x,y = input().split()

x,y = map(int,input().split(“ “))

l,b = [int(x) for x in input().split(“ ”)]

For function overloading :

Multiple Dispatch Decorator Can be installed by:

```
pip3 install multipledispatch
```

```
from multipledispatch import dispatch

@dispatch(int,int)
def mult(a, b):
    print(a * b)

@dispatch(int,int,int)
def mult(a, b, c):
    print(a * b * c)


mult(2, 3, 5)
mult(2, 3)
```

***args for non keyword arguments :**

```
def sum(*args):

    sm = 0

    for i in args:
        sm += i

    print(sm)


sum(2,3,4,5)
```

output:

```
14
```

**kwargs for keyword arguments :

```
def fun(**kwargs):
    for key,value in  kwargs.items():
        print(key,value)


fun(name = "saif", roll = 101)
```

output:

```
name saif
roll 101
```

For default parameter :

- Any number of parameters in a function can have a default value.
- The conventional syntax for using default parameters states that once we have passed a default parameter, all the parameters to its right must also have default values.
- In other words, non-default parameters cannot follow default parameters.

For example: if we had defined the function header as:

```
def greet(wish = "Happy Birthday", name):
    #statement
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Suppose list1 is [2, 3, 2, 14, 25].

```
print(list1[::-1])
```

[25, 14, 2, 3, 2]

```
# Number of rows
n = int(input())

input= [[int(col) for col in input.split()] for row in range n]

print(input)
```

User Input:

```
4
9 8 5 2
5 6 10 6
10 8 5 5
10 10 8 6
```

Output:

```
[[9, 8, 5, 2 ], [5, 6, 10, 6 ], [10, 8, 5, 5 ], [10, 10, 8, 6]]
```

```
l = list(map(int,input().split()))
```

String concatenation :

1. Using + operator
2. Using join() method
3. Using % operator
4. Using format() function

(1) Using + operator :

```
var1 = "Coding "
var2 = "Ninjas"

# + Operator is used to combine strings
var3 = var1 + var2

print(var3)
```

```
Coding Ninjas
```

(2) Using join function :

```
var1 = "Coding"
var2 = "Ninjas"


# join() method is used to join strings with an empty character as a separator("")
var3 = "".join([var1, var2])
print(var3)


# This time we use space(" ") as a separator for join() method.
var3 = " ".join([var1, var2])
print(var3)


li = ['One', 'Two', 'Three']
sep = '-Separator-'
print(sep.join(li))

 var = "TESTING"
 sep = "*"
 print(sep.join(var)) # String is also an iterable as it is a collection of characters.
```

```
CodingNinjas
Coding Ninjas
One-Separator-Two-Separator-Three
T*E*S*T*I*N*G
```

(3) Using % operator :

```
var1 = "Coding"
var2 = "Ninjas"

# % Operator is used here to combine the string stored in var1 and var2
print("%s %s" % (var1, var2))
```

```
Coding Ninjas
```

(4) Using format :

```
var1 = "Coding"
var2 = "Ninjas"

# format function is used here to
# combine the string
print("{} {}".format(var1, var2))
```

```
Coding Ninjas
```

To slice a string :

```
slice(stop)
slice(start, stop, step)
```

```
string[start:end:step]
```

OOPS :

def __init__(self): —---------> Constructor

Instance attributes :

Attributes created in **.__init__()** are called **instance** attributes. An instance attribute's value is specific to a particular instance of the class. All **Car** objects have a **name** and a **topSpeed**, but the values for the **name** and **topSpeed** attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

```
class Student:
    def __init__(self,name,rollno):
        self.name = name
        self.rollno = rollno

    def printDetails(self):
        print("name: ",self.name)
        print("rollno: ",self.rollno)


s = Student("parikh",101)
s.printDetails()
```

Class attributes :

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of. **__init__()**.

```
class Car:
    # Class attribute
    color = "Black"


    def __init__(self, name, topSpeed):
          self.name = name
          self.topSpeed= topSpeed
```

Inheritance :

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass by accessing the **__init__** method of the superclass in the following format:

```
<SuperClassName>.__init__(self,<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' **__init__** function, otherwise it will throw an error.

```
class Polygon:

    #Constructor
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    #Take user input for side lenghts
    def inputSides(self):
        self.sides = [int(input("Enter Side: ")) for i in range(self.n)]

    #Print the sides of the polygon
    def displaySides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):

    def __init__(self):
        #Calling constructor of superclass
        Polygon.__init__(self, 3)

    def findArea(self):
        a, b, c = self.sides
        #Calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s * (s-a) * (s-b) * (s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

Try & Except :

```
l = ['a',0,2]

for ele in l :
    try :
        print("The entry is : ",ele)
        r = 1/int(ele)
        print("The reciprocal : ",r)

    except Exception as e :
        print("Error Occurred ",e.__class__)
    print("------------------------------------")
```

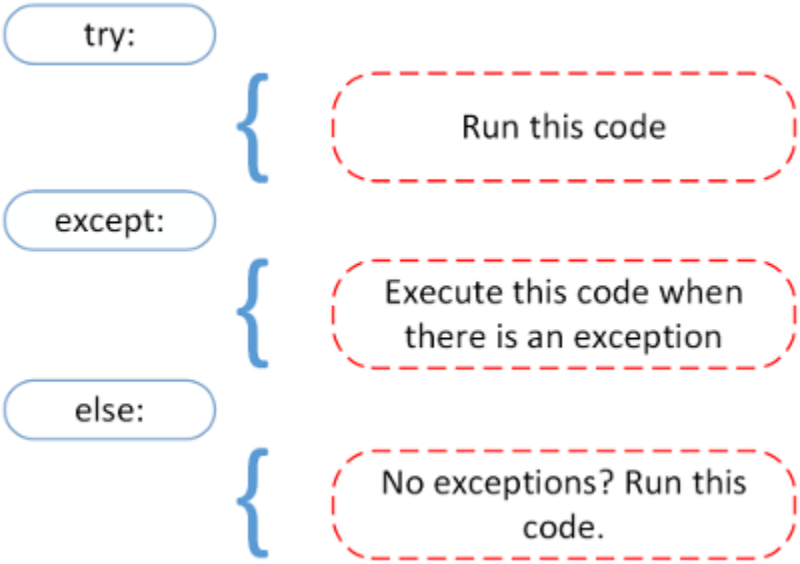
```
Finished in 60 ms
The entry is :  a
Error Occurred  <class 'ValueError'>
------------------------------------
The entry is :  0
Error Occurred  <class 'ZeroDivisionError'>
------------------------------------
The entry is :  2
The reciprocal :  0.5
------------------------------------
```

```
try:
    c = 2/1

except Exception as e:
    print("can't divide by zero")
    print(e)

else:
    print("Hi I am else block")
```

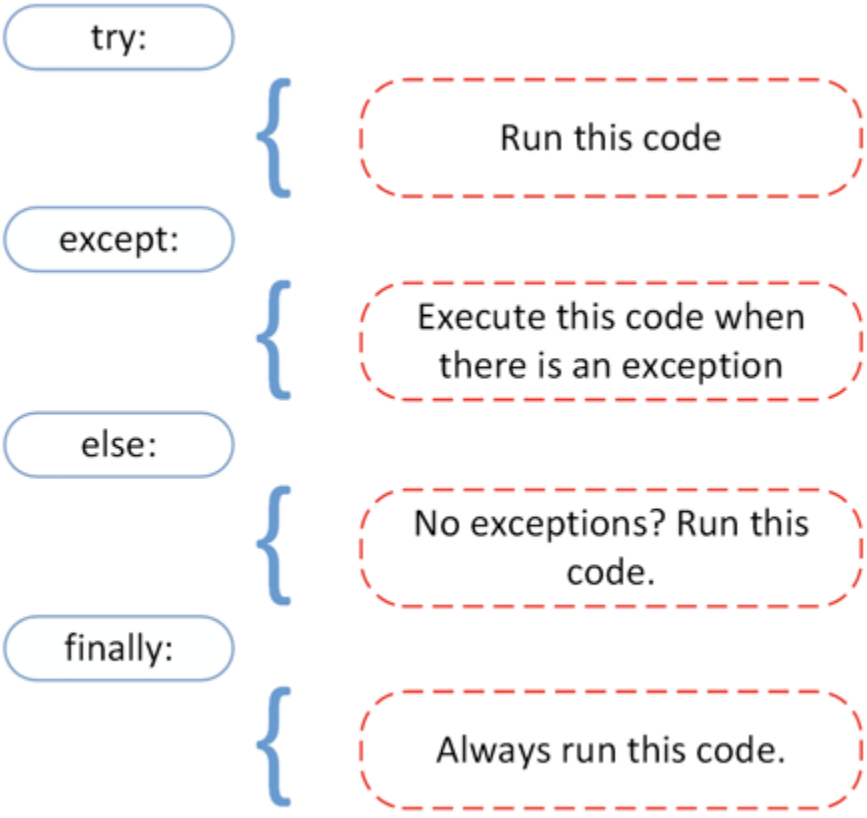
Output:

```
Hi I am else block
```

Try-Except-Else :



Try-Except-Else-Finally :



DefaultDict()

A DefaultDict is also a sub-class to the dictionary and returns a new dictionary-like object. Whenever it is used to provide some default values for the key that does not exist it never raises a KeyError.

Syntax:

```
class collections.defaultdict(default_factory)
```

Here, default_factory is a function that provides the default value for the dictionary created. If this parameter is absent then there is a KeyError that gets raised otherwise the very first argument of 'defaultdict' is 'default_factory' as a default data type for the dictionary.

```
from collections import defaultdict
d = defaultdict(int)
print(d['B'])

str_ = "rishabh arya"
for i in str_:
    d[i] += 1
print(d)
```

```
Finished in 55 ms
0
defaultdict(<class 'int'>, {'B': 0, 'r': 2, 'i': 1, 's': 1, 'h':
2, 'a': 3, 'b': 1, ' ': 1, 'y': 1})
```

OrderedDict()

Ordered dictionaries are similar to regular dictionaries with some extra capabilities relating to ordering operations.
An OrderedDict is also a sub-class of dictionary but unlike a dictionary, it remembers the order in which the keys were inserted.

```python
# A Python program to demonstrate working
# of OrderedDict
from collections import OrderedDict

print("This is a standard python Dict:\n")
d = {}
d['John'] = 1
d['Mary'] = 2
d['Lucy'] = 3
d['Brian'] = 4
# The order of the keys and values inserted won't be retained here
for key, value in d.items():
        print(key, value)

print("\nThis is an Ordered Dict:\n")
od = OrderedDict()
od['John'] = 1
od['Mary'] = 2
od['Lucy'] = 3
od['Brian'] = 4
# The order of the keys and values inserted would be preserved
for key, value in od.items():
        print(key, value)
```

Output:

```
This is a standard python Dict:

Brian 4
Mary 2
John 1
Lucy 3

This is an Ordered Dict:

John 1
Mary 2
Lucy 3
Brian 4
```

Counters

A counter is a subclass of the dictionary. It is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. It is used to keep the count of the elements in an iterable in the form of an unordered dictionary where the key represents the element in the iterable and value represents the count of that element in the iterable.

Note: It is equivalent to a bag or multiset of other languages.

```
# A Python program to show different
# ways to create Counter
from collections import Counter

# With sequence of items
print(Counter('codingninjas'))

# with dictionary
print(Counter({'n': 3, 'i': 2, 'd': 1, 'o': 1, 'a': 1, 'g': 1, 'c': 1, 'j': 1, 's': 1}))

# with keyword arguments
print(Counter(n=3, i=2, d=1, o=1, a=1, g=1, c=1, j=1, s=1))
```

Output:

```
Counter({'n': 3, 'i': 2, 'o': 1, 'a': 1, 's': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
Counter({'n': 3, 'i': 2, 's': 1, 'o': 1, 'a': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
Counter({'n': 3, 'i': 2, 's': 1, 'o': 1, 'a': 1, 'd': 1, 'c': 1, 'j': 1, 'g': 1})
```

ChainMap

A 'ChainMap' groups multiple dictionaries or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.
A 'ChainMap' class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.
A ChainMap basically encapsulates many dictionaries into a single unit and returns a list of dictionaries.

```
from collections import ChainMap
d1 = {'JS' : 'ReactJS','TS' : 'ReactTS'}
d2 = {'JS' : 'DosJS','TS' : 'DosTS'}
d3 = {'JS' : 'KarJS','TS' : 'KarTS'}
```

Output :

```
ChainMap({'JS': 'ReactJS', 'TS': 'ReactTS'}, {'JS': 'DosJS', 'TS': 'DosTS'}, {'JS': 'KarJS', 'TS': 'KarTS'})
```

NamedTuple()

A NamedTuple is a function for tuples with **Named Fields** and can be seen as an extension of the built-in tuple data type. Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code and return a tuple object with names for each position which the ordinary tuples lack. For example, consider a tuple names of books where the first element represents the book's name, second represents the author's name and the third element represents the number of pages in the book. Suppose for calling book name instead of remembering the index position you can actually call the element by using the name argument, then it will be really easy for accessing tuples element. This functionality is provided by the NamedTuple.

```
# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()
Book = namedtuple('Book',['name','author','pages'])

# Adding values
book = Book('Clean Code','Robert Cecil Martin','431')

# Access using index
print ("Getting the Book's author using index : ",end ="")
print (book[1])

# Access using name
print ("The Book's name using keyname : ",end ="")
print (book.name)
```

Output:

```
Getting the Book's author using index : Robert Cecil Martin
The Book's name using keyname : Clean Code
```

Deque

```
# Python code to demonstrate working of append(), appendleft()

from collections import deque

# initializing deque
de = deque([10,9,8])

# using append() to insert element at right end inserts 'A' at the end of deque
de.append("A")

# printing modified deque
print ("The deque after appending at right is : ")
print (de)

# using appendleft() to insert element at right end inserts 6 at the beginning of deque
de.appendleft(6)

# printing modified deque
print ("The deque after appending at left is : ")
print (de)
```

Output:

```
The deque after appending at right is :
deque([10, 9, 8, 'A'])
The deque after appending at left is :
deque([6, 10, 9, 8, 'A'])
```

UserDict

This class simulates a dictionary. The instance's (contents) are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If 'initialdata' is provided (which is the first parameter that a UserDict expects/takes), data is initialized with its contents; note that a reference to 'initialdata' will not be kept, allowing it to be used for other purposes.
UserDict is a dictionary-like container that acts as a wrapper around the dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from dict; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute. This container is used when someone wants to create their own dictionary with some modified or new functionality.

```
# Python program to demonstrate userdict

from collections import UserDict
```

```
# Creating a Dictionary where deletion is not allowed
class MyDict(UserDict):

        # Function to stop deletion from dictionary
        def append(self, val):
            self.data.update(val)

        def __del__(self):
                raise RuntimeError("Cannot Delete elements")

        # Function to stop pop from dictionary
        def pop(self, s = None):
                raise RuntimeError("Cannot Delete elements")

        # Function to stop popitem from Dictionary
        def popitem(self, s = None):
                raise RuntimeError("Cannot Delete elements")

# Driver's code
d = MyDict({'One':1,'Two': 2,'Three': 3})

print(d.data)
```

```
d.append({'Four': 4})

print(d)

d.pop('One')
```

Output:

```
{'One': 1, 'Three': 3, 'Two': 2}
{'One': 1, 'Three': 3, 'Two': 2, 'Four': 4}
Traceback (most recent call last):
  File "main.py", line 47, in <module>
    d.pop('One')
  File "main.py", line 31, in pop
    raise RuntimeError("Cannot Delete elements")
RuntimeError: Cannot Delete elements
Exception ignored in: <bound method MyDict.__del__ of {'One': 1, 'Three': 3, 'Two': 2, '
Traceback (most recent call last):
  File "main.py", line 26, in __del__
RuntimeError: Cannot Delete elements
```

UserList

The UserList class simulates a list in which the instance's contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance's contents are initially set to a copy of the list, defaulting to the empty list []. The list can be iterable, for example, a real Python list or a UserList object. UserList is a list-like container that acts as a wrapper around the list objects. This is useful when someone wants to create their own list with some modified or additional functionality.

```
# Creating a List where deletion is not allowed
class MyList(UserList):

  # Function to stop deletion from List
  def remove(self, s = None):
      raise RuntimeError("Deletion is not allowed")

  # Function to stop pop from List
  def pop(self, s = None):
      raise RuntimeError("Deletion is not allowed")

# Driver's code
L = MyList(['A', 'B', 'C', 'D'])

print("Original List")

# Inserting to List o)
print(L)

# Deliting From List
L.remove()
d.pop('B')
```

Output:

```
Original List
After Insertion
['A', 'B', 'C', 'D', 'E']
Traceback (most recent call last):
  File "main.py", line 39, in <module>
    L.remove()
  File "main.py", line 22, in remove
    raise RuntimeError("Deletion is not allowed")
RuntimeError: Deletion is not allowed
```

UserString

User string is a string-like container and just like UserDict and UserList, it acts as a wrapper around string objects. It is used when someone wants to create their own strings with some modified or additional functionality.

```
from collections import UserString


# Creating a Mutable String
class Mystring(UserString):

    # Function to append to
    # string
    def append(self, s):
        self.data += s

    # Function to rmeove from
    # string
    def remove(self, s):
        self.data = self.data.replace(s, "")


# Driver's code
s1 = Mystring("Coding")
print("Original String:", s1.data)

# Appending to string
s1.append("n")
print("String After Appending:", s1.data)

# Removing from string
s1.remove("g")
print("String after Removing:", s1.data)
```

Output:

```
Original String: Coding
String After Appending: Codingn
String after Removing: Codinn
```