

# Informatics for complete beginners

Gavin Sinclair

Version 0.1.1  
22 June 2021





# Contents

<b>Introduction</b>	<b>5</b>
<b>1 Operating on a fixed set of numbers</b>	<b>7</b>
101 Classify a triangle (worked example) . . . . .	8
102 Calculate a gradient (worked example) . . . . .	10
103 Who is the tallest (1)? (strongly hinted) . . . . .	11
104 The cheapest TV . . . . .	13
105 Shopping (1) . . . . .	14
106 Jogging (1) . . . . .	15
107 Who is the tallest (2)? (worked example) . . . . .	16
108 A fair wage (1) . . . . .	17
<b>2 Looping through data</b>	<b>19</b>
201 Who is the tallest? (3) (worked example) . . . . .	20
202 Shopping (2) . . . . .	22
203 Scrabble tally . . . . .	24
204 Area calculator . . . . .	25
205 Drought . . . . .	27
206 Cute numbers . . . . .	28
207 High-wire walk . . . . .	29
<b>3 Looping to solve problems</b>	<b>31</b>
301 How many factors? (worked example) . . . . .	34
302 Jogging (2) . . . . .	36
<b>4 Operating on a list of numbers</b>	<b>37</b>
401 Shopping (3) . . . . .	38
402 A fair wage (2) . . . . .	39
403 Addition carry . . . . .	40
<b>5 Miscellaneous problems</b>	<b>41</b>
501 Name, age and hourly rate (worked example) . . . . .	41
502 Golf (1) . . . . .	43
<b>6 Two-dimensional data</b>	<b>45</b>
601 Cash grab (1) (worked example) . . . . .	46
602 Cash grab (2) . . . . .	49
603 Cash grab (3) . . . . .	50
604 Golf (2) . . . . .	51



# Introduction

If you are a complete beginner at Informatics, then the examples and exercises in this book aim to bring you up to the level of “relative beginner”. This may seem like a modest aim, but it is not. It is like setting a car in motion: it takes a lot of engine power to get a stationary car moving, but not so much to keep it moving. You will need considerable brainpower, focus and effort to take the first steps in Informatics, but once you have a reasonable idea of what’s going on, you will be able to direct your own progress. There will still be a lot to learn, but you won’t be weighed down by confusion over the basics.

The rest of this introduction is as brief as possible so you can get started!

**What is Informatics?** At the simplest level, it is a form of computer programming dealing with *information*. A problem is stated, and you write a program that will solve that problem *given some information*. Here are some examples from this book:

- Given the heights of four people, determine the height of the tallest person.
- Given an arbitrary number of quantity-price pairs, determine the total amount spent.
- Given a grid of squares containing cash amounts, find the location that has the highest total in the surrounding squares.

These are all simple problems because this is a book that focuses on solving simple problems. When you enter Informatics competitions you will be exposed to much harder problems, akin to finding the shortest path through a maze or solving a Rubik’s Cube.

**How to use this book** Each chapter contains some teaching material, one or two fully worked examples, and some exercises. Read the teaching material carefully and try out its code examples. All exercises, including the worked examples, are numbered, and you will use the **learninformatics** Python environment to complete the exercises and have them confirmed as correct.

**Setting up the learninformatics environment** Full instructions follow, but the information is best conveyed by a demonstration. See the video at [Ref: YouTube link](#).

While it is possible to use a Python environment on your own computer,<sup>1</sup> we prefer here to outline the simpler approach of using the website [replit.com](#).

1. Log in to your account on replit.com, or create one if you don’t already have one. If it asks for your preferred languages, choose Python, Ruby and Clojure.
2. Create a new repl<sup>2</sup> in the Python language. Title it *LearnInformatics*.
3. Create a file in your repl called `learninformatics.py`.

---

<sup>1</sup>See [Ref: Appendix A](#) for details.

<sup>2</sup>repl is an initialism for read-eval-print-loop. It really means an interactive programming environment.

4. Go to the URL footnoted,<sup>3</sup> copy all of the text, and paste it into the file you just created.
5. In `main.py`, type: `import learninformatics as l`
6. Run your code with the green Run button.
7. In the *Console*, type `l.update()` and hit Enter/Return. This will create the file `DATA.txt` in your project. You can run `l.update()` at any time to ensure you have the latest data.

You should now be ready to go.

**Using the `learninformatics` environment** You'll get the hang of it by following instructions later on, but here is a summary. Say you are working on exercise 205. You will add the following to `main.py`:<sup>4</sup>

```

1  def ex205(IN, OUT):
2      # Get input data from IN.
3      # Solve the problem.
4      # Write the answer to OUT.

```

When you *Run* your code, nothing will happen, because all you have is a bunch of definitions. But in the *Console* you can type `l.test(205)` and it will execute your code in `ex205` and provide it with the sample input data (specified in the exercise) and check that the code produces the correct output. When you are getting correct results for the sample data, you will type `l.judge(205)` to see if your code works for a wider variety of inputs.

This might sound complicated, but it's a much easier workflow than the normal Informatics practice in which each problem is solved in its own file and submitted to a website for marking.

Here are example commands and what they do:

Command	Action
<code>l.run(101)</code>	Run your function <code>ex101</code> interactively, taking input from the keyboard and sending output to the screen.
<code>l.run(example)</code>	Run your function <code>example</code> interactively.
<code>l.test(205)</code>	Run your function <code>ex205</code> , using the sample data set(s) as input and checking that your code generates the correct output.
<code>l.judge(205)</code>	Run your function <code>ex205</code> multiple times, using a variety of (secret) data sets and checking that your code generates the correct output in all cases.
<code>l.info()</code>	Show some information about the <code>learninformatics</code> environment: software version, data version, available exercises.
<code>l.update()</code>	Update to the latest available data.

---

<sup>3</sup><https://raw.githubusercontent.com/gsinclair/learninformatics/master/learninformatics.py>

<sup>4</sup>Your code will accrete many small sections like this: `ex101`, `ex102`, ...

# Chapter 1

## Operating on a fixed set of numbers

In these problems, you will read in just one number, or a list of numbers of a known size. You will calculate something or somethings based on the input and write it to output.

The example problems and solutions shown here demonstrate how to read numbers (integers and floats) from `IN` and write results to `OUT`. Note that when we write a float<sup>1</sup> to `OUT`, we must round it to the specified number of decimal places to ensure predictable output.<sup>2</sup>

The following lines of code demonstrate how to input and output integers and floats.

```
1 import learninformatics as l
2
3 def example1(IN, OUT):
4     a = int(IN.readline())
5     b = float(IN.readline())
6
7     print("a squared:", a*a, file=OUT)
8     print("b squared:", b*b, file=OUT)
9     print("b squared (2dp):", round(b*b,2), file=OUT)
```

You should enter this code, then run it, then type `l.run(example1)` in your shell. It will wait while you enter two numbers (an integer and a float) and will then immediately output the squares that the code promises.

The first two problems in this chapter are complete worked examples to show input and output in some more context and to demonstrate some decision-making. After that, problems will have fewer hints as you are expected to apply the skills learned in the earlier problems.

To assist in your learning, you should have a notebook where you write out your solution to each problem, along with notes on what you have learned. You will find that you refer to it often.

When you solve actual numbered problems, you will write functions with names like `ex101` (short for “exercise 101”). You can then run the function interactively, as you did with `example1`, but you can also do something even better: run it against test data and judging data. These ideas were explained in the Introduction.

---

<sup>1</sup>In computing, decimal numbers are generally called “floats”, short for “floating point numbers”, because the location of the decimal point can float around (consider 104.32 and 1.0432, for instance).

<sup>2</sup>If you use Python, or pretty much any programming language, to calculate  $0.1 + 0.2$  then you will get a surprising result. It’s not Python’s fault; we have to live with the fact that computers, by default, don’t handle decimal numbers particularly well.

## 101 Classify a triangle (worked example)

**Question** Read three integers representing sides of a triangle and determine whether the triangle is equilateral, isosceles, or scalene.

### Sample

IN	OUT	IN	OUT	IN	OUT
17	scalene	13	isosceles	5	equilateral
14		13		5	
13		8		5	

### Solution

```
1  def ex101(IN, OUT):
2      a = int(IN.readline())
3      b = int(IN.readline())
4      c = int(IN.readline())
5
6      if a == b and a == c:
7          answer = 'equilateral'
8      elif a == b or a == c or b == c:
9          answer = 'isosceles'
10     else:
11         answer = 'scalene'
12
13     print(answer, file=OUT)
```

**Explanation** Lines 2–4 read the three lines in IN into integer variables  $a$ ,  $b$  and  $c$ . Lines 6–11 determine the answer, and Line 13 writes the answer to OUT.

This program is very much about making decisions (`if/elif/else`) with a small amount of data. Note the use of `and` and `or` in making the decisions. `elif` is short for “else if”.

Line 1 wraps this code up in a function called `ex101`, which is what the `learninformatics` environment expects to find when you...well, see below.

### Running the code

The Introduction explained the features of the `learninformatics` environment. Now we see them for ourselves.

After you have *Run* your code in [replit.com](https://replit.com), try the following in the console.

- `l.run(101)`, then use the keyboard to enter the values 15, 15 and 13. The answer `isosceles` should immediately be displayed.
- `l.run(101, "15\n19\n20\n")`. This puts the data in directly, instead of waiting for you to type it. The answer `scalene` should immediately be displayed.
- `l.test(101)`. This runs your code against the three samples shown above and checks that the correct output is produced. If the output is incorrect, or if an error occurs, a helpful report is printed so you can try to work out what went wrong.
- `l.judge(101)`. This runs your code against a greater variety of inputs and checks for the correct output. The inputs remain secret, the information printed if something goes wrong is quite basic. If your code passes all tests, a success token is printed as a reward.

Note that the name of the function containing your solution is important. If you call it `training101` or `unicorn`, or anything other than `ex101`, then the actions above will not work.

As you proceed through the exercises, you will have many `def`s in your code, one for each exercise. This is excellent: a record of your work that you can look back on.

## 102 Calculate a gradient (worked example)

**Question** Read four floats representing points  $(x_1, y_1)$  and  $(x_2, y_2)$  and find the *gradient* of the interval  $AB$ , which is typically called  $m$  and is calculated as follows:

$$m = \frac{\text{rise}}{\text{run}} = \frac{y_2 - y_1}{x_2 - x_1}.$$

Output the gradient, rounded to three decimal places. If the gradient is undefined (because the run is zero), output `undefined`.

### Sample

IN	OUT	IN	OUT	IN	OUT
5	0.5	-2.76	-1.618	3.7	<code>undefined</code>
1		-1.01		9.5	
10		3.14159		3.7	
3.5		-10.559		17	

**Explanation** The answers given reflect the following calculations:

$$\frac{3.5 - 1}{10 - 5} = 0.5 \quad \frac{-10.559 - (-1.01)}{3.14159 - (-2.76)} = -1.692594\ldots \quad \frac{17 - 9.5}{3.7 - 3.7} = \frac{7.5}{0}$$

### Solution

```
1  def ex102(IN, OUT):
2      x1 = float(IN.readline())
3      y1 = float(IN.readline())
4      x2 = float(IN.readline())
5      y2 = float(IN.readline())
6
7      rise = y2 - y1
8      run  = x2 - x1
9
10     if run == 0.0:
11         print('undefined', file=OUT)
12     else:
13         answer = rise / run
14         print(round(answer,3), file=OUT)
```

**Explanation** Lines 2–5 read the four floats from `IN` and assign them to sensible variable names. Lines 7–8 calculate the rise and run, which are important values in determining the gradient. Lines 10–13 determine the answer, looking out for the undefined case. Line 15 outputs the answer to `OUT`, rounding it to three decimal places as required.

**Running the code** The function is named `ex102`, so you can:

- Run the code interactively with `l.run(102)`.
- Test the correctness of the code against the sample data with `l.test(102)` and against other judging data with `l.judge(102)`.

## 103 Who is the tallest (1)? (strongly hinted)

**Question** In the schoolyard you determine who is the tallest by standing back to back. Among your online friends, though, the only way to find this out is to compare numbers. There are four people in your group, and so when everyone has entered their height in centimetres, all that remains is to pick the largest number out of the list.

Your program will read four numbers from `IN` and write the largest number to `OUT`.

### Sample

IN	OUT
147	171
165	
171	
168	

**Scratch** We've seen several examples now of how to read an integer from `IN`. For this problem, we need to do it four times.

```
1     a = int(IN.readline())
2     b = int(IN.readline())
3     c = int(IN.readline())
4     d = int(IN.readline())
```

We now have all our data. All that remains is to decide which is the largest of  $a$ ,  $b$ ,  $c$  and  $d$ .

```
1     if a >= b and a >= c and a >= d:
2         ...
```

If the condition above is true, then  $a$  is the largest of the four, or at least the equal largest. What do we do in this scenario?

```
1     if a >= b and a >= c and a >= d:
2         answer = a
```

We store the value of  $a$  in a new variable called `answer`.

To the two lines of code provided above, you can add six more to make a complete determination of the largest value among  $a$ ,  $b$ ,  $c$  and  $d$ .

**Solution** A complete solution looks like this, except for the six lines you need to add.

```
1     def ex103(IN, OUT):
2         a = int(IN.readline())
3         b = int(IN.readline())
4         c = int(IN.readline())
5         d = int(IN.readline())
6
7         if a >= b and a >= c and a >= d:
8             answer = a
9             #
10            # ...six more lines...
11            #
12
13     print(answer, file=OUT)
```

If you're really stuck, you can look at the answers provided.

**Afterword** The detailed assistance provided in this example was painstaking, so this will be gradually decreased over the next few problems.

As you coded this problem, you might have had two questions.

- Is this *really* the best way to find the largest of four numbers?
- What if there were 400 numbers instead of four?

These questions will be addressed in *Who is the tallest? (2)* and *Who is the tallest? (3)* respectively.

## 104 The cheapest TV

**Question** You walk the aisles of Bing Lee, looking for a new TV. You narrow it down to five choices and decide to go with the cheapest. How much will you pay?

Your program will read five integers (representing prices in dollars) from **IN** and write the smallest number to **OUT**.

### Sample

<b>IN</b>	<b>OUT</b>
499	325
565	
325	
400	
717	

**Scratch** If you have learned the lessons of the previous exercise well, you will complete this one in no time.

## 105 Shopping (1)

**Question** You've just returned from the shops with the items you were asked to get. Your dad asked "How much was it?", but you don't remember! You have to work it out quickly. You bought exactly three *kinds* of items, but different *quantities* of each.

**Input** The input contains six lines:

- quantity 1
- price 1
- quantity 2
- price 2
- quantity 3
- price 3

The quantities are integers in the range 1–10 and the prices are positive floats less than 100 (representing \$100).

**Output** The output is a single float containing the total price, and it must be rounded to two decimal places.

### Sample

IN	OUT
6	292.64
2.50	
8	
1.75	
3	
87.88	

**Explanation** The output represents \$292.64, which is  $6(\$2.50) + 8(\$1.75) + 3(\$87.88)$ .

## 106 Jogging (1)

**Question** Sarah is about to take up jogging. She will start jogging  $D$  metres per day and increase her daily distance by  $I$  metres each day. When she reaches her target of  $T$  metres per day, she will stop the daily increase and just keep a consistent jogging pattern.

How many days after she starts will she reach her target?

### Input

The input contains three lines, all of which are values in metres:

- $D$ , the distance at which she starts jogging;
- $I$ , the amount by which she increases her jog each day;
- $T$ , her target daily jog, after which she doesn't increase it any further.

All values are integers, and you are guaranteed that  $D < D + I < T$ , that is, she will have to apply at least two increases before reaching her target.

**Output** You will write a single integer to `OUT`, that being the number of days after starting that she reaches her target.

### Sample

IN	OUT	IN	OUT
100	4	700	3
50		20	
300		750	

**Explanation** In the first sample, her running pattern over the days goes like this:

Day	Distance (m)
1	100
2	150
3	200
4	250
5	300

She reaches her target on day 5, which is four days after she starts jogging, so the answer is 4.

In the second sample, her running pattern *would* proceed 700, 720, 740, 760, but 760 m actually exceeds her target of 750 m, so she would in fact run 750 m on the fourth and subsequent days. In any case, she reached/exceeded her target on day 4, which is three days after she started, so the answer is 3.

**Scratch** It is intended that you use simple mathematics to work this out. As a very large hint, consider the expression  $(300 - 100)/50$  in the first sample. As another hint, be aware of the difference between `/` and `//` in Python. Finally, although it's not strictly necessary, you might benefit from searching for examples of `math.floor` and `math.ceil`.

## 107 Who is the tallest (2)? (worked example)

This question takes another look at *Who is the tallest? (1)*. The question is the same, but the method of solution is different.

**Question** In the schoolyard you determine who is the tallest by standing back to back. Among your online friends, though, the only way to find this out is to compare numbers. There are four people in your group, and so when everyone has entered their height in centimetres, all that remains is to pick the largest number out of the list.

Your program will read four numbers from **IN** and write the largest number to **OUT**.

### Sample

IN	OUT
147	171
165	
171	
168	

**Scratch** Just like the earlier problem, we need to read four integers from **IN**. But this time, instead of making a lot of comparisons between  $a$ ,  $b$ ,  $c$  and  $d$ , we will use Python's built-in **max** function.

If you try the following Python expressions in your console...

```
1     max(5, 1, 2, 9)
2     max(7, 2, 5, 11, 6)
3     max(-51, 73, 33, -1, 0, 12)
```

...then you will learn with some relief that Python has pretty much solved the “Who is the tallest?” problem for us.

### Solution

```
1     def ex107(IN, OUT):
2         a = int(IN.readline())
3         b = int(IN.readline())
4         c = int(IN.readline())
5         d = int(IN.readline())
6
7         answer = max(a, b, c, d)
8
9         print(answer, file=OUT)
```

**Afterword** The **max()** function can take any number of arguments, or it can find the maximum of a (potentially very long) list. These are details for later. But here are two teasers that you can run in the console and think about.

```
1     max(n*n for n in [-6, -4, -1, 0, 3])
2     max(len(x) for x in ["Emma", "Caitlin", "Tim"])
```

## 108 A fair wage (1)

**Question** There are six employees in your company all doing roughly the same kind of work. Some have worked for you for a longer time, or are generally more productive, so you pay them a bit more. But once a year you take a look to see if the overall distribution of wages seems to be fair.

The first thing you look for is the *range* of wages: the highest minus the lowest. You have a rule that the range should be less than 10% of the highest wage.

**Input** From `IN` you will read six floats, each representing a weekly wage.

**Output** To `OUT` you will write three values:

- The range, rounded to two decimal places
- The highest wage, rounded to two decimal places
- `yes` or `no` according to whether your wages are fair

### Sample

IN	OUT	IN	OUT
535.00	62.5	535.00	52.5
517.50	580.0	517.50	570.0
580.00	no	570.00	yes
575.89		570.00	
553.60		553.60	
521.45		521.45	

**Explanation** In the first sample, the range is about 10.78% of the maximum wage, thus the wages are not fair. In the second sample, the range is about 9.21% of the maximum wage, thus the wages are fair.

**Scratch** After completing *Who is the tallest? (2)*, you should be comfortable reading six numbers and finding their maximum. To find the minimum, just use `min` instead of `max`.

See the first code listing in Chapter 1 if you need a refresher on outputting floats rounded to two decimal places.



## Chapter 2

# Looping through data

In the problems you've encountered so far, you would:

- read a fixed number of items from `IN`;
- perform some calculation on those items;
- write a result to `OUT`.

I'm sure you understand, though, that most problems don't come with a fixed number of items of input, and sooner or later you will need to write code that can handle input of any size. Well, that time is now.

## 201 Who is the tallest? (3) (worked example)

**Question** You run a photography studio for sporting teams. Before a team comes in for a photo shoot, you ask them to submit the height of each person in centimetres. This helps you to arrange studio props appropriately. In particular, you want to know the height of the tallest person in the photo so that the artwork on the wall can be placed in the best position.

**Input** A positive integer  $N$ , followed by  $N$  lines, each containing a person's height in centimetres.

### Sample

IN	OUT	IN	OUT
8	181	5	128
165		127	
177		128	
172		128	
180		128	
175		127	
179			
181			
180			

**Explanation** In the first sample, there are eight heights given and the greatest of these is 181. In the second sample, there are five heights given and the greatest of these is 128. The fact that the “greatest” height occurred three times does not matter.

**Scratch** In all problems so far, we have known exactly how much data to read. Here we read the first line and it *tells* us how much data to read! This means we need a program loop.

Loops are discussed in more detail in [Ref: Chapter 3](#), but we only need a very basic use of them here: to repeat an action  $N$  times.

The beginning of our solution looks like (excluding the `ex201` function header):

```
1     N = int(IN.readline())
2     for i in range(N):
3         height = int(readline())
4         # ... do something with height ...
```

It is hopefully clear that if the first line of input is 8, then the code above will read that value and then read eight more lines.

So what to do with each new height that we read? We are trying to find the maximum height, so it would be good to keep track of that as we go. We start off initialising it to zero:

```
1     maxheight = 0
```

Within the loop, each new height we read is *potentially* the maximum height. How do we know whether it is? Well, if it's greater than our current maximum, of course!

```
1     if height > maxheight:
2         # ...
```

And if it *is* greater than our current maximum, then we update our current maximum.

```
1     if height > maxheight:
2         maxheight = height
```

This means that every time the loop runs, the `maxheight` variable contains the *greatest height seen so far*. And therefore, when the loop has run its prescribed number of times, `maxheight` contains the greatest height overall.

It's pretty simple, really, but you need to internalise every little detail of this solution and adapt the techniques to the coming problems.

## Solution

Putting all the above together, we get:

```
1  def ex201(IN, OUT):
2      N = int(IN.readline())
3      maxheight = 0
4
5      for i in range(N):
6          height = int(IN.readline())
7          if height > maxheight:
8              maxheight = height
9
10     print(maxheight, file=OUT)
```

**Notes** For now, we treat the code `for i in range(N)` as a magic incantation telling Python to do something  $N$  times. The *loop variable*  $i$  is being completely ignored in this code, because we don't care how many times we've been through the loop so far. But we need to have a loop variable, so  $i$  will do.

It is tempting to shorten the name `maxheight` to `max`. But as you can see from the way `max` is coloured, we shouldn't do that: `max` is a special word in Python that we don't want to interfere with.

It is worth looking at a *trace table* of this algorithm. Sketching such a table *before* you write any code is a valuable skill in solving problems. We are tracing the data used in the first sample. The  $i$  column is included to show the looping but rendered in grey because we don't actually use this variable.

$i$	$height$	$maxheight$
-	-	0
0	165	165
1	177	177
2	172	
3	180	180
4	175	
5	179	
6	181	181
7	180	

The `maxheight` column shows exactly when that variable is updated, which occurs in Line 8.

## 202 Shopping (2)

**Question** You are in the middle of a sizeable shopping trip and realise you may not have enough money to pay for everything you have put in your trolley. Quickly, you write a program that takes in the quantities and prices of all the items you intend to purchase and reports the total price.

### Input

The first line of input is  $N$ , which is the number of quantity-item pairs. Following this are  $2N$  lines, each pair of which contains the quantity and the price. The example will make this clear.

- $0 < N < 10\,000$ .
- Each quantity  $Q_i$  is an integer that satisfies  $0 < Q_i < 1000$ .
- Each price  $P_i$  is a float that satisfies  $0.00 < P_i < 1000.00$ .

**Output** The output is a single float containing the total price, and it must be rounded to two decimal places.

### Sample

IN	OUT
5	97.37
4	
2.99	
1	
3.15	
2	
14.95	
19	
0.14	
7	
7.10	

**Explanation** The output represents \$97.37, which is  $4(\$2.99)+1(\$3.15)+2(\$14.95)+19(\$0.14)+7(\$7.10)$ .

**Scratch** After reading the integer  $N$ , we use it as we did in the last exercise to drive the loop:

```
1   for i in range(N):  
2       # ...
```

The difference here is that we read *two* values each time through the loop, so the code will look like this:

```
1   for i in range(N):  
2       qty = int(IN.readline())  
3       pri = float(IN.readline())  
4       # ...
```

Hopefully you can work out what to do to make this a complete solution.

Here is a trace table of the sample data.

<i>i</i>	<i>qty</i>	<i>pri</i>	<i>total</i>
			0.00
0	4	2.99	11.96
1	1	3.15	15.11
2	2	14.95	45.01
3	19	0.14	47.67
4	7	7.10	97.37

## 203 Scrabble tally

**Question** Four friends—Albert, Betty, Charlie and Dotty—have played a lot of Scrabble games over the years. The competition has been pretty close, but they've all kind of forgotten who has won how many games. Luckily, when moving house recently, Dotty unearthed a long list of the winners of all their games! By processing this list, you will be able to give them a full tally.

**Input** The first line of `IN` contains  $N$ , the number of winners listed. Following this are  $N$  lines, each of which contains the name `Albert`, `Betty`, `Charlie` or `Dotty`, representing the winner of one game.

**Output** `OUT` contains four lines, reporting a tally as seen in the sample.

### Sample

IN	OUT
10	Albert: 3
Dotty	Betty: 0
Albert	Charlie: 2
Albert	Dotty: 5
Charlie	
Dotty	
Albert	
Dotty	
Dotty	
Dotty	
Charlie	

**Scratch** In this problem, you are reading *strings* for the first time. “String” is the computer-science word for textual data, presumably because the characters are “strung” together. To read a string, we could simply call `IN.readline()`, with no `int(...)` or `float(...)` wrapper because we certainly do not want Python to attempt to convert our names to numbers.

There's a sting quite literally in the tail, though, because when we read a line of data from `IN`, the resulting string has a newline character at the end, and we don't really want that.

## 204 Area calculator

**Question** As you pore over your Maths homework, you tire of the seemingly endless and pointless area calculations you have to do. All the circles, semicircles, rectangles, squares, parallelograms and triangles demand so much attention, and you only have so much to give. But if you write a program to do the calculations for you, you'll get this wretched work done and have time for a camomile tea while watching *Rick & Morty* before bed.

### Input

The input contains any number of lines, which represent the shape names and the values required to calculate them. For circles and semicircles you get one value: the radius. For rectangles, parallelograms and triangles, you get base and (perpendicular) height. For squares, you get the length of one side. The last line of input is simply `stop`.

Your input will contain ten area calculations or fewer.

**Output** The output contains several lines, each of which is the result of an area calculation, which must be a float rounded to three decimal places.

### Sample

IN	OUT
circle	14.451
4.6	30.0
triangle	85.5
12	361.0
5	25.92
parallelogram	
19	
4.5	
square	
19	
rectangle	
7.2	
3.6	
stop	

**Scratch** This problem differs from the earlier ones in two ways:

1. We are *not* told at the beginning how many lines there will be.
2. Depending on the shape, we may need to read one line or two.

Therefore, you deserve some help with this problem.

To solve #1, we want to loop *endlessly* and break out of the loop when we read `stop`. This can be done with the following code structure:

```
1  while True:  
2      # ...  
3      if (condition):  
4          break  
5      # ...
```

The `while True` part creates the endless loop, and the `break` statement gets out of it. Naturally, the *condition* for us to break out is having read the word `stop`.

Solving #2 is not difficult and you probably don't need help with it, but just in case, there is a hint on the next page.

## Hint

Here is the structure of the code, excluding the function header.

```
1  while True:
2      shape = IN.readline().strip()
3      if shape == 'stop':
4          break
5      elif shape == 'circle':
6          radius = float(IN.readline())
7          area   = 3.141592654 * radius * radius
8          print(round(area,3), file=OUT)
9      elif shape == 'rectangle':
10         base   = float(IN.readline())
11         height = float(IN.readline())
12         # ...
13     # etc.
```

## 205 Drought

**Question** (Sourced from ORAC) Years of drought have hit rural Australia hard. With catchment levels at an all time low, you decide to purchase a rainwater tank. Soon the winter rains arrive, and the tank slowly begins to fill.

You begin to wonder just when your tank will be entirely full. A friend in the weather bureau has kindly lent you rainfall predictions for the next few days. Given these predictions, and the size of your rainwater tank, write a program to determine how many days your tank takes to fill.

**Input** The first two lines of the input file will contain the values  $N$  and  $C$ , where  $N$  is the number of days the weather predictions last, and  $C$  is the capacity of your rainwater tank in litres. You are guaranteed that  $1 \leq N \leq 1000$ , and that  $C$  is a positive integer no greater than the total amount of rain that falls over the  $N$  days.

The remaining  $N$  lines of input will describe the rainfall levels for each day in order. Each line will contain a single integer between 0 and 1 000 000: the amount of rain (in litres) that will fall over your rainwater tank that day.

**Output** Your output should consist of a single integer: the number of days until your rainwater tank fills.

### Sample

IN	OUT	IN	OUT
6	4	6	5
10		11	
2		2	
3		3	
3		3	
2		2	
2		2	
4		4	

**Explanation** In both examples, the total rainfall changes as follows:

Day	Running total (L)
1	2
2	5
3	8
4	10
5	12
6	16

Hence a 10-litre water tank is full after 4 days and an 11-litre water tank is full after 5 days.

**Scratch** If you initialise a variable to track the running total to zero at the beginning and keep it updated as you read in the data, you should be able to solve this.

Good to know: you can `break` out of a `for` loop just like you can a `while` loop.

## 206 Cute numbers

**Question** (Sourced from ORAC) For you, numbers have personalities. The number 4 is elegant, 18 is strong and 42 is enigmatic. And, of course, any number ending in 0 is cute. The more zeroes at the end of a number, the cuter that number is. Therefore 70, 36 640 and 1 800 090 are only a little bit cute (ending in just one zero), whereas 400 and 99 200 are very cute (ending in two zeroes) and 30 000 is really really cute (ending in four zeroes).

Your task is to read in a number  $N$  and determine how many zeros are at the end of that number, so you can tell just how cute the number is.

**Input** The first line of input will consist of the single integer  $d$ , telling you how many digits are in the number  $N$ . You are guaranteed that  $1 \leq d \leq 100\,000$ .

Following this will be  $d$  additional lines, each containing a single digit (0, 1, 2, 3, 4, 5, 6, 7, 8 or 9). These will be the digits of  $N$ , written from left to right. You are guaranteed that the first digit of  $N$  will not be zero.

**Output** You must write a single integer as output, representing the number of zeroes at the end of  $N$ .

### Sample

IN	OUT	IN	OUT
5	2	7	1
9		1	
9		9	
2		0	
0		0	
0		0	
		9	
		0	

**Explanation** The first example describes  $N = 99\,200$ , which ends in two zeros. The second example describes  $N = 1\,800\,090$ , which contains many zeroes within but has only one zero at the end.

## 207 High-wire walk

**Question** Philippe the high-wire artist is planning his next audacious high-wire walk in New York City. He wants to walk over as many buildings as possible, but at a prescribed height.

Fortunately, he has a list of consecutive building heights and he wants you to assist in finding the best option for his walk.

For example, say there are 12 buildings in the list and Philippe wants to walk at a height of 80 metres. Then the possible options are highlighted below.

100 **50** **60** 90 110 **20** **50** **40** **70** 130 110 70

Note that the 70 at the end is not highlighted because there is no tall building to the right of it that can be used as an anchor.

The best option for Philippe in this scenario is to walk over four buildings.

**Input** The first two lines contain the integers  $N$  and  $H$ , which represent the number of buildings in the list and the height at which Philippe wants to walk.

Following that are  $N$  positive integers representing the heights  $B_i$  of the buildings.

You are guaranteed that  $5 \leq N \leq 1000$ ,  $10 \leq H \leq 500$  and  $10 \leq B_i \leq 1000$ .

**Output** You will output one non-negative integer: the largest number of buildings over which Philippe can walk.

### Sample

IN	OUT	IN	OUT	IN	OUT
12	4	6	1	6	0
80		80		80	
100		100		20	
50		70		20	
60		90		100	
90		20		100	
110		20		20	
20		20		20	
50					
40					
70					
130					
110					
70					

**Explanation** The first sample data matches the scenario given in the example.

In the second sample, the greatest number of suitably small buildings are all the way to the right, so they can't be used. That leaves the small interval of 1 unit in the middle as the answer.

In the third sample, the buildings to the left and right of the two towers in the middle can't be used, and the answer is zero: there is nowhere in this scenario where Philippe can perform his high-wire act.

**Note** The second and third samples above are what might be called “interesting” data, or, to use a term of art, “pathological” data that is designed to fail. Such data can appear in many Informatics problems but it is not usually revealed in the samples: you have to force yourself to consider it in order to solve the problem fully.



# Chapter 3

## Looping to solve problems

In Chapter Ref: 2, or whatever, you learned to code *loops* for the purpose of reading an arbitrary number of lines of data. But the problems you solved were still quite direct in nature. For example, you might read many lines like `circle` followed by 40, but the act of calculating  $\pi(40)^2$  is still very straightforward.

In this chapter we will generally read a small fixed amount of data but will need to use loops to solve the problem itself. This is something that beginners tend to find very difficult. So let's break it down with two simple examples that will be developed fully in the first two problems:

- How many factors does a number (42, say) have? The simplest way to answer this question is to consider every possible number between 1 and 42 (inclusive) and count how many of them divide 42 evenly, which in computing terms means leaving a remainder of zero. Thankfully, Python makes it easy to operate on the number 1, then the number 2, then 3, and so on until we reach 42.
- How far does Sarah<sup>1</sup> run in (say) 100 days, given she runs 500 m on the first day and increases her run by 75 m each day thereafter until she reaches 1500 m per day? Clearly, her running pattern over time will be

500, 575, 650, . . . , 1325, 1400, 1475, 1500, 1500, 1500, . . .

and we need to add 100 terms of that sequence. So we are doing *something* 100 times (adding a number to a running total, which begins at zero), but the details (how much we add) potentially changes each time.

Before we get stuck in to those two problems, we see below **two ways** of looping to do something really simple: list the 12 times tables up to 5.

```
1  def times_tables_example_1():
2      n = 1
3      while n < 6:
4          print(f'12 x {n} = {12*n}')
5          n = n + 1
6      print('Done')
```

If you type this in and run your code, you can then run `times_tables_example_1()` in the shell and see the output:

```
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
```

---

<sup>1</sup>See Ref: Jogging (1) if you haven't already

```

12 x 4 = 48
12 x 5 = 60
Done

```

In the code above, *n* is a *loop counter* that takes on the values 1, 2, 3, 4, 5. It takes on these values because we explicitly set it to 1, then the loop runs so long as *n* < 6, and we explicitly increment it each time the loop runs. Thus Lines 4–5 (the *loop body*) run five times, controlled by the *loop condition* on Line 3. When the loop condition decides enough is enough (that is, the condition *n* < 6 becomes **False**), control passes to Line 6, and the code then finishes.

The description above makes two mentions of the variable *n* being manipulated *explicitly*. Python offers an *implicit* alternative, where we basically say “I want *n* to range from 1 to 5” and the low-level details are taken care of for us. This is demonstrated in the code below.

```

1  def times_tables_example_2():
2      for n in range(1,6):
3          print(f'12 x {n} = {12*n}')
4      print('Done')

```

The output from our two code listings is exactly the same. Beginners often prefer the first style because you can see everything that is happening. It doesn’t take long until they prefer the second style, though, because looping is such a common occurrence in programs that we tend to appreciate it being supported directly.

Note that `range(1,6)` produces the values 1, 2, 3, 4, 5 as we wish. It’s known as an inclusive-exclusive range: the first value (1) is *included*; the second value (6) is *excluded*. There are good reasons for this that won’t be detailed here.<sup>2</sup>.

Here are a few more code snippets that you can run if you like to get a bit more experience with how `for` loops work.

```

1  def for_loop_examples():
2      for x in range(-8,9):
3          print(f'{x} squared is {x*x}')
4
5      for name in ['Steven', 'Peta', 'Jessica']:
6          print(f'Hi {name}!!!!')
7
8      for m in range(5):
9          print(f"This is printed five times. I don't about m, but m = {m}")
10
11     import time
12     for t in range(4,20,3):
13         print(f'Sleeping for {t*100} milliseconds')
14         time.sleep(t * 100)

```

As shown above, `range` can take either one, two or three arguments. To be clear:

Call this...	...and you'll get...
<code>range(5)</code>	0, 1, 2, 3, 4
<code>range(3,9)</code>	3, 4, 5, 6, 7, 8
<code>range(4,14,2)</code>	4, 6, 8, 10, 12
<code>range(10,0,-1)</code>	10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Just for fun, we can finish this introduction with a blast off, implemented two ways.

---

<sup>2</sup>Consider that, in a sense, `range(10,25) + range(25,30) == range(10,30)`

```
1  def blastoff_1():
2      import time
3      for n in range(10,0,-1):
4          print(n)
5          time.sleep(1000)
6          print("Blast off!!!")
7
8  def blastoff_2():
9      import time
10     n = 10
11     while n > 0:
12         print(n)
13         time.sleep(1000)
14         n = n - 1
15         print("Blast off!!!")
```

## 301 How many factors? (worked example)

**Question** Given a positive number  $N$ , determine how many factors it has.

**Input** A single integer  $N > 0$ .

**Output** A single integer representing the number of factors of  $N$ .

**Sample**

IN	OUT	IN	OUT	IN	OUT
12	6	441	9	73	2

**Explanation** The number 12 has six factors: 1, 2, 3, 4, 6, 12. The number 441 has 9 factors: 1, 3, 7, 9, 21, 49, 63, , 147, 441. The prime number 73 has two factors: 1, 73.

**Scratch** To get the factors of 12, we will consider every number from 1 to 12.<sup>3</sup> Here is a logical setting out of what needs to be done. First of all, initialise *count* to zero. Then let *f* (called *f* because it's a potential factor) take on values 1 to 12 as the table shows.

<i>f</i>	factor of 12?	<i>count</i>
1	True	1
2	True	2
3	True	3
4	True	4
5	False	
6	True	5
7	False	
8	False	
9	False	
10	False	
11	False	
12	True	6

When the loop has completed, we have *count* = 6 and that is the answer we write to OUT.

Here is a pseudo-code<sup>4</sup> algorithm for what we need to do.

```
set Count to 0
foreach f in 1..12 {
    if f is a factor of 12 {
        increase Count
    }
}
output Count
```

To solve the problem, we need to do two things:

- generalise from 12 to  $N$ , the number that we get from IN;
- express *f* is a factor of  $N$  in Python.

The first is easy. The second requires familiarity with the *modulus* operator, which in Python (in fact, in most programming languages) is spelled %. It gives you the remainder from a division. Consider these examples:

<sup>3</sup>If we were smarter, we'd consider 1 to 6, then count 12 as an automatic factor. And there's an even smarter way than that! But we're going for simplicity here.

<sup>4</sup>Pseudo-code is simply code *ideas* expressed in general without using a specific programming language.

```
10 % 4    --> 2      (10 divided by 4 leaves remainder 2)
11 % 4    --> 3      (11 divided by 4 leaves remainder 3)
12 % 4    --> 0      (12 divided by 4 leaves no remainder -- it's divisible)
```

Thus  $f$  is a factor of  $N$  translates into Python as  $N \% f == 0$ .

### Solution

Here is the algorithm expressed in Python, in both looping styles.

```
1     def ex301(IN, OUT):
2         N = int(IN.readline())
3         count = 0
4         for f in range(1,N+1):
5             if N % f == 0:
6                 count = count + 1
7             print(count, file=OUT)
8
9     def ex301(IN, OUT):
10        N = int(IN.readline())
11        count = 0
12        f = 1
13        while f <= N:
14            if N % f == 0:
15                count = count + 1
16            f = f + 1
17        print(count, file=OUT)
```

## 302 Jogging (2)

**Question** Sarah is about to take up jogging. She will start jogging  $D$  metres per day and increase her daily distance by  $I$  metres each day. When she reaches her target of  $T$  metres per day, she will stop the daily increase and just keep a consistent jogging pattern.

Find the total distance she jogs in  $N$  days.

### Input

The input contains four lines.

- $D$ , the distance (m) at which she starts jogging;
- $I$ , the amount by which she increases her jog each day (m);
- $T$ , her target daily jog (m), after which she doesn't increase it any further.
- $N$ , the number of days for which we will calculate the total distance.

All values are integers, and you are guaranteed that  $D < D + I < T$ , that is, she will have to apply at least two increases before reaching her target.

**Output** You will write a single integer to **OUT**: the total distance she jogs in  $N$  days.

### Sample

IN	OUT	IN	OUT
100	1350	700	4410
50		20	
2000		750	
6		6	

**Explanation** In the first sample, over the first six days, Sarah runs a total of  $100 + 150 + 200 + 250 + 300 + 350 = 1350$  metres.

In the second sample, over the first six days, Sarah runs a total of  $700 + 720 + 740 + 750 + 750 + 750 = 4410$  metres.

**Scratch** It is intended here that you sum the distances in a loop. It is possible to work these out using mathematical formulas, but that is: (a) complicated; and (b) counter to the spirit of this chapter.

## **Chapter 4**

# **Operating on a list of numbers**

In these problems, ...

## 401 Shopping (3)

**Note** This is the same as *Shopping (2)* but with the data presented differently: all the quantities are grouped and all the prices are grouped. This makes it more difficult because you can't just calculate the total as you go; you need to store the values in lists first.

**Question** You are in the middle of a sizeable shopping trip and realise you may not have enough money to pay for everything you have put in your trolley. Quickly, you write a program that takes in the quantities and prices of all the items you intend to purchase and reports the total price.

### Input

The first line of input is  $N$ , which is the number of quantity-item pairs. Following this are  $2N$  lines:  $N$  lines of quantities and  $N$  lines of their respective prices.

- $0 < N < 10\,000$ .
- Each quantity  $Q_i$  is an integer that satisfies  $0 < Q_i < 1000$ .
- Each price  $P_i$  is a float that satisfies  $0.00 < P_i < 1000.00$ .

**Output** The output is a single float containing the total price, and it must be rounded to two decimal places.

### Sample

IN	OUT
5	45.01
4	
1	
3	
19	
7	
2.99	
3.15	
14.95	
0.14	
7.10	

**Explanation** The output represents \$97.37, which is  $4(\$2.99)+1(\$3.15)+2(\$14.95)+19(\$0.14)+7(\$7.10)$ .

## 402 A fair wage (2)

**Question** In the years since you completed *A fair wage (1)*, your company has grown and grown, so the quaint idea of reading in just six numbers and assigning each to its own variable to each has pretty much faded from memory. You now have  $N$  workers and want to do the same test: check that the *range* of wages is no greater than 10% of the maximum wage. Except in recent times you've found the 10% measure to be a little too inflexible. So now the target percentage  $P$  will be included in your data.

**Input** From **IN** you will read:

- the integer  $N$  ( $5 < N < 1000$ ), the number of employees at your company;
- the float  $P$  ( $1.0 < P < 30.0$ ), the percentage used to determine whether the wages are fair;
- $N$  lines, each containing a float representing an employee's weekly wage.

**Output** To **OUT** you will write three values:

- The range of wages, rounded to two decimal places
- The highest wage, rounded to two decimal places
- **yes** or **no** according to whether your wages are fair

### Sample

IN	OUT
7	35.42
8.5	635.17
635.17	yes
622.25	
631.02	
631.02	
628.56	
599.75	
608.10	

**Explanation** In the sample, the range is about 5.58% of the maximum wage, which is less than the target percentage 8.5%, so the wages are fair.

**Scratch** You know how to find the `max` or `min` of a group of individual numbers, but in this problem you need the maximum and minimum of a *list* of numbers. That's no problem. The Python functions are very flexible:

```
1   max(3,7,5,1,2)           # -> 7
2   max([3,7,1,5,2])         # -> 7
```

There are five arguments in line 1 and only one argument (a list) in line 2, but the result is the same. `max` can take any number of arguments and knows what to do if it is passed a single list.

## 403 Addition carry

**Note:** This exercise assumes some experience with lists.

**Question** When two numbers are added, carry can be involved. Given two numbers, perform the addition using the normal primary-school algorithm and output the number of times a carry was performed.

Both numbers provided in IN will be 1–20 digits long. They will not necessarily be the same length.

### Sample

IN OUT

19526

3

32287

IN OUT

232

0

51

**Explanation** The first example involves the following additions:

- $6 + 7$  (generating a carry)
- $2 + 8 + 1$  (generating a carry)
- $5 + 2 + 1$
- $9 + 2$  (generating a carry)
- $1 + 3 + 1$

There are three carries.

The second example clearly involves no carries.

**Scratch** To solve this problem we need to look at each number as a list of (numeric) digits instead of a number *per se*. So we read in the two numbers as strings and use some special code that extracts the digits.

```
1 S1 = IN.readline()      # "19526"
2 S2 = IN.readline()      # "2287"
3 A = [int(x) for x in S1] # [1, 9, 5, 2, 6]
4 B = [int(x) for x in S2] # [2, 2, 8, 7]
```

The code `for x in S1` iterates through each character '`'1'`', '`'9'`', '`'5'`', '`'2'`', '`'6'`' in `S1`, and of course `int(x)` converts each of those to an integer. Enclosing that code in `[...]` collects the generated values in a list. This kind of code, called a *list comprehension*, is a kind of superpower that Python has, although it may be a bit daunting at first.

For the algorithm we are presently implementing, we would really like our two digit lists to be in reverse order. The following code will do it.

```
1 S1.reverse()           # S1 is now [6, 2, 5, 9, 1]
2 S2.reverse()           # S2 is now [7, 8, 2, 2]
```

And now the digits line up and you can move *forward* through the two reversed lists.

**Solution** The rest of the code for this problem is an exercise for you.

# Chapter 5

## Miscellaneous problems

There are no new skills in this chapter, just a range of problems for you to try.

There is one new thing to learn about reading input, however, and that is **reading multiple inputs on the same line**. Problem 401 below demonstrates this.

### 501 Name, age and hourly rate (**worked example**)

There is no “question” here, really. You will simply read data about multiple people from three lines of **IN**, and write that data to **OUT** in a more readable format. In so doing, you will learn how to read multiple values from one line, and how to use Python’s format strings.

#### Sample

IN	OUT
James Lynne Toni Gerard	James is 23 years old and earns \$18.50 per hour
23 21 20 21	Lynne is 21 years old and earns \$21.00 per hour
18.5 21.25 22.30 21	Toni is 20 years old and earns \$22.30 per hour
	Gerard is 21 years old and earns \$21.00 per hour

**Scratch** To read multiple items from one line, we use Python’s `split()` function.<sup>1</sup> If you try `"one two three four".split()` you’ll get the idea of what `split()` does.

So the four names in the sample can be read with:

```
1     names = IN.readline().split()  
2             # -> ['James', 'Lynne', 'Toni', 'Gerard']
```

The ages and hourly rates, however, need to be converted to integers and floats:

```
1     ages  = [int(x) for x in IN.readline().split()]  
2             # -> [23, 21, 20, 21]  
3  
4     rates = [float(x) for x in IN.readline().split()]  
5             # -> [18.5, 21.25, 22.3, 21.0]
```

These are using Python’s superpower, list comprehensions, which enable us to transform one list—say, `['23', '21', '20', '21']`—into another, like `[23, 21, 20, 21]`.

You have two choices when it comes to understanding the above code:

---

<sup>1</sup>Technically, `split()` is a *method* of the `String` class. That’s more detail than we want at the moment. We’ll think of it as a *function* that acts directly on string objects.

- Memorise how to read multiple strings or ints or floats from one line, and leave it at that; or
- Understand fully how the code works so that you can use list comprehensions to power up your own code.

The choice is yours, but you *must* do at least #1 above.

## Solution

```

1  def train401(IN, OUT):
2      names = IN.readline().split()
3      ages = [int(x) for x in IN.readline().split()]
4      rates = [float(x) for x in IN.readline().split()]
5
6      for i in range(len(names)):
7          rate = round(rates[i],2)
8          text = f'{names[i]} is {ages[i]} years old and earns ${rate} per hour'
9          print(text, file=OUT)

```

**Afterword** Using another list comprehension, we can do all the float rounding in one go. Here is a more sophisticated solution. Note Lines 5 and 8.

```

1  def train401(IN, OUT):
2      names = IN.readline().split()
3      ages = [int(x) for x in IN.readline().split()]
4      rates = [float(x) for x in IN.readline().split()]
5      rates = [round(x,2) for x in rates]
6
7      for i in range(len(names)):
8          text = f'{names[i]} is {ages[i]} years old and earns ${rates[i]} per hour'
9          print(text, file=OUT)

```

Also, we can push formatted strings further to create a nice tabular output. This will not pass the test for problem 501, so it's named differently, but you can still run it with `l.run(train401a)` and provide your own data.

```

1  def train401a(IN, OUT):
2      names = IN.readline().split()
3      ages = [int(x) for x in IN.readline().split()]
4      rates = [float(x) for x in IN.readline().split()]
5      rates = [round(x,2) for x in rates]
6
7      print(f'{Name:12} {Age:>6} {Rate:>7}', file=OUT)
8      for i in range(len(names)):
9          text = f'{names[i]:12} {ages[i]:>6} {rates[i]:>7}'
10         print(text, file=OUT)

```

## 502 Golf (1)

**Question** Two people play a short game (nine holes) of golf, and you must write a program to determine the winner.

Normally, the winner is the person with the lowest total score, where “score” means “number of shots required to complete all holes”. But sometimes, players prefer to count *holes* instead of *shots* to determine the winner. And this is one of those times.

Say Jennifer won five holes, Kate won three holes, and one hole was tied. Then the result would be **Jennifer won by 2 holes**. If Jennifer and Kate both won four holes and one was tied, then we have to decide the winner by overall points, and the result might be **Kate won by 3 points**. If both holes and points are tied, then the result is **Jennifer and Kate tied**.

**Input** From IN you will read data from *three lines*:

1. The names of the two players
2. The scores for the first player (nine integers)
3. The scores for the second player (nine integers)

**Output** To OUT you will write the result, as suggested by the paragraph above.

### Sample

IN	OUT
Jennifer Kate	Kate won by 1 hole
5 4 5 3 3 4 6 4 5	
4 4 5 4 4 4 5 4 4	

IN	OUT
Jennifer Kate	Jennifer won by 2
5 4 5 3 3 4 6 4 4	points
4 4 5 4 4 4 5 4 4	

IN	OUT
Jennifer Kate	There was no winner
5 4 5 3 3 4 6 4 4	
4 4 5 4 4 4 5 4 4	

**Note:** **TODO: massage the data!**

### Explanation

- In the first sample, Jennifer won two holes and Kate won three holes, so a winner can be determined by holes alone.
- In the second sample, Jennifer and Kate won three holes each, so points total points need to be considered. The points for Jennifer and Kate were 38 and 40 respectively **Note: TODO: check!**, and a lower score is better.
- In the third sample, Jennifer and Kate are equal on both holes and points.

**Scratch** Ensure you've read the notes at the start of this chapter about reading multiple values from one line.



## **Chapter 6**

# **Two-dimensional data**

Intro text...

## 601 Cash grab (1) (worked example)

Behind the gates of the walled city *Adventis* lies an enormous  $8 \times 10$  board with 80 tiles, each one of them filled with coins. You are invited to choose which tile to stand on and will be given one minute to scoop up as much cash as you can. Taking a look from on high, you reason that wherever you stand, one minute will only afford you enough time to get the cash on that tile and on all the immediately surrounding tiles. That is, nine tiles in total.

On a scrap of paper, you jot down your estimate of how much cash is on each tile, and you have only moments to decide where you will be placed.

**Input** Eight rows of ten non-negative integers, each representing the cash amount on one tile. The rows are numbered 1–8 and the columns 1–10, but these do not appear in the input.

For simplicity's sake, as this is your first exposure to two-dimensional data, all border tiles will have a value of 0.

**Output** Two numbers `row` `col` identifying the most profitable place to stand.

### Sample

IN	OUT
0 0 0 0 0 0 0 0 0 0	5 4
0 1 2 1 3 2 4 3 1 0	
0 4 1 2 1 3 0 2 0 0	
0 0 5 0 3 2 1 3 1 0	
0 2 7 8 5 4 2 0 3 0	
0 1 6 3 4 2 0 0 5 0	
0 3 0 2 2 1 2 2 3 0	
0 0 0 0 0 0 0 0 0 0	

**Explanation** The location (5, 4) (in row-column notation) is the square with value 8, and the nine cells within reach are 5 0 3 7 8 5 6 3 4, which clearly produces a higher total than any other starting cell.<sup>1</sup>

**Scratch** We need to simulate standing in many different places and counting the surrounding cash, and keep a running “best location”. How many places do we need to try? Well, the zeros on the outside mean there’s no point standing on them. There’s no point standing *next to* a row or column of zeros either, so that means we try all locations in the rectangle (2, 2)–(8, 8).

But first, we need to load the data in. We will create a list called `DATA` which is really a list of *rows*, each of which is a list of values. That is, when we’re done, we’ll have (referring to the sample data):

```
1     DATA == [[0,0,0,0,0,0,0,0,0,0],      # Row 1 (index 0)
2             [0,1,2,1,3,2,4,3,1,0],      # Row 2 (index 1)
3             [0,4,1,2,1,3,0,2,0,0],      # Row 3 (index 2)
4             # ...
5             [0,0,0,0,0,0,0,0,0,0]]     # Row 10 (index 9)
```

Thus we have two-dimensional data as a list of lists. If we want to access the 4 at row 2, column 7, then we use `DATA[1][6]`, noting zero-based indexing. Just typing `DATA[2]` gives us the entirety of row 3.

Here is a way to get the values from IN into our variable `DATA`.

---

<sup>1</sup>We can see the answer by eye in this case, but with different data it would not be so easy.

```

1   DATA = []                      # Start with an empty list and append each row.
2   for i in range(8):
3       row = [int(x) for x in IN.readline().split()]
4       DATA.append(row)

```

Now to actually access all the data we want, from (2, 2) down to (6, 8), we can use a loop within a loop. This is the normal way to deal with two-dimensional data.

```

1   for row in range(2,7):
2       for col in range(2,9):
3           print(f'Row {row+1}, Column {col+1}: {DATA[row][col]}')

```

It is worth trying this code to ensure you know what it does and why. Note that we are translating between the 0-based indexing of lists and the 1-based language of the problem.

The double-loop above simulates standing in the 49 different locations from (2, 2) to (6, 8). So what do we want to do in each location? We want to count all the cash in that location and its eight neighbours.

```

1   for row in range(2,7):
2       for col in range(2,9):
3           cash = DATA[row-1][col-1] + DATA[row-1][col] + DATA[row-1][col+1] + \
4                  DATA[row][col-1] + DATA[row][col] + DATA[row][col+1] + \
5                  DATA[row+1][col-1] + DATA[row+1][col] + DATA[row+1][col+1]

```

With code like that, we might prefer to use `r` and `c` instead of `row` and `col`. That will appear in the solution.

More than just count the cash, though, we need to see whether this is the best cash amount we've encountered so far, and if so, remember where it is. For that we will use two variables: `bestcash` and `location`. We will store the location as a (row,column) pair.<sup>2</sup>

## Solution

```

1   def train501(IN, OUT):
2       DATA = []
3       for i in range(8):
4           row = [int(x) for x in IN.readline().split()]
5           DATA.append(row)
6
7       bestcash = 0
8       location = (0,0)
9
10      for r in range(2,7):
11          for c in range(2,9):
12              cash = DATA[r-1][c-1] + DATA[r-1][c] + DATA[r-1][c+1] + \
13                  DATA[r][c-1] + DATA[r][c] + DATA[r][c+1] + \
14                  DATA[r+1][c-1] + DATA[r+1][c] + DATA[r+1][c+1]
15              if cash > bestcash:
16                  # We have a new potential answer.
17                  bestcash = cash
18                  location = (r+1, c+1)
19
20      row, column = location
21      print(row, column, file=OUT)

```

---

<sup>2</sup>Python conveniently allows us to store pairs or triples or ... of values in round brackets. These are called *tuples*, and once you've created them you can't change them. You'll learn why not when you encounter the *dictionary* data type.

**Explanation** Much of the code was explained in *Scratch* above. In Lines 7–8 we initialise the two variables that help us find the best location, and in Lines 15–18 we use them. In Line 18 we add 1 to the row and column because that is the numbering system used in the problem. In Line 20 we *unpack the tuple* into its two individual values so that we can print them out with a space between. If we had called `print(location)` then the output would be `(5,4)`.

## 602 Cash grab (2)

The question is the same as *Cash grab (1)* but the board size is now variable. The first line of IN will contain the values  $R$  and  $C$ , the number of rows and columns, respectively. Following this will be  $R$  lines, each containing  $C$  non-negative integers.

### Sample

IN	OUT
...	...

### Solution

```
1  def train501(IN, OUT):  
2      pass
```

## 603 Cash grab (3)

Variable size.

### Sample

IN	OUT
----	-----

...	...
-----	-----

### Solution

```
1     def train501(IN, OUT):  
2         pass
```

## 604 Golf (2)

This question varies the earlier *Golf (1)* with an arbitrary number of players and holes.

**Question** A group of  $N$  people play a game of golf (with  $Q$  holes), and you must write a program to determine the winner. As before, whoever wins the most holes is the winner. If there is a tie for most holes won, then it's whoever took the fewest overall shots.

**Input** The first line of IN contains the integers  $N$  and  $Q$ , the number of players and number of holes, respectively. The next line contains  $N$  space-separated strings, which are the first names of all the players. Then there are  $N$  lines containing  $Q$  integers each, which are the players' scores on the  $Q$  holes.

### Output

- If one player won more holes than any other player, you will output a line like `John won by 4 holes`.
- If holes are tied but one player scored a lower total than any other, you will output a line like `Samantha won by 3 points`.
- Otherwise, you will output `There was no winner`.

### Sample

IN	OUT
4 7	Kate won by 1 hole
Sam Steve Terri Kate	
5 3 5 6 7 5 5	
4 3 5 4 5 5 4	
4 4 4 4 4 5 3	
3 3 4 5 4 3 4	

IN	OUT
...	...
IN	OUT
...	...

### Explanation

- In the first sample, the winner of the seven holes were (using player numbers 1–4): 4, -, -, -, -, 4, 3. Thus player 4 (Kate) won by one hole.
- In the second sample, ...
- In the third sample, ...

**End of document**