

Operator	Description	Example
in	Evaluates to true, if it finds a variable in the specified sequence and false otherwise.	**x in y**, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true, if it does not find a variable in the specified sequence and false otherwise.	**x not in y**, here not in results in a 1 if x is not a member of sequence y.

Example:

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]
if ( a in list ):
   print ("Line 1 - a is available in the given list")
else:
   print ("Line 1 - a is not available in the given list")
if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")
```

Output:

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
```

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Example:

```
a = 20
b = 20
print ('Line 1','a=',a,':',id(a), 'b=',b,':',id(b))
if ( a is b ):
     print ("Line 2 - a and b have same identity")
else:
     print ("Line 2 - a and b do not have same identity")
if ( id(a) == id(b) ):
     print ("Line 3 - a and b have same identity")
else:
     print ("Line 3 - a and b do not have same identity")
b = 30
print ('Line 4','a=',a,':',id(a), 'b=',b,':',id(b))
if ( a is not b ):
     print ("Line 5 - a and b do not have same identity")
else:
     print ("Line 5 - a and b have same identity")
```

Output: ??

Python Operators Precedence

Operator	Meaning
**	Exponentiation (raise to the power)
~, +, -	complement, unary plus and unary minus (method names for the last two are +@ and -@)
*, /, %, //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise `XOR' and regular `OR'
<=, <, >, >=	Comparison operators
==, !=	Equality operators
=, %=, /=, //=, -=, +=, *= **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators