builtin

July 5, 2018

1 Builtin

Builtin data types are the *core* data types that a programming language provides, **right out of the box**. In this scenario, you do **not** need to `import <package>` in py or `import "<package>"` in js. There are just there for you!

These are the stuffs that **Guido van Rossum** and **Brendan Eich** created, in the beginning. We will dig into some of them, here.

Please bear with me. And hopefully in the end you will be smilling after understanding some awesome builtins in both python and javascript.

1.1 Number

A computer is just a computing unit, so let's start with numbers, as it's *everywhere*.

Here, we focus on, **how python and javascript deal with numbers?**

In python, numbers were dealt as two types:

- `int` which is a signed integer with unlimited precision (well...not completely true though, as it is still being limited by your memory space to store every bit of your *grossly* large number)
- `float` which is a finite precision, implemented as IEEE-754 double precision (64-bit) or `double` from C, ported towards cpython.

But, what the heck is it about 64-bit?

bit is a smallest unit in our old-fashioned computing system so it's either 0 or 1, basically only two states. With 64bit-machine it means: 1 bit is used for the sign, 11 bits for exponent and 52 bits are for the fraction or significant. There you have it to store floats.

Taken this with the way our encoding works, we end up with 53 bits of binary precision for the float. What does it mean? It means that up to 2^{53}, it could be converted to float without losing any precision. Or the biggest number, limited up to 2^{53}, that can be represented correctly by both `float` and `int` types. There lies the limit of doing arithmetic with floats.

Meanwhile in javascript, number has only a single representation as 64-bit, like python float, which is `Number`, in which `Number.isInteger(1.0) === Number.isInteger(1)`. To get `Number.isInteger()` returns false you need to add at least 2 decimals, such as the following expression would return true, `Number.isInteger(1.52) === false`

Great, it is only one type of number, i.e, `Number` in js, meaning it is simple and convenient. Unfortunately..., the precision is limited to 53 bits, i.e, 2^{53}. Beyond that, it will come as a surprise as we will see later on.

The safe range integer, x, for javascript: $-2^{53} + 1 < x < 2^{53} - 1 $

Note that this basic repo does not touch on any kind of detail implementations of floating-point. But for your curiosity, please read further to Goldberg, 1991, *What every computer scientist should know about floating-point arithmetic*.

Enough with the theory, here we go next to put our hands dirty, playing with number types and also a few of different literal types of numbers, and *last but not least*, a demo of rounding-error we should be **aware of** to avoid floating-number *errors* in both languages

1.1.1 `binary`, `octal`, and `hexa` types

Both in py and js, the syntax to represent these different types are similar:

- `binary: 0b`
- `octal: 0o`
- `hexa: 0x`

Below are the gist for different base number conversions.

```
In [16]: # convert to decimal
         great_bin = 0b1111100111
         great_oct = 0o1747
         great_hex = 0x3E7

         print(f"0b1111100111 is {great_bin}")
         print(f"0o1747 is {great_oct}")
         print(f"0x3E7 is {great_hex}")

         # convert from decimal
         print(f"binary of 9999 {bin(999)[2:]}")
         print(f"oct of 9999 {oct(999)[2:]}")
         print(f"hex of 9999 {hex (999)[2:]}")
```

```
0b1111100111 is 999
0o1747 is 999
0x3E7 is 999
binary of 9999 1111100111
oct of 9999 1747
hex of 9999 3e7
```

```
In [1]: // convert to decimal
        let great_bin =  Number(0b1111100111);
        let great_oct = Number(0o1747);
        let great_hex = Number(0x3E7);

        console.log(`0b1111100111 is ${great_bin}`);
        console.log(`0o1747 is ${great_oct}`);
        console.log(`0x3E7 is ${great_hex}`);

        // convert from decimal
        console.log(`binary of 999 ${Number(999).toString(base=2)}`);
        console.log(`oct of 999 ${Number(999).toString(base=8)}`);
        console.log(`hex of 999 ${Number(999).toString(base=16)}`);
```

```
0b1111100111 is 999
0o1747 is 999
0x3E7 is 999
binary of 999 1111100111
oct of 999 1747
hex of 999 3e7
```

```
Out[1]: undefined
```

1.1.2 Rounding errors

Python, fortunately does not have a integer problem like javascript. Unfortunately, both languages suffer off-precision with floating numbers

```
In [11]: # python number limit
         from IPython.core.interactiveshell import InteractiveShell
         InteractiveShell.ast_node_interactivity = "all"

         import sys
         sys.float_info
         print("arimethic with int works as expected")
         print(f"2**53 is {2**53}")
         print(f"2**53 + 1 is {2**53 + 1}")
         print(f"2**53 + 2 is {2**53 + 2}")
         print(f"2**53 + 3 is {2**53 + 3}")

         # below is the scenario
         print("arimethic with float works as surprised")
         print(f"2**53 is {float(2**53)}")
         print(f"2**53 + 1 is {float(2**53 + 1)} which is wrong")
         print(f"2**53 + 2 is {float(2**53 + 2)} which is correct again")
         print(f"2**53 + 3 is {float(2**53 + 3)} which is wrong again")

         print("decimal arimethic is represented as finite precision")
         print(f"0.9-0.6 is {0.9-0.6}")
```

```
Out[11]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.225073
```

```
arimethic with int works as expected
2**53 is 9007199254740992
2**53 + 1 is 9007199254740993
2**53 + 2 is 9007199254740994
2**53 + 3 is 9007199254740995
arimethic with float works as surprised
```

```
2**53 is 9007199254740992.0
2**53 + 1 is 9007199254740992.0 which is wrong
2**53 + 2 is 9007199254740994.0 which is correct again
2**53 + 3 is 9007199254740996.0 which is wrong again
decimal arimethic is represented as finite precision
0.9-0.6 is 0.30000000000000004
```

```
In [1]: // javascript number limit
        Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1
        Number.MIN_SAFE_INTEGER === Math.pow(2, 53) * -1 + 1

        console.log("arimethic with number works as surprised");
        console.log(`2**53 is ${2**53}`);
        console.log(`2**53 + 1 is ${2**53 + 1} which is wrong`);
        console.log(`2**53 + 2 is ${2**53 + 2} which correct`);
        console.log(`2**53 + 3 is ${2**53 + 3} which is wrong`);

        console.log("decimal arimethic is represented as finite precision");
        console.log(`0.9-0.6 is ${0.9-0.6}`);
```

```
arimethic with number works as surprised
2**53 is 9007199254740992
2**53 + 1 is 9007199254740992 which is wrong
2**53 + 2 is 9007199254740994 which correct
2**53 + 3 is 9007199254740996 which is wrong
decimal arimethic is represented as finite precision
0.9-0.6 is 0.30000000000000004
```

```
Out[1]: undefined
```