KICKING OFF YOUR LEARNING

Day 1: Numbers, Arithmetic, and Printing to the Console

←

→

Welcome to the first day of our [30 Days of Python]() series! Let's talk about different types of number in Python, printing values, and performing simple arithmetic. We're also going to briefly discuss the concept of expressions.

If you haven't checked out the preliminary ["Day 0"]() post, I'd recommend you take a look at that before reading any further, as it contains information on how to get set up so you can follow along with the code examples.

A little bit about Python

The first version of Python was released 30 years ago in 1990. Since then, the language has undergone a huge amount of change, and has grown into one of the most popular programming languages in the world.

One of the main strengths of Python is the readability of its code, which makes it extremely accessible. Often Python code reads just like English. Python is also extremely flexible, and can be found in a wide range of industries, from machine learning to web development.

At the time of writing there are two major Python standards in active use.

First we have Python 2.7 which is the most current version of Python 2. Python 2 has technically reached its end of life, and will no longer get any additional updates, but there's still a huge amount of Python 2 code out there!

The second branch of Python is Python 3, with the most current version being Python 3.9. Python 3 was intended to be a replacement for Python 2, and boasts a number of improvements along with some significant syntax changes. Code written in Python 2 may not be compatible with Python 3, and even if the code runs we may end up with unexpected behaviour.

Throughout this series, we'll be working with Python 3.

Basic numbers

In Python, we have several different ways of representing numbers, the most common of which are integers and floating point numbers—also known as floats.

Integers and floats are basic numerical types in Python. Integers are used to represent whole numbers, and floats are used for real numbers with some decimal component. For example, we'd use an integer for the number 3, but a float for 3.141.

Creating an integer in Python is very simple: we just have to write an integral number (also known as a *whole number*). For example, we can create an integer with the value `4` like so:

```
4
```

We can also write negative integers by putting a `-` before the number:

```
-6
```

Floats are created in much the same way, but floats have some *decimal* component. In other words, there's something after the decimal point, even if that thing is `0`.

All of these are valid floats:

```
4.0
-156457.3
3.141
```

However, I'm sure that some of you have tried to run this code, and nothing seems to happen. At the very least, we don't get any output.

We need a way to see the numbers we're creating.

The `print` function

Luckily, Python has a built in *function* that outputs things to the console window for us. This function is called `print`.

We're going to cover functions in a lot of detail throughout this series, but for now you can think of functions as bundles of code that we can run to perform some specific action.

Somebody else has kindly written the (quite complicated) code that allows us to print things to the console, and they've bundled it up in a function called `print` . We can run this function to perform the action of printing.

The process of running a function like this is often referred to as “calling” a function.

Instead of "calling", some say "running" or "executing" a function. They all mean the same thing!

Let's call the `print` function to print the value `4` to the console:

```
print(4)
```

We can use this function multiple times if we want to, and values get printed out in the order of our `print` calls:

```
print(4)
print(3.141)
print(12345)
```

First we get `4` , then on a separate line we get `3.141` , and then finally we get `12345` on yet another line. As you can see, we don’t need to do anything special to print floats.

Style note

When we call a function, the parentheses (round brackets) are written directly next to the function name. We should be writing this:

```
print(4)
```

Not this:

```
print (4)
```

This isn’t going to affect how the code works—Python doesn’t care if we put spaces here—it’s just considered poor style, and developing a good coding style is very important. Good style is going to help make your code more readable, which is one of the most important metrics to consider when it comes to writing code.

A quick look at expressions

Before we can move onto the next section we need to introduce a new word: *expression*.

If something in our code evaluates to a value, it’s an expression. Bear with me, this is going to make a lot more sense in a moment.

As it happens, every piece of code we’ve written so far is an expression, and there are many different types of expression in Python.

For example, all the integers we've written are expressions. When Python sees each integer, it can use the value they represent.

We're also about to see some simple arithmetic operations like `5 + 7`, and those operations are expressions too. In the case of `5 + 7`, when Python sees that, it evaluates the expression to another integer: `12`.

When we write `print()`, that is also a type of expression called a *function call expression*.

Does that mean that `print()` has a value?

Yes, it does, and we’ll talk a lot more about this when we get to functions later in the series.

Let's look at arithmetic operators, and we’ll start to see why this concept of an expression is important.

Arithmetic operators

'

One of the things that we do all the time in Python (and programming in general) is basic arithmetic. We accomplish this primarily through the use of operator symbols like `+`, `-`, `/`, etc.

For example, if we want to add two numbers together, we can place the `+` operator between those two numbers:

```
1 + 2
3.4 + 11
8 + 4.0
```

When we use an operator like this, we're writing an expression. So when we write `1 + 2`, Python evaluates that and we get an integer with the value `3`.

This is important to keep in mind, because it affects our output when we use something like `print`. If we try to print something like this:

```
print(1 + 2)
```

Python is not going to print out the operation, because by time `print` runs, `1 + 2` has already been evaluated to a single value: the integer, `3`.

Let's look at a few more examples:

```
print(1 + 2)
print(3.4 + 11)
print(8 + 4.0)
```

If you ran the code above, you might have noticed something interesting. Instead of getting `3`, `14.4`, and `12` printed to the console, the final result was `12.0`.

This is because if either of the operands for `+` is a float, the expression evaluates to a float. The same is true of several of the

other operators, including `-` and `*` (which is used for multiplication).

These two operators work in exactly the same way as `+`, so we don't need to talk about them in too much detail, but here are a few examples:

```
print(7 - 5)
print(5 - 11.0)
print(-4 - 9)
print(4 * 7)
print(2 * 29.0)
print(8.2 * 34)
```

Division is performed using the `/` operator, but when performing division the result is *always* a float. It doesn't matter if both of the numbers are integers, or if the result would usually be a whole number. You can see this by running the code below:

```
print(5 / 6.5)
print(20 / 2)
print(5.5 / 0.5)
```

Using multiple operators

Sometimes we need to perform more complicated calculations, and this may involve the use of multiple operators. This is totally fine in Python, and we can chain as many of these arithmetic operators together as we like. Just make sure that things aren't becoming hard to read. This is no good:

```
5 + 4 + 8 + 565 + 454.0 + 9 + 2 + 11 + 3 + 20 + 45 + 6
7
```

We can also chain together different operators: they don't all have to be `+` or `-`, etc.

```
5 * 3 - 6 + 2 / 4
```

When we do this, the order the expressions are evaluated in corresponds to BODMAS (or PEMDAS, etc., if you learnt a different acronym in school).

Grouping operations with parentheses

Just like in normal mathematics, we can use parentheses to group operations, and these expressions will be given precedence in the order of evaluation.

```
(4 - 5) * (5 + 3) / 2
```

In the line above, `(4 - 5)` is evaluated first, then `(5 + 3)`. The multiplication is then performed, and finally the division. The result is an integer with the value `-4`.

We can put this whole thing inside the parentheses when calling `print` if we want to see the output:

```
print((4 - 5) * (5 + 3) / 2)
```

Style note

When we write binary operators (operators with two operands) like `+`, `-`, and `*`, we should put a space on either side of the operator. We don’t want to be writing code like this:

```
1/11+4-3/2
```

Adding spaces makes everything much easier to read:

```
1 / 11 + 4 - 3 / 2
```

However, it can sometimes be appropriate to forego this space to group operations together. For example, like this:

```
1/11 + 4 - 3/2
```

Grouping the operations in this way helps to remind readers of the order of operations, making clear that we're not dealing with something like `1 / (11 + 4)`.

Exercises

Now that we've been through the material for this post, here are a few exercises so that you can practice:

1. Print your age to the console.
2. Calculate and print the number of days, weeks, and months in 27 years. Don't worry about leap years!
3. Calculate and print the area of a circle with a radius of 5 units. You can be as accurate as you like with the value of pi.

As part of this exercise, you may want to look into what other operators are available to us in Python.

When you're done, you can find solutions to the exercises here.

I hope you found this first post helpful and interesting! See you on Day 2, where you'll tackle strings and handling user input!