# Exercise: Objects and Classes

Problems for exercises and homework for the [“Python Fundamentals” course @ SoftUni](https://softuni.bg/opencourses/python-fundamentals-course).

Check your solutions here:<https://judge.softuni.bg/Contests/950>.

## Distance Between Points

Write a method to calculate the distance between two points **p1** {**x1**, **y1**} and **p2** {**x2**, **y2**}. Write a program to read **two points** (given as two integers) and print the **Euclidean distance** between them.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 4  6 8 | 5.000 |
| 3 4  5 4 | 2.000 |
| 8 -2  -1 5 | 11.402 |

### Hints

* Create a **class** Point holding properties X and Y.
* Write a method CalcDistance(p1, p2) that returns the distance between the given points – a number.
* Use [this formula](http://www.cut-the-knot.org/pythagoras/DistanceFormula.shtml) to calculate the distance between two points. How it works?
  + Let's have two points **p1** {**x1**, **y1**} and **p2** {**x2**, **y2**}
  + Draw a right-angled triangle
  + Side **a = |x1 - x2|**
  + Side **b = |y1 - y2|**
  + Distance == side **c** (hypotenuse)
  + **c2** = **a2** + **b2** (Pythagorean theorem)
  + Distance = **c** =



* You can use [**math.sqrt(number)**](https://msdn.microsoft.com/en-us/library/system.math.sqrt(v=vs.90).aspx) method for calculating a square root.
* Print the distance **formatted to the** **3rd decimal point**.

from math import sqrt  
  
class Point:  
 def \_\_init\_\_(self,x, y):  
 self.x = x  
 self.y = y  
  
def calc\_distance(a,b):  
 distance = sqrt(a\*\*2 + b\*\*2)  
 return distance  
  
data = input()  
x1 = float(data.split()[0])  
y1 = float(data.split()[1])  
  
data1 = input()  
x2 = float(data1.split()[0])  
y2 = float(data1.split()[1])  
  
first\_point = Point(x1,y1)  
second\_point = Point(x2,y2)  
  
side\_a = abs(first\_point.x-second\_point.x)  
side\_b = abs(first\_point.y-second\_point.y)  
  
print(f"{calc\_distance(side\_a,side\_b):.3f}")

## Closest Two Points

Write a program to read **n** points and find the **closest two** of them.

### Input

The **input** holds the number of points n and n lines, each holding a point {X and Y coordinate}.

### Output

* The **output** holds the shortest distance and the closest two points.
* If several pairs of points are equally close, print **the first** of them (from top to bottom).

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visualization** | **Comments** |
| 4  3 4  6 8  2 5  -1 3 | 1.414  (3, 4)  (2, 5) |  | The closest two points are **{3, 4}** and **{2, 5}** at distance 1.4142135623731 ≈ **1.414**. |
| 3  12 -30  6 18  6 18 | 0.000  (6, 18)  (6, 18) |  | Two of the points have the same coordinates **{6, 18}**, so the distance between them is **0**. |
| 3  1 1  2 2  3 3 | 1.414  (1, 1)  (2, 2) |  | The pairs of points {{1, 1}, {2, 2}} and {{2,2}, {3,3}} stay at the same distance, but the first pair is {**{1, 1}**, **{2, 2}**}. The distance between them is 1.4142135623731 ≈ **1.414**. |

### Hints

* Use the **class** Point you created in the previous task.
* Create an array points that will keep all points.
* Create a method find\_closest\_points(points) that will check distance **between every two pairs** from the array of points and returns the two closest points in a new array.
* Print the **closest distance** and the **coordinates** of the two closest points.

from math import sqrt  
  
class Point:  
 def \_\_init\_\_(self,x, y):  
 self.x = x  
 self.y = y  
  
 def calc\_distance(self,p2):  
 a = p2.x - self.x  
 b = p2.y - self.y  
 return sqrt(a\*\*2 + b\*\*2)  
  
n = int(input())  
list\_points = []  
  
for i in range(1, n+1):  
  
 data = list(map(int, input().split()))  
 x1 = data[0]  
 y1 = data[1]  
 point = Point(x1,y1)  
  
 list\_points.append(point)  
  
min\_distance = list\_points[0].calc\_distance(list\_points[1])  
first\_point = list\_points[0]  
second\_point = list\_points[1]  
  
for i in list\_points:  
 for j in list\_points:  
 current\_distance = i.calc\_distance(j)  
 if list\_points.index(i) != list\_points.index(j) and current\_distance < min\_distance:  
 min\_distance = current\_distance  
 first\_point = i  
 second\_point = j  
  
print(f"{min\_distance:.3f}")  
print(f"({first\_point.x}, {first\_point.y})")  
print(f"({second\_point.x}, {second\_point.y})")

## Rectangle Position

Write a program to **read two rectangles** {left, top, width, height} and print whether the first is inside the second.

The input is given as two lines, each holding a rectangle, described by 4 integers: **left**, **top**, **width** and **height**.

### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Visualization** | **Comments** |
| 4 -3 6 4  2 -3 10 6 | Inside |  | The first rectangle stays **inside** the second. |
| 2 -3 10 6  4 -5 6 10 | Not inside |  | The rectangles intersect, no the first is **not insid**e the second. |

### Hints

* Create a class Rectangle holding properties Top, Left, Width and Height.
* Define calculated properties Right and Bottom.
* Define a method is\_inside(rectangle). A rectangle r1 is inside another rectangle r2 when:
  + r1.left ≥ r2.left
  + r1.right ≤ r2.right
  + r1.top ≤ r2.top
  + r1.bottom ≤ r2.bottom
* Create a method to **read** a Rectangle.
* Combine all methods into a single program.

class Rectangle:  
 def \_\_init\_\_(self, left, top, width, height):  
 self.left = left  
 self.top = top  
 self.width = width  
 self.height = height  
  
data1 = list(map(float, input().split()))  
data2 = list(map(float, input().split()))  
  
left1 = data1[0]  
top1 = data1[1]  
width1 = data1[2]  
height1 = data1[3]  
  
left2 = data2[0]  
top2 = data2[1]  
width2 = data2[2]  
height2 = data2[3]  
  
rectangle1 = Rectangle(left1,top1,width1,height1)  
rectangle2 = Rectangle(left2,top2,width2,height2)  
  
if rectangle1.left >= rectangle2.left and rectangle1.top >= rectangle2.top \  
 and rectangle1.width + rectangle1.left <= rectangle2.width + rectangle2.left \  
 and rectangle1.height + rectangle1.top <= rectangle2.height + rectangle2.top:  
 print(f"Inside")  
else:  
 print(f"Not inside")

## Exercises

Exercises are fun … Especially when they represent a problem from your exercises.

Implement a **class Exercise**, which has a **topic** (**string**), a **course\_name** (**string**), a **judge\_contest\_link** (**string**), and **problems** (**collection** of **strings**).

You will receive several input lines containing information about a single exercise in the following format:

{topic} -> {course\_name} -> {judge\_contest\_link} -> {problem1}, {problem2}. . .

You need to store every exercise in a **Collection** of **Exercises**. When you receive the command “go go go”, you end the input sequence.

You must print every exercise, in the following format:

“Exercises: {topic}

Problems for exercises and homework for the "{course\_name}" course @ SoftUni.

Check your solutions here: {judge\_contest\_link}

1. {problem1}

2. {problem2}

. . .”

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ObjectsAndSimpleClasses -> ProgrammingFundamentalsExtended -> https://judge.softuni.bg/Contests/439 -> Exercises, OptimizedBankingSystem, Animals, Websites, Boxes, BoxIntersection, Messages  go go go | Exercises: ObjectsAndSimpleClasses  Problems for exercises and homework for the "ProgrammingFundamentalsExtended" course @ SoftUni.  Check your solutions here: https://judge.softuni.bg/Contests/439  1. Exercises  2. OptimizedBankingSystem  3. Animals  4. Websites  5. Boxes  6. BoxIntersection  7. Messages |

class Exercise:  
 def \_\_init\_\_(self,course, topic,link, problems):  
 self.course = course  
 self.topic = topic  
 self.link = link  
 self.problems = problems  
  
my\_list = []  
  
data = input()  
  
while data != 'go go go':  
  
 args = data.split(" -> ")  
 course\_name = args[0]  
 topic\_name = args[1]  
 course\_link = args[2]  
 list\_problems = args[3].split(", ")  
  
 exercise = Exercise(course\_name,topic\_name,course\_link,list\_problems)  
 my\_list.append(exercise)  
 data = input()  
  
for item in my\_list:  
 print(f"Exercises: {item.course}")  
 print(f'Problems for exercises and homework for the "{item.topic}" course @ SoftUni.')  
 print(f"Check your solutions here: {item.link}")  
  
 for j in range(0, len(item.problems)):  
 print(f"{j+1}. {item.problems[j]}")

## Optimized Banking System

Create a **class** **BankAccount** which has a **Name** (**string**), **Bank** (**string**) and **Balance** (**decimal**).

You will receive several input lines, containing information in the following way:

{bank} | {accountName} | {accountBalance}

You need to store every given Account. When you receive the command “**end**” you must **stop** the input sequence.

Then you must print all **Accounts**, **ordered** by their **balance**, in **descending order**, and then by **length** of the **bank name**, in **ascending order**.

The accounts must be printed in the following way “{accountName} -> {balance} ({bank})”.

**Note**: **Numbers** must be printed rounded to the **2nd** decimal digit.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| DSK | Ivan | 504.403  DSK | Pesho | 2000.4031  DSK | Aleksander | 20000.0001  Piraeus | Ivan | 504.403  Piraeus | Aleksander | 20000.0001  end | Aleksander -> 20000.00 (DSK)  Aleksander -> 20000.00 (Piraeus)  Pesho -> 2000.40 (DSK)  Ivan -> 504.40 (DSK)  Ivan -> 504.40 (Piraeus) |

import operator  
class BankAccount:  
 def \_\_init\_\_(self,bank, name, balance):  
 self.bank = bank  
 self.name = name  
 self.balance = balance  
my\_list = []  
data = input()  
  
while data != "end":  
 args = data.split(" | ")  
 bank\_name = args[0]  
 cust\_name = args[1]  
 money = float(args[2])  
  
 my\_account = BankAccount(bank\_name, cust\_name, money)  
 my\_list.append(my\_account)  
 data = input()  
  
sorted\_bank = sorted(my\_list, key=operator.attrgetter("bank"))  
sorted\_list = sorted(sorted\_bank, key=operator.attrgetter("balance"), reverse=True)  
  
for item in sorted\_list:  
 print(f"{item.name} -> {item.balance:.2f} ({item.bank})")

## Animals \*

You have been given the task to create classes for several sophisticated animals.

Create a **class Dog** which has a **name** (**string**), **age** (**int**) and **number\_of\_legs** (**int**).

Create a **class Cat** which has a **name** (**string**), **age** (**int**) and **intelligence\_quotient** (**int**).

Create a **class Snake** which has a **name** (**string**), **age**(**int**) and **cruelty\_coefficient** (**int**).

Create a **method** in **each class** which is called produce\_sound(). The method should print on the console a string depending on the class:

* If it’s a **Dog**, you should print “**I'm a Distinguishedog, and I will now produce a distinguished sound! Bau Bau.**”
* It it’s a **Cat**, you should print “**I'm an Aristocat, and I will now produce an aristocratic sound! Myau Myau.**”
* If it’s a **Snake**, you should print “**I'm a Sophistisnake, and I will now produce a sophisticated sound! Honey, I'm home.**”

Now for the real deal. You will receive several input commands, which will register animals or make them produce sounds, until you receive the command “**I’m your Huckleberry**”.

The commands will be in the following format:

{class} {name} {age} {parameter}

The **class** will be either “**Dog**”, “**Cat**” or “**Snake**”. The **name** will be a simple **string**, which can contain **any** ASCII character BUT **space**. The **age** will be an **integer**. The **parameter**, will be an **integer**. **Depending** on the **class** it would either be **number of legs**, **IQ**, or **cruelty coefficient**.

Register each animal, and keep them in **collections**, by your **choice**, so that you can **ACCESS THEM BY NAME**. You will most likely need 3 collections, to store the different animals inside them.

Between the register commands you might receive a command in the following format:

talk {name}

You must then make the **animal** with the **given name**, **produce a sound**.

When you receive the ending command, you should print every animal in the following format:

* If it’s a **Dog**, you should print “**Dog: {name}, Age: {age}, Number Of Legs: {numberOfLegs}**”
* It it’s a **Cat**, you should print “**Cat: {name}, Age: {age}, IQ: {intelligenceQuotient}**”
* If it’s a **Snake**, you should print “**Snake: {name}, Age: {age},** **Cruelty: {crueltyCoefficient}**”

Print first the **Dogs**, then the **Cats**, and lastly – **The Snakes**.

### Constraints

* You can assume that there will be **no duplicate** names (**even** in **different animals**).
* All input data will be **valid**. There will be **no invalid** input lines.
* The **name** in the talk command, will **always** be **existent**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Dog Sharo 3 4  Cat Garfield 5 200  Snake Alex 25 1000  talk Sharo  talk Garfield  talk Alex  I'm your Huckleberry | I'm a Distinguishedog, and I will now produce a distinguished sound! Bau Bau.  I'm an Aristocat, and I will now produce an aristocratic sound! Myau Myau.  I'm a Sophistisnake, and I will now produce a sophisticated sound! Honey, I'm home.  Dog: Sharo, Age: 3, Number Of Legs: 4  Cat: Garfield, Age: 5, IQ: 200  Snake: Alex, Age: 25, Cruelty: 1000 |
| Dog Bau 5 10  Cat Myau 5 100  Dog Georgi 20 1000  Cat Bojo 4 20  talk Bojo  I'm your Huckleberry | I'm an Aristocat, and I will now produce an aristocratic sound! Myau Myau.  Dog: Bau, Age: 5, Number Of Legs: 10  Dog: Georgi, Age: 20, Number Of Legs: 1000  Cat: Myau, Age: 5, IQ: 100  Cat: Bojo, Age: 4, IQ: 20 |

class Dog:  
 def \_\_init\_\_(self,name, age, legs):  
 self.name = name  
 self.age = age  
 self.legs = legs  
  
 def sound\_dog(self):  
 sound\_dog = "I'm a Distinguishedog, and I will now produce a distinguished sound! Bau Bau."  
 return sound\_dog  
  
class Cat:  
 def \_\_init\_\_(self,name, age, intelligence):  
 self.name = name  
 self.age = age  
 self.intelligence = intelligence  
  
 def sound\_cat(self):  
 sound\_cat = "I'm an Aristocat, and I will now produce an aristocratic sound! Myau Myau."  
 return sound\_cat  
  
class Snake:  
 def \_\_init\_\_(self,name, age, cruelty):  
 self.name = name  
 self.age = age  
 self.cruelty = cruelty  
  
 def sound\_snake(self):  
 sound = "I'm a Sophistisnake, and I will now produce a sophisticated sound! Honey, I'm home."  
 return sound  
  
def is\_in\_there(a,animal):  
 for i in a:  
 if animal == i.name:  
 return True  
 return False  
  
data = input()  
  
list\_cat = []  
list\_dog = []  
list\_snake = []  
  
while data != "I'm your Huckleberry":  
 args = data.split()  
 if args[0] == "Dog":  
 list\_dog.append(Dog(args[1],args[2],args[3]))  
 elif args[0] == "Cat":  
 list\_cat.append(Cat(args[1], args[2], args[3]))  
 elif args[0] == "Snake":  
 list\_snake.append(Snake(args[1], args[2], args[3]))  
 elif args[0] == "talk":  
 if is\_in\_there(list\_cat, args[1]):  
 print(Cat.sound\_cat(args[1]))  
 elif is\_in\_there(list\_dog, args[1]):  
 print(Dog.sound\_dog(args[1]))  
 elif is\_in\_there(list\_snake, args[1]):  
 print(Snake.sound\_snake(args[1]))  
 data = input()  
  
if len(list\_dog) > 0:  
 for animal in list\_dog:  
 print(f"Dog: {animal.name}, Age: {animal.age}, Number Of Legs: {animal.legs}")  
if len(list\_cat) > 0:  
 for animal in list\_cat:  
 print(f"Cat: {animal.name}, Age: {animal.age}, IQ: {animal.intelligence}")  
if len(list\_snake) > 0:  
 for animal in list\_snake:  
 print(f"Snake: {animal.name}, Age: {animal.age}, Cruelty: {animal.cruelty}")

## my resolution:

class Dog:  
 def \_\_init\_\_(self, name, age, legs):  
 self.name = name  
 self.age = age  
 self.legs = legs  
  
class Cat:  
 def \_\_init\_\_(self, name, age, intelligence):  
 self.name = name  
 self.age = age  
 self.intelligence = intelligence  
  
class Snake:  
 def \_\_init\_\_(self, name, age, cruelty):  
 self.name = name  
 self.age = age  
 self.cruelty = cruelty  
  
data = input()  
dog\_list = []  
cat\_list = []  
snake\_list = []  
while data != "I'm your Huckleberry":  
 args = data.split(" ")  
 if args[0] != "talk":  
 animal = args[0]  
 animal\_name = args[1]  
 animal\_age = args[2]  
 animal\_parameter = args[3]  
  
 if animal == "Dog":  
 my\_dog = Dog(animal\_name,animal\_age,animal\_parameter)  
 dog\_list.append(my\_dog)  
 elif animal == "Cat":  
 my\_cat = Cat(animal\_name, animal\_age, animal\_parameter)  
 cat\_list.append(my\_cat)  
 elif animal == "Snake":  
 my\_snake = Snake(animal\_name,animal\_age, animal\_parameter)  
 snake\_list.append(my\_snake)  
 else:  
 for item in dog\_list:  
 if args[1] == item.name:  
 print(f"I'm a Distinguishedog, and I will now produce a distinguished sound! Bau Bau.")  
 for item in cat\_list:  
 if args[1] ==item.name:  
 print(f"I'm an Aristocat, and I will now produce an aristocratic sound! Myau Myau.")  
 for item in snake\_list:  
 if args[1] == item.name:  
 print(f"I'm a Sophistisnake, and I will now produce a sophisticated sound! Honey, I'm home.")  
  
 data = input()  
  
for item in dog\_list:  
 print(f"Dog: {item.name}, Age: {item.age}, Number Of Legs: {item.legs}")  
for item in cat\_list:  
 print(f"Cat: {item.name}, Age: {item.age}, IQ: {item.intelligence}")  
for item in snake\_list:  
 print(f"Snake: {item.name}, Age: {item.age}, Cruelty: {item.cruelty}")

## 7. Websites

You have been tasked to create an ordered database of websites. For the task you will need to create a **class** **Website**, which will have a **Host**, a **Domain** and **Queries**.

The **Host** and the **Domain** are simple **strings**.   
The **Queries**, is **Collections** of **strings**.

You will be given several input lines in the following format:

{host} | {domain} | {query1,query2. . .}

Note: There will **always** be a **host** and a **domain**, but there **might** **NOT** be **ANY** **queries**.

The input sequence ends, when you receive the command “**end**”. Then you must print **all websites** in the following format:

https://www.{host}.{domain}/query?=[{query1]&[{query2}]&[query3]. . .

In case there are **NO** **queries**, just print:

https://www.{host}.{domain}

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| softuni | bg | user,course,homework  judge.softuni | bg | contest,bg  google | bg | search,query  zamunda | net  end | https://www.softuni.bg/query?=[user]&[course]&[homework]  https://www.judge.softuni.bg/query?=[contest]&[bg]  https://www.google.bg/query?=[search]&[query]  https://www.zamunda.net |

class Website:  
 def \_\_init\_\_(self,host, domain, queries=None):  
 self.host = host  
 self.domain = domain  
 self.queries = queries if queries is not None else False  
  
data = input()  
  
web\_list = []  
  
while data != "end":  
 args = data.split(" | ")  
 host\_name = args[0]  
 domain\_name = args[1]  
 if len(args) > 2:  
 list\_queries = args[2].split(",")  
 my\_webs = Website(host\_name,domain\_name,list\_queries)  
 else:  
 my\_webs = Website(host\_name,domain\_name)  
  
 web\_list.append(my\_webs)  
 data = input()  
  
for web in web\_list:  
 if not web.queries:  
 print(f"https://www.{web.host}.{web.domain}")  
 else:  
 q = "]&[".join(web.queries)  
 print(f"https://www.{web.host}.{web.domain}/query?=[{q}]")

## Boxes

Create a **class Box**, which will represent a **rectangular** **box**. The Box should have **UpperLeft** (**Point**), **UpperRight** (**Point**), **BottomLeft** (**Point**), **BottomRight** (**Point**).

Create, or use from the Lab, the **class Point** which has **X** (**int**) and **Y** (**int**) – coordinates in **2D space**. Move the CalculateDistance() method in the **Point class**, exactly as it is. Then use “Point.CalculateDistance(point1, point2)” signature, to **use** the **method**.

Create **2 methods** in the **Box class**:  
CalculatePerimeter(width, height)  
CalculateArea(width, height).

Make them **return** **integers**, representing the **perimeter** and **area** of the **box**.

The formulas are respectively – (2 \* Width + 2 \* Height) and (Width \* Height).

The **Width** is the **distance** **between** the **UpperLeft** and the **UpperRight** Points, and **ALSO** – the **Bottomleft** and the **BottomRight** Points.

The **Height** is the **distance** **between** the **UpperLeft** and the **BottomLeft** Points, and **ALSO** – the **UpperRight** and the **BottomRight** Points.

You will receive several input lines in the following format:

{X1}:{Y1} | {X2}:{Y2} | {X3}:{Y3} | {X4}:{Y4}

Those will be the coordinates to **UpperLeft**, **UpperRight**, **BottomLeft** and **BottomRight** (**IN THE SAME ORDER**).

When you receive the command “**end**”. You must print **all Boxes** in the following format:

“Box: {width}, {height}

Perimeter: {perimeter}

Area: {area}”

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 0:2 | 2:2 | 0:0 | 2:0  -3:0 | 0:0 | -3:-3 | 0:-3  -2:2 | 2:2 | -2:-2 | 2:-2  end | Box: 2, 2  Perimeter: 8  Area: 4  Box: 3, 3  Perimeter: 12  Area: 9  Box: 4, 4  Perimeter: 16  Area: 16 |

import math  
  
class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def calculate\_distance(self,other\_point):  
 x = abs(self.x - other\_point.x)  
 y = abs(self.y - other\_point.y)  
 return math.sqrt(x \*\* 2 + y \*\* 2)  
  
class Box:  
 def \_\_init\_\_(self,upper\_left, upper\_right, bottom\_left, bottom\_right):  
 self.upper\_left = upper\_left  
 self.upper\_right= upper\_right  
 self.bottom\_left = bottom\_left  
 self.bottom\_right = bottom\_right  
 self.width = Point.calculate\_distance(self.upper\_left, self.upper\_right)  
 self.height = Point.calculate\_distance(self.upper\_left, self.bottom\_left)  
 self.perimeter = self.width \* 2 + self.height \* 2  
 self.area = self.width \* self.height  
  
data = input()  
  
list\_boxes = []  
  
while data != "end":  
 u\_l = data.split(" | ")[0]  
 u\_r = data.split(" | ")[1]  
 b\_l = data.split(" | ")[2]  
 b\_r = data.split(" | ")[3]  
  
 u\_l\_p = Point(int(u\_l.split(":")[0]), int(u\_l.split(":")[1]))  
 u\_r\_p = Point(int(u\_r.split(":")[0]), int(u\_r.split(":")[1]))  
 b\_l\_p = Point(int(b\_l.split(":")[0]), int(b\_l.split(":")[1]))  
 b\_r\_p = Point(int(b\_r.split(":")[0]), int(b\_r.split(":")[1]))  
  
 my\_box = Box(u\_l\_p, u\_r\_p, b\_l\_p, b\_r\_p)  
 list\_boxes.append(my\_box)  
  
 data = input()  
  
for box in list\_boxes:  
 print(f"Box: {int(box.width)}, {int(box.height)}")  
 print(f"Perimeter: {int(box.perimeter)}")  
 print(f"Area: {int(box.area)}")

## Messages \*

Create a **class** **User**, which has a **Username** (**string**), and **ReceivedMessages** (**Collection** of **Messages)**.   
Create a **class** **Message**, which has a **Content** (**string**) and a **Sender** (**User**).

You will have to store a messaging history for every user. The input consists of 2 commands:

“register {username}”

“{senderUsername} send {recipientUsername} {content}”

The **register command**, registers a **user** with the **given username**.

The **send command**, sends a **message**, from the **given sender**, to the **given recipient**, with the **given content**. That means that you must **add** the **message** to the **recipient’s ReceivedMessages**.  
If **even one** of the **given names** does **NOT** exist, **ignore** the command.

When you receive the command “**exit**” you must end the input sequence. After that you will receive **2 usernames**, **separated** by a **space**.

You must **print all messages**, sent, **between the two users**, corresponding to **the given usernames**. The messages should be printed in a specified way. You should print **first** a **message** **SENT** from the **first user**, **then** a **message** **SENT** from the **second user**, then a **message** from **the first user**, and **so** **on**. If one of the **collections** of **messages** has **more** **elements** than the **other**, just print the **remaining elements** from it.

The first user’s messages must be printed in the following way:  
“{firstUser}: {content}”

The second user’s message must be printed in the following way:

“{content} :{secondUser}”

When you print the whole output, it should look like this:

{firstUser}: {content1}

{content1} :{secondUser}

{firstUser}: {content2}

{content2} :{secondUser}

. . .

In case there are **NO** messages **between** the two users, print “**No messages**”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| register Ivan  register Pesho  Ivan send Pesho pesho  Ivan send Pesho pesho\_tam\_li\_si?  Pesho send Ivan kaji\_vanka  Pesho send Ivan tuk\_sum  Pesho send Ivan chakai\_che\_bachkam  Ivan send Pesho kvo\_stava  Ivan send Pesho kak\_si  Ivan send Pesho deka\_izbega\_be?  Ivan send Pesho pecaaa!!!  exit  Ivan Pesho | Ivan: pesho  kaji\_vanka :Pesho  Ivan: pesho\_tam\_li\_si?  tuk\_sum :Pesho  Ivan: kvo\_stava  chakai\_che\_bachkam :Pesho  Ivan: kak\_si  Ivan: deka\_izbega\_be?  Ivan: pecaaa!!! |
| register John  John send Harry harry\_you\_there?  register Harry  John send Harry harry?  register Donald  Harry send John yeah\_sorry\_was\_out...  Harry send John wassup?  Donald send John Yo\_John?  Donald send Jonh You\_there?  John send Harry thank\_god!!  John send Harry I\_need\_you!  exit  John Harry | John: harry?  yeah\_sorry\_was\_out... :Harry  John: thank\_god!!  wassup? :Harry  John: I\_need\_you! |

class User:  
 def \_\_init\_\_(self,username):  
 self.username = username  
 self.received\_msg = []  
  
class Message:  
 def \_\_init\_\_(self,content,user):  
 self.content = content  
 self.user = user  
  
users = []  
while True:  
 #Read input  
 user\_input = input()  
  
 #Check if exit  
 if user\_input == "exit":  
 break  
  
 #Split input by space  
 user\_input\_args = user\_input.split(" ")  
  
 #if input is for registering  
 if user\_input\_args[0]=="register":  
 #make new User with username  
 user = User(user\_input\_args[1])  
 #Add username to users list  
 users.append(user)  
 else:  
 sender = user\_input\_args[0]  
 receiver = user\_input\_args[2]  
 #If does not have registration for receiver or sender  
 receiver\_exist = False  
 sender\_exist = False  
  
  
 for u in users:  
 #Check if sender exists  
 if u.username == sender:  
 sender\_exist = True  
 #Check if receiver exists  
 if u.username == receiver:  
 receiver\_exist = True  
  
 #Skip if someone not exists  
 if not receiver\_exist or not sender\_exist:  
 continue  
  
 #Create new message  
 content = user\_input\_args[3]  
 message = Message(content,sender)  
  
 #Find the user who received message and add to his list of messages  
 user\_receiver = list(filter(lambda u : u.username == sender,users))[0]  
 user\_receiver.received\_msg.append(message)  
  
#Read users for report  
users\_input = input().split()  
first = users\_input[0]  
second = users\_input[1]  
  
#Find users from the list of users  
first\_user = list(filter(lambda u : u.username == first,users))[0]  
second\_user = list(filter(lambda u : u.username == second,users))[0]  
  
#Get users messages lists  
first\_user\_messages = first\_user.received\_msg  
second\_user\_messages = second\_user.received\_msg  
  
#Get biggest list count  
steps = max(len(first\_user.received\_msg),len(second\_user.received\_msg))  
  
#If no messages - print  
if steps == 0:  
 print("No messages")  
  
  
for i in range(steps):  
 #If more messages print first  
 if len(first\_user\_messages) > 0:  
 msg = first\_user\_messages.pop(0)  
 print(f"{first\_user.username}: {msg.content}")  
 # If more messages print first  
 if len(second\_user\_messages) > 0:  
 msg = second\_user\_messages.pop(0)  
 print(f"{msg.content} :{second\_user.username}")