05-Built-in-Scalar-Types

September 7, 2016

1 Built-In Types: Simple Values

When discussing Python variables and objects, we mentioned the fact that all Python objects have type information attached. Here we'll briefly walk through the built-in simple types offered by Python. We say "simple types" to contrast with several compound types, which will be discussed in the following section.

Python's simple types are summarized in the following table:

Python Scalar Types

Type	Example	Description
`int`	`x = 1`	integers (i.e., whole numbers)
`float`	`x = 1.0`	floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

We'll take a quick look at each of these in turn.

1.1 Integers

The most basic numerical type is the integer. Any number without a decimal point is an integer:

```
In [1]: x = 1
        type(x)
```

```
Out[1]: int
```

Python integers are actually quite a bit more sophisticated than integers in languages like `C`. C integers are fixed-precision, and usually overflow at some value (often near 2^{31} or 2^{63}, depending on your system). Python integers are variable-precision, so you can do computations that would overflow in other languages:

```
In [2]: 2 ** 200
```

```
Out[2]: 1606938044258990275541962092341162602522202993782792835301376
```

Another convenient feature of Python integers is that by default, division up-casts to floating-point type:

```
In [3]: 5 / 2

Out[3]: 2.5
```

Note that this upcasting is a feature of Python 3; in Python 2, like in many statically-typed languages such as C, integer division truncates any decimal and always returns an integer:

```
# Python 2 behavior
>>> 5 / 2
2
```

To recover this behavior in Python 3, you can use the floor-division operator:

```
In [4]: 5 // 2

Out[4]: 2
```

Finally, note that although Python 2.*x* had both an `int` and `long` type, Python 3 combines the behavior of these two into a single `int` type.

1.2 Floating-Point Numbers

The floating-point type can store fractional numbers. They can be defined either in standard decimal notation, or in exponential notation:

```
In [5]: x = 0.000005
        y = 5e-6
        print(x == y)

True


In [6]: x = 1400000.00
        y = 1.4e6
        print(x == y)

True
```

In the exponential notation, the `e` or `E` can be read "...times ten to the...", so that `1.4e6` is interpreted as 1.4×10^6.

An integer can be explicitly converted to a float with the `float` constructor:

```
In [7]: float(1)

Out[7]: 1.0
```

1.2.1 Aside: Floating-point precision

One thing to be aware of with floating point arithmetic is that its precision is limited, which can cause equality tests to be unstable. For example:

```
In [8]: 0.1 + 0.2 == 0.3
```

```
Out[8]: False
```

Why is this the case? It turns out that it is not a behavior unique to Python, but is due to the fixed-precision format of the binary floating-point storage used by most, if not all, scientific computing platforms. All programming languages using floating-point numbers store them in a fixed number of bits, and this leads some numbers to be represented only approximately. We can see this by printing the three values to high precision:

```
In [9]: print("0.1 = {0:.17f}".format(0.1))
        print("0.2 = {0:.17f}".format(0.2))
        print("0.3 = {0:.17f}".format(0.3))
```

```
0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

We're accustomed to thinking of numbers in decimal (base-10) notation, so that each fraction must be expressed as a sum of powers of 10:

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

In the familiar base-10 representation, we represent this in the familiar decimal expression: 0.125.

Computers usually store values in binary notation, so that each number is expressed as a sum of powers of 2:

$$1/8 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

In a base-2 representation, we can write this 0.001_2, where the subscript 2 indicates binary notation. The value $0.125 = 0.001_2$ happens to be one number which both binary and decimal notation can represent in a finite number of digits.

In the familiar base-10 representation of numbers, you are probably familiar with numbers that can't be expressed in a finite number of digits. For example, dividing 1 by 3 gives, in standard decimal notation:

$$1/3 = 0.333333333\cdots$$

The 3s go on forever: that is, to truly represent this quotient, the number of required digits is infinite!

Similarly, there are numbers for which binary representations require an infinite number of digits. For example:

$$1/10 = 0.00011001100110011\cdots_2$$

Just as decimal notation requires an infinite number of digits to perfectly represent $1/3$, binary notation requires an infinite number of digits to represent $1/10$. Python internally truncates these representations at 52 bits beyond the first nonzero bit on most systems.

This rounding error for floating-point values is a necessary evil of working with floating-point numbers. The best way to deal with it is to always keep in mind that floating-point arithmetic is approximate, and *never* rely on exact equality tests with floating-point values.

1.3 Complex Numbers

Complex numbers are numbers with real and imaginary (floating-point) parts. We've seen integers and real numbers before; we can use these to construct a complex number:

```
In [10]: complex(1, 2)

Out[10]: (1+2j)
```

Alternatively, we can use the "`j`" suffix in expressions to indicate the imaginary part:

```
In [11]: 1 + 2j

Out[11]: (1+2j)
```

Complex numbers have a variety of interesting attributes and methods, which we'll briefly demonstrate here:

```
In [12]: c = 3 + 4j

In [13]: c.real  # real part

Out[13]: 3.0

In [14]: c.imag  # imaginary part

Out[14]: 4.0

In [15]: c.conjugate()  # complex conjugate

Out[15]: (3-4j)

In [16]: abs(c)  # magnitude, i.e. sqrt(c.real ** 2 + c.imag ** 2)

Out[16]: 5.0
```

1.4 String Type

Strings in Python are created with single or double quotes:

```
In [17]: message = "what do you like?"
         response = 'spam'
```

Python has many extremely useful string functions and methods; here are a few of them:

```
In [18]: # length of string
         len(response)

Out[18]: 4

In [19]: # Make upper-case. See also str.lower()
         response.upper()
```

```
Out[19]: 'SPAM'

In [20]: # Capitalize. See also str.title()
         message.capitalize()

Out[20]: 'What do you like?'

In [21]: # concatenation with +
         message + response

Out[21]: 'what do you like?spam'

In [22]: # multiplication is multiple concatenation
         5 * response

Out[22]: 'spamspamspamspamspam'

In [23]: # Access individual characters (zero-based indexing)
         message[0]

Out[23]: 'w'
```

For more discussion of indexing in Python, see "Lists".

1.5 None Type

Python includes a special type, the `NoneType`, which has only a single possible value: `None`. For example:

```
In [24]: type(None)

Out[24]: NoneType
```

You'll see `None` used in many places, but perhaps most commonly it is used as the default return value of a function. For example, the `print()` function in Python 3 does not return anything, but we can still catch its value:

```
In [25]: return_value = print('abc')

abc


In [26]: print(return_value)

None
```

Likewise, any function in Python with no return value is, in reality, returning `None`.

1.6 Boolean Type

The Boolean type is a simple type with two possible values: `True` and `False`, and is returned by comparison operators discussed previously:

```
In [27]: result = (4 < 5)
         result

Out[27]: True

In [28]: type(result)

Out[28]: bool
```

Keep in mind that the Boolean values are case-sensitive: unlike some other languages, `True` and `False` must be capitalized!

```
In [29]: print(True, False)

True False
```

Booleans can also be constructed using the `bool()` object constructor: values of any other type can be converted to Boolean via predictable rules. For example, any numeric type is False if equal to zero, and True otherwise:

```
In [30]: bool(2014)

Out[30]: True

In [31]: bool(0)

Out[31]: False

In [32]: bool(3.1415)

Out[32]: True
```

The Boolean conversion of `None` is always False:

```
In [33]: bool(None)

Out[33]: False
```

For strings, `bool(s)` is False for empty strings and True otherwise:

```
In [34]: bool("")

Out[34]: False

In [35]: bool("abc")
```

```
Out[35]: True
```

For sequences, which we'll see in the next section, the Boolean representation is False for empty sequences and True for any other sequences

```
In [36]: bool([1, 2, 3])

Out[36]: True

In [37]: bool([])

Out[37]: False
```