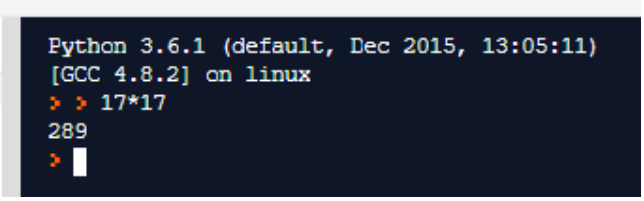
Quadratic Solver: A very first Python project

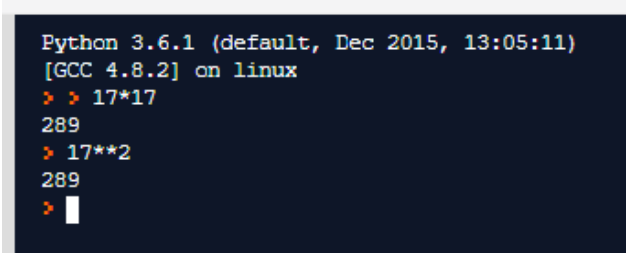
The aim: *Give solutions, where possible, to a user-defined quadratic equation.*

Step 1: Use the interactive shell as a calculator.

The shell is the bit that interprets code, sitting on the right of the screen. It effectively allows you to write and execute your code all at once. It's a good way to test simple snippets of code to work out what type of result you'd expect, and it's also a quick easy way to perform calculations:

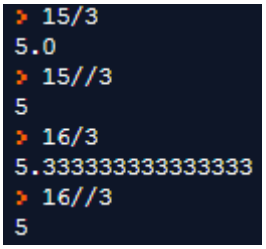


You can probably guess what * means in Python from the above example. + and – work as expected, and / means ÷, but powers need the symbol ** (unlike in Excel, or Google, or WolframAlpha, where ^ is the standard):



$\mathbb{R}$ or $\mathbb{Z}$?

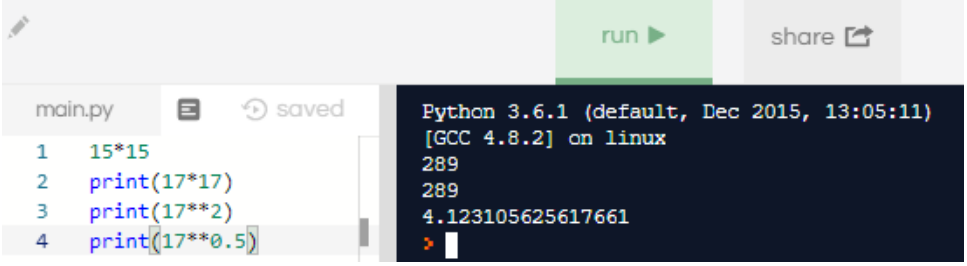
Python stores data differently depending on the data type. We've already worked with the 'int' type (that's integers, including 0 and negatives).
The other common numerical data type is 'float', or floating point numbers, which essentially translates to any real number.
This is why Python has two different types of division: integer division uses a double slash //, while normal division uses a single slash /.



Can you see how integer division works from the examples given opposite?
Can you work out why 15//3 gives 5 while 15/3 gives 5.0 in the example?

Step 2: Writing a program

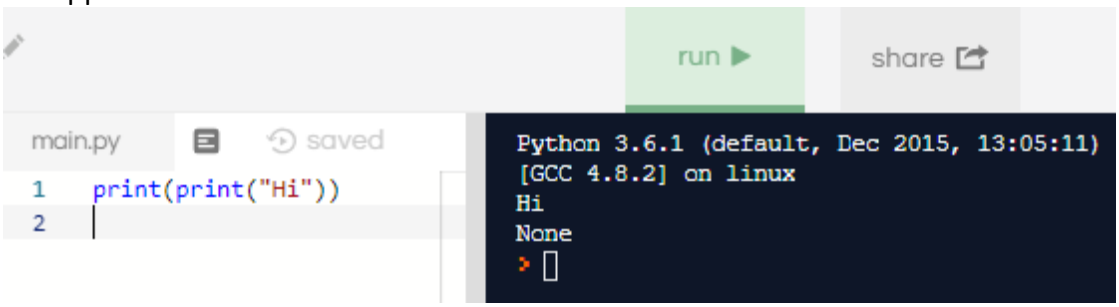
The program you write is a list of instructions that will be carried out sequentially by the Python interpreter. Here you can construct a more complex set of instructions, and when you're ready, click 'Run' to have the whole thing run at once. *But* notice that if you try the same commands we just did, although Python will dutifully calculate the answers for you, it won't automatically display the results on the screen for you. If you want stuff to appear in the shell on the right, enclose it in a print function. This makes sense: most of the calculations we will do don't need to be shown to the user, so we actively choose to output to the screen using the print function when necessary (and later, we might want to save our data to a text file instead, so we'll use a different output method altogether).



Functions

Functions (and their cousins 'Methods') are a big deal in Python, but they're a fairly natural idea to mathematicians: each function will have specific inputs or types of input you can provide it with, and it will carry out certain pre-defined processes, finally giving you a particular result as the output. The input values are called 'arguments', and are usually enclosed in brackets just like mathematical function notation. The mathematical function $\sin(\ )$ takes an angle as its input (or argument), and outputs a value between 1 and -1 as its output. The Python function $\mathrm{print}(\ )$ takes a value (typically text, but can be a single numerical value), and carries out the behind-the-scenes magic that displays that value on the screen.

Result or output?
Look at the snippet below.

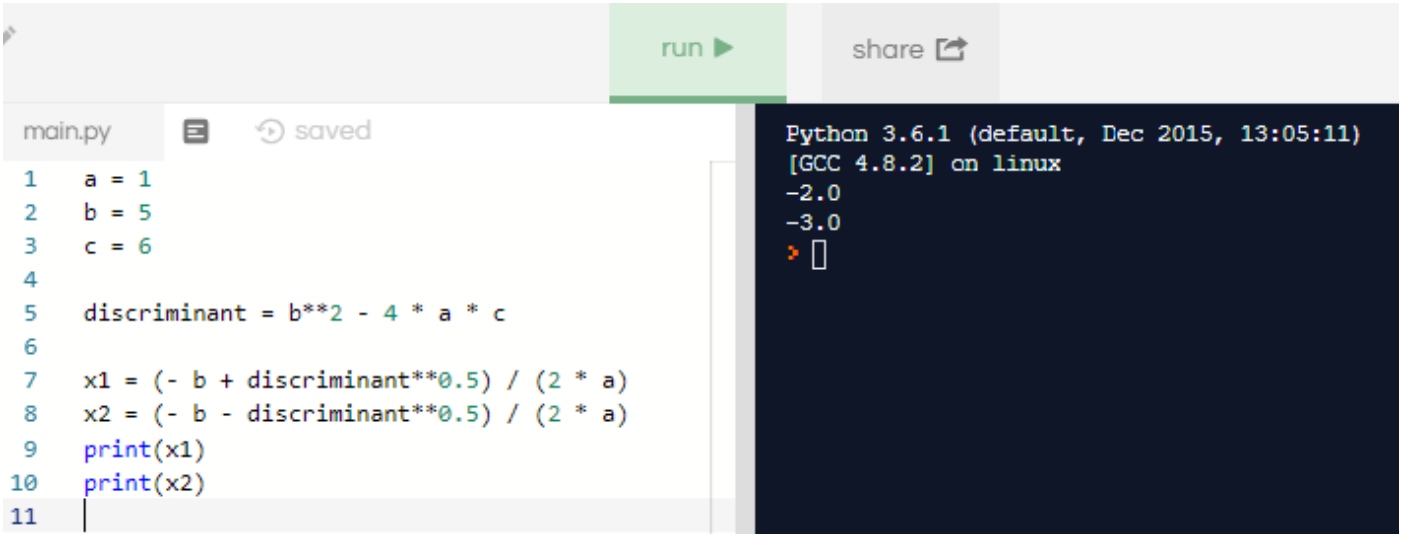


Bewildering, isn't it? The difference between what a function *does* and what a function *returns* as its output is a subtle one, but helps enormously later on. By default, *every* function in Python has to have a 'return value', or output. This is often obvious: if you use the function $\mathrm{factorial}(n)$ you would quite rightly expect the output value to be the factorial of n. However, some functions exist only to *do* something rather than to *produce* something. The factorial example is like a factory which is given raw materials and whose sole purpose is to generate a resulting finished product. Feed it an input of 5, and it'll dutifully work on it until finally returning 120.
But the print function's job is publicity. Give it an input value of 5, and it will tell the interactive shell to display it, which is exactly what you want, but if you then ask what finished product it has for you, the only sensible answer is 'None'. Which is actually a Python data type in its own right.

The code above starts up the print function (which is equivalent to running a mini-program which must complete before it can move on to the next line of your code). But then it has to evaluate the input provided, which is itself a print function. So that function is 'called' (the computer carries out the code), and dutifully returns a value of 'None'. *However*, in the process of telling the *first* print function what to print, the *second* print function did its job: it displayed *its* input (the text string "Hi") to the interactive shell. Since this happens before the function completes its task and returns its output value ('None'), the message "Hi" appears first, followed by 'None'.

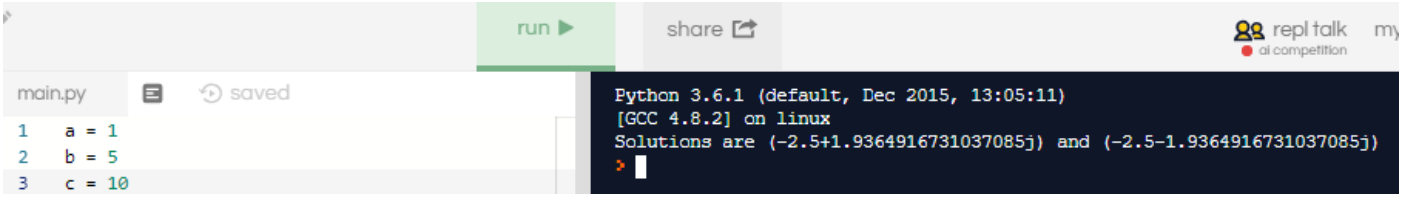
Step 3: A program that calculates
It's about time we made use of our ability to calculate things. We start out by defining our variables. This is just like algebra: letters can be assigned numerical values, so we can set values for a, b and c right at the start of our program. We've got quite a bit of flexibility here, though, and unless the context is obvious (like the quadratic formula), it's best to be more descriptive than a single letter. To make the code easier to read and debug (error-check), I've broken down the process into steps. First, I calculate the discriminant, then I work out each of the two possible roots, and finally print them. Once you've written your code, press the green 'run' button at the top to see the output. Experiment with different values for a, b and c and see if you can break your code by finding an edge case.

Have a go at doing this for yourself, then turn over to see an example of how it might be done in Python…

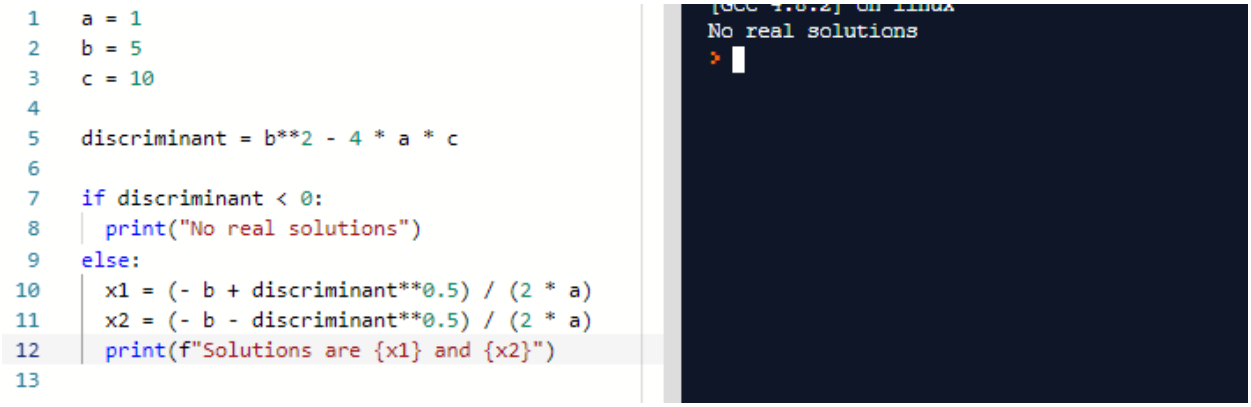


There are a few improvements we can make straight away that will add to the readability of your output. Python 3 comes with something called f-strings. Essentially they are a way to print strings and values together, because the print function can't take multiple different data types. Read the example below and you'll soon figure out how it works:

The second thing we should be wondering about is what to do if there are no real solutions. Firstly, look what happens when we give it a quadratic like that:



It turns out Python 3 can handle complex numbers, too (although it apparently uses the engineers convention of j rather than i). But what if you only want real solutions? We'll need to incorporate a test. This will check if a particular condition is satisfied, then carry out a different set of instructions depending on the result:



This brings up the next important aspect of coding in Python (other languages do this differently): the spaces at the front of a line (use 'Tab' if you want to add these in to maintain consistency) *do* make a difference to the code. The code above knows to print "No real solutions" when the discriminant is negative, and to do the final three lines only if it isn't negative. Look what a small change can do:

```
1   a = 1
2   b = 5
3   c = 10
4
5   discriminant = b**2 - 4 * a * c
6
7   if discriminant < 0:
8     print("No real solutions")
9   else:
10    x1 = (- b + discriminant**0.5) / (2 * a)
11    x2 = (- b - discriminant**0.5) / (2 * a)
12  print(f"Solutions are {x1} and {x2}")
13
```

```
[GCC 4.8.2] on linux
No real solutions
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(f"Solutions are {x1} and {x2}")
NameError: name 'x1' is not defined
>
```

We get an error! Let's see what it teaches us. It tells us exactly where Python realized it couldn't complete the program (line 12), so we know it's that final print statement that's causing an issue. You'll notice it quite happily carried out the first bit of the program before hitting a wall, too – sometimes the error will only occur some time after starting the program, when a particular condition fails to be met, for instance. The specific error here is a NameError, and Python tells us it doesn't know what to do with x1. If we trace through the program we'll find that x1 only gets a mention when it is first defined within the 'else' clause. In the previous example, the print statement at the end only occurred after x1 and x2 had been calculated, but in this example it'll try to carry out that instruction even if the 'else' clause was skipped. This bug is a sneaky one, too, because if we try it with different initial values that did have solutions, we'd never have noticed it.

Here's another refinement: get the values of a, b and c directly from the user. This is a bit slicker, because then nobody needs to see your code to make use of the program. Try this:

main.py saved

```
1   print("Quadratic Equation solver:")
2   a = float(input("Enter the value of a:"))
3   b = float(input("Enter the value of b:"))
4   c = float(input("Enter the value of c:"))
5
6   discriminant = b**2 - 4 * a * c
7
8   if discriminant < 0:
9     print("No real solutions")
10  else:
11    x1 = (- b + discriminant**0.5) / (2 * a)
12    x2 = (- b - discriminant**0.5) / (2 * a)
13    print(f"Solutions are {x1} and {x2}")
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Quadratic Equation solver:
Enter the value of a:1
Enter the value of b:5
Enter the value of c:6.2
Solutions are -2.2763932022500213 and -2.723606797
>
```

The input function prints a text string to the shell just like print, but then waits for the user to input a value and press enter before moving on. The input function then returns the value entered so we can use it. In this case, it is also necessary to convert the text provided by the user (which is hopefully a number, but as far as Python is concerned is simply a piece of typed text right now) into a number which can be used in calculations. We do this conversion using the float function, which takes text and, where possible, converts it to the floating point type (ie any real number).

In conclusion

Hopefully this brief introduction has given you a taste of what you can use Python for, and also given you an idea of how to learn some Python independently: Start with little snippets of code, and try to figure out what they do by changing things and breaking things. The best way to learn anything is to start trying, and the more mistakes you can make, and the quicker you can make them, the more rapidly you will learn!