03 Types, type conversions and floating point arithmetic

September 13, 2020

1 Introduction

We have thus far avoided discussing directly *types*. The '*type*' is the type of object that a variable is associated with. For example, the type could be an integer or a real number. The type affects how a computer stores the object in memory, and how operations, such as multiplication and division, are performed.

In *statically typed* languages, like C and C++, types come up from the very beginning because you usually need to specify types explicitly in your programs. Python is a *dynamically typed* language, which means that types are deduced when a program is run. This is why we have been able to postpone the discussion of types until now. It is important to have a basic understanding of types, and how types can affect how your programs behave. One can go very deep into this topic, especially for numerical computations, but we will cover the general concept from a high level, show some examples, and highlight some potential pitfalls for engineering computations.

We will look at types in particular in terms of *floating point* and *integer* numbers as these are particularly important in scientific and engineering computations. It is a dry topic but is important background information that you need to know for later. The below real-life accounts highlight what can go wrong without an awareness of types and how computers process numbers.

1.1 Patriot Missile failure and the Ariane 5 explosion

There have been numerous accidents due to programs not correctly handling types, type conversions and floating point arithmetic. Here are two examples:

1. In 1991, a US Patriot missile failed to intercept an Iraqi Scud missile at Dhahran in Saudi Arabi, leading to a loss of life. The subsequent investigation found that the Patriot missile failed to intercept the Scud missile due to a software flaw. The software developers did not account for the effects of 'floating point arithmetic'. This led to a small error in computing the time, which in turn caused the Patriot to miss the incoming Scud missile.

   We will reproduce the precise mistake the developers of the Patriot Missile software made. See https://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran for more background on the interception failure.

2. Poor programming related to how computers store numbers led in 1996 to a European Space Agency *Ariane 5* unmanned rocket exploding shortly after lift-off. The rocket payload, worth US$500 M, was destroyed. You can find background at https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure. We will reproduce the mistake, and show how a few lines code would have saved over US$500 M.

1.2 Background: bits and bytes

An important part of understanding types is appreciating how computer storage works. Computer memory is made up of *bits*, and each bit can take on one of two values - 0 or 1. A bit is the smallest building block of memory. Bits are very fine-grained, so for many computer architectures the smallest 'block' we can normally work with is a *byte*. One byte is made up of 8 bits. This why when we talk about bits, e.g. a 64-bit operating system, the number of bits will almost always be a multiple of 8 (one byte).

The 'bigger' a thing we want to store, the more bytes we need. This is important for engineering computations since the the number of bytes used to store a number determines the accuracy with which the number can be stored, or how big or small the number can be. The more bytes the greater the accuracy, but the price to be paid is higher memory usage. Also, it can be more expensive to perform operations like multiplication and division when using more bytes.

1.3 Objectives

- Introduce primitive data types (booleans, strings and numerical types)
- Type inspection
- Basic type conversion
- Introduction to pitfalls of floating point arithmetic

2 What is type?

All variables have a 'type', which indicates what the variable is, e.g. a number, a string of characters, etc. In 'statically typed' languages we usually need to be explicit in declaring the type of a variable in a program. In a dynamically typed language, such as Python, variables still have types but the interpreter can determine types dynamically.

Type is important because it determines how a variable is stored, how it behaves when we perform operations on it, and how it interacts with other variables. For example, multiplication of two real numbers is different from multiplication of two complex numbers.

3 Introspection

Before getting into types, we look at how we can check the type in Python. A powerful feature of Python is *introspection*. This means that we can probe a program to ask about the type of a variable. To check the type of a variable we use the function `type`:

[1]:
```
x = True
print(type(x))

a = 1
print(type(a))

a = 1.0
print(type(a))
```

```
<class 'bool'>
<class 'int'>
<class 'float'>
```

Note that `a = 1` and `a = 1.0` are different types! This distinction is very important for numerical computations. More on this further down.

Use `type` freely when exploring and testing to develop an understanding for what your program is doing.

4 Booleans

You have already seen the 'Boolean' type that can take on one of two values - true or false. This is the simplest type.

[2]:
```
a = True
b = False
test = a or b  # test will be True if a or b are True
print(test, type(test))
```

```
True <class 'bool'>
```

In principle, we could represent a boolean with just one bit (0 or 1 switch).

5 Strings

A string is a collection of characters. We have been using strings in previous activities for printing informative messages. In Python we create a string using single or double quotes (the choice is personal preference), e.g.

```
my_string = 'This is a string.'
```

or

```
my_string = "This is a string."
```

Below we assign a string to a variable, display the string, and then check its type:

[3]:
```
my_string = "This is a string."
print(my_string)
print(type(my_string))
```

```
This is a string.
<class 'str'>
```

We can perform many different operations on strings. We can extract a particular character as a new string:

[4]:
```
# Get 3rd character (Python counts from zero)
s2 = my_string[2]
print(s2)
print(type(s2))
```

```
i
<class 'str'>
```

or extract a range of characters:

[5]:
```
# Get first six characters, print and check type
s3 = my_string[0:6]
print(s3)
print(type(s3))

# Get last four characters and print
s4 = my_string[-4:]
print(s4)
```

```
This i
<class 'str'>
ing.
```

We can add strings together:

[6]:
```
introduction = "My name is:"
name = "Joanne"

personal_introduction = introduction + " " + name
print(personal_introduction)
```

```
My name is: Joanne
```

We can also check the length (number of characters) of a string using `len`:

[7]:
```
print(len(personal_introduction))
```

```
18
```

There are *many* more operations that can be performed on strings. We will see more in later activities.

6 Numeric types

Numeric types are important in many computing applications, and particularly in scientific and engineering programs. Python 3 has three native numerical types:

- integers (`int`)
- floating point numbers (`float`)
- complex numbers (`complex`)

This is typical for most programming languages, although there can be some subtle differences.

6.1 Integers

Integers (`int`) are whole numbers, and can be postive or negative. Integers should be used when a value can only take on a whole number, e.g. the year, or the number of students following this

course. Python infers the type of a number from the way we input it. It will infer an `int` if we assign a number with no decimal place:

[8]:
```
a = 2
print(type(a))
```

```
<class 'int'>
```

If we add a decimal point, the variable type becomes a `float` (more on this later)

[9]:
```
a = 2.0
print(type(a))
```

```
<class 'float'>
```

Integer operations that result in an integer, such as multiplying or adding two integers, are performed exactly (there is no error). This does however depend on a variable having enough memory (sufficient bytes) to represent the result.

6.1.1 Integer storage and overflow

In most languages, a fixed number of bits are used to store a given type of integer. In C and C++ a standard integer (`int`) is usually stored using 32 bits (it is possible to declare shorter and longer integer types). The largest integer that can be stored using 32 bits is $2^{31} - 1 = 2,147,483,647$. We explain later where this comes from. The message for now is that for a fixed number of bits, there is a bound on the largest number that can be represented/stored.

Integer overflow Integer overflow is when an operation creates an integer that is too big to be represented by the given integer type. For example, attempting to assign $2^{31} + 1$ to a 32-bit integer will cause an overflow and potentially unpredictable program response. This would usually be a *bug*.

The Ariane 5 rocket explosion in 1996 was caused by integer overflow. The rocket navigation software was taken from the older, slower Ariane 4 rocket. The program assigned the rocket speed to a 16-bit integer (the largest number a 16-bit integer can store is $2^{15} - 1 = 32767$), but the Ariane 5 could travel faster than the older generation of rocket and the speed value exceeded 32767. The resulting integer overflow led to failure of the rocket's navigation system and explosion of the rocket; a very costly rocket and a very expensive payload were destroyed. We will reproduce the error that caused this failure when we look at *type conversions*.

Python avoids integer overflows by dynamically changing the number of bits used to represent an integer. You can inspect the number of bits required to store an integer in binary (not including the bit for the sign) using the function bit_length:

[10]:
```
a = 8
print(type(a))
print(a.bit_length())
```

```
<class 'int'>
4
```

We see that 4 bits are necessary to represent the number 8. If we increase the size of the number dramatically by raising it to the power of 12:

[11]:
```
b = a**12
print(b)
type(b)
print(b.bit_length())
```

```
68719476736
37
```

We see that 37 bits are required to represent the number. If the `int` type was limited to 32 bits for storing the value, this operation would have caused an overflow.

Gangnam Style In 2014, Google switched from 32-bit integers to 64-bit integers to count views when the video "Gangnam Style" was viewed more than 2,147,483,647 times, which is the limit of 32-bit integers (see https://plus.google.com/+YouTube/posts/BUXfdWqu86Q).

Boeing 787 Dreamliner bug Due to an integer overflow bug, the electricity generators on a Boeing 787 will shut down if the plane is powered continuously for 248 days, due to an overflow. The 'quick fix' was to make sure that generator control units do not operate for more than 248 days. See https://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-control and https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf for background.

6.2 Floating point storage

Most engineering calculations involve numbers that cannot be represented as integers. Numbers that have a decimal point are stored using the `float` type. Computers store floating point numbers by storing the sign, the significand (also known as the mantissa) and the exponent, e.g.: for 10.45

$$10.45 = \underbrace{+}_{\text{sign}} \underbrace{1045}_{\text{significand}} \times \underbrace{10^{-2}}_{\text{exponent}=-2}$$

Python uses 64 bits to store a `float` (in C and C++ this is known as a `double`). The sign requires one bit, and there are standards that specify how many bits should be used for the significand and how many for the exponent.

Since a finite number of bits are used to store a number, the precision with which numbers can be represented is limited. As a guide, using 64 bits a floating point number is precise to 15 to 17 significant figures. More on this, and why the Patriot missile failed, later.

6.2.1 Floats

We can declare a float by adding a decimal point:

[12]:
```
a = 2.0
print(a)
print(type(a))
```

```
b = 3.
print(b)
print(type(b))
```

```
2.0
<class 'float'>
3.0
<class 'float'>
```

or by using `e` or `E` (the choice between `e` and `E` is just a matter of taste):

[13]:
```
a = 2e0
print(a, type(a))

b = 2e3
print(b, type(b))

c = 2.1E3
print(c, type(c))
```

```
2.0 <class 'float'>
2000.0 <class 'float'>
2100.0 <class 'float'>
```

6.2.2 Complex numbers

A complex number is a more elaborate float with two parts - the real and imaginary components. We can declare a complex number in Python by adding `j` or `J` after the complex part of the number:

[14]:
```
a = 2j
print(a, type(a))

b = 4 - 3j
print(b, type(b))
```

```
2j <class 'complex'>
(4-3j) <class 'complex'>
```

The usual addition, subtraction, multiplication and division operations can all be performed on complex numbers. The real and imaginary parts can be extracted:

[15]:
```
print(b.imag)
print(b.real)
```

```
-3.0
4.0
```

and the complex conjugate can be taken:

[16]:
```
print(b.conjugate())
```

```
(4+3j)
```

We can compute the modulus of a complex number using `abs`:

[17]:
```
print(abs(b))
```

```
5.0
```

More generally, `abs` returns the absolute value, e.g.:

[18]:
```
a = -21.6
a = abs(a)
print(a)
```

```
21.6
```

7 Type conversions (casting)

We can often change between types. This is called *type conversion* or *type casting*. In some cases it happens implicitly, and in other cases we can instruct our program to change the type.

If we add two integers, the results will be an integer:

[19]:
```
a = 4
b = 15
c = a + b
print(c, type(c))
```

```
19 <class 'int'>
```

However, if we add an `int` and a `float`, the result will be a float:

[20]:
```
a = 4
b = 15.0  # Adding the '.0' tells Python that it is a float
c = a + b
print(c, type(c))
```

```
19.0 <class 'float'>
```

If we divide two integers, the result will be a `float`:

[21]:
```
a = 16
b = 4
c = a/b
print(c, type(c))
b = 2
```

```
4.0 <class 'float'>
```

When dividing two integers, we can do 'integer division' using `//`, e.g.

[22]:
```
a = 16
b = 3
c = a//b
print(c, type(c))
```

```
5 <class 'int'>
```

in which case the result is an `int`.

In general, operations that mix an `int` and `float` will generate a `float`, and operations that mix an `int` or a `float` with `complex` will return a `complex` type. If in doubt, use `type` to experiment and check.

7.1 Explicit type conversion

We can explicitly change the type (perform a cast), e.g. cast from an `int` to a `float`:

[23]:
```
a = 1
print(a, type(a))

a = float(a)  # This converts the int associated with 'a' to a float, and␣
 ↪assigns the result to the variable 'a'
print(a, type(a))
```

```
1 <class 'int'>
1.0 <class 'float'>
```

Going the other way,

[24]:
```
y = 1.99
print(y, type(y))

z = int(y)
print(z, type(z))
```

```
1.99 <class 'float'>
1 <class 'int'>
```

Note that rounding is applied when converting from a `float` to an `int`; the values after the decimal point are discarded. This type of rounding is called 'round towards zero' or 'truncation'.

A common task is converting numerical types to-and-from strings. We might read a number from a file as a string, or a user might input a value which Python reads in as a string. Converting a float to a string:

[25]:
```
a = 1.023
b = str(a)
print(b, type(b))
```

```
1.023 <class 'str'>
```

and in the other direction:

[26]:
```
a = "15.07"
b = "18.07"

print(a + b)
print(float(a) + float(b))
```

```
15.0718.07
33.14
```

If we tried

```
print(int(a) + int(b))
```

we could get an error that the strings could not be converted to `int`. It works in the case:

[27]:
```
a = "15"
b = "18"
print(int(a) + int(b))
```

```
33
```

since these strings can be correctly cast to integers.

7.2 Ariane 5 rocket explosion and type conversion

The Ariane 5 rocket explosion was caused by an integer overflow. The speed of the rocket was stored as a 64-bit float, and this was converted in the navigation software to a 16-bit integer. However, the value of the float was greater than 32767, the largest number a 16-bit integer can represent, and this led to an overflow that in turn caused the navigation system to fail and the rocket to explode.

We can demonstrate what happened in the rocket program. We consider a speed of 40000.54 (units are not relevant to what is being demonstrated), stored as a `float` (64 bits):

[28]:
```
speed_float = 40000.54
```

If we first convert the float to a 32-bit `int` (we use NumPy to get integers with a fixed number of bits, more on NumPy in a later notebook):

[29]:
```
import numpy as np
speed_int = np.int32(speed_float)  # Convert the speed to a 32-bit int
print(speed_int)
```

```
40000
```

The conversion behaves as we would expect. Now, if we convert the speed from the `float` to a 16-bit integer:

[30]:
```
speed_int = np.int16(speed_float)
print(speed_int)
```

```
-25536
```

We see clearly the result of an integer overflow since the 16-bit integer has too few bits to represent the number 40000.

The Ariane 5 failure would have been averted with pre-launch testing and the following few lines:

[31]:
```
if abs(speed_float) > np.iinfo(np.int16).max:
    print("***Error, cannot assign speed to 16-bit int. Will cause overflow.")
    # Call command here to exit program
else:
    speed_int = np.int16(speed_float)
```

```
***Error, cannot assign speed to 16-bit int. Will cause overflow.
```

These few lines and careful testing would have saved the $500M payload and the cost of the rocket.

The Ariane 5 incident is an example not only of a poor piece of programming, but also very poor testing and software engineering. Careful pre-launch testing of the software would have detected this problem. The program should have checked the value of the velocity before performing the conversion, and triggered an error message that the type conversion would cause an overflow.

8 Binary representation and floating point arithmetic

8.1 Binary (base 2) representation

Computers store data using 'bits', and a bit is a switch that can have a value of 0 or 1. This means that computers store numbers in binary (base 2), whereas we almost always work with decimal numbers (base 10). For example, the binary number 110 is equal to $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = 6$ (read 110 right-to-left). Below is a table with decimal (base 10) and the corresponding binary (base 2) representation of some numbers. See https://en.wikipedia.org/wiki/Binary_number if you want to learn more.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

To represent any integer, all we need are enough bits to store the binary representation. If we have n bits, the largest number we can store is $2^{n-1} - 1$ (the power is $n - 1$ because we use one bit to store the sign of the integer).

We can display the binary representation of an integer in Python using the function `bin`:

[32]:
```
print(bin(2))
print(bin(6))
print(bin(110))
```

```
0b10
0b110
0b1101110
```

The prefix `0b` is to denote that the representation is binary.

8.2 Floating point numbers

We introduced the representation

$$10.45 = \underbrace{+}_{\text{sign}} \underbrace{1045}_{\text{significand}} \times \underbrace{10^{-2}}_{\text{exponent}}$$

earlier. However, this was a little misleading because computers do not use base 10 to store the significand and the exponent, but base 2.

When using the familiar base 10, we cannot represent 1/3 exactly as a decimal. If we liked using base 3 (ternary numeral system) for our mental arithmetic (which we really don't), we could represent 1/3 exactly. However, fractions that are simple to represent exactly in base 10 might not be representable in another base. A consequence is that fractions that are simple in base 10 cannot necessarily be represented exactly by computers using binary.

A classic example is $1/10 = 0.1$. This simple number cannot be represented exactly in binary. On the contrary, $1/2 = 0.5$ can be represented exactly. To explore this, let's assign the number 0.1 to the variable `x` and print the result:

[33]:
```
x = 0.1
print(x)
```

```
0.1
```

This looks fine, but the `print` statement is hiding some details. Asking the `print` statement to use 30 characters we see that `x` is not exactly 0.1:

[34]:
```
print('{0:.30f}'.format(x))
```

```
0.100000000000000005551115123126
```

The difference between 0.1 and the binary representation is the *roundoff error* (we'll look at print formatting syntax in a later activity). From the above, we can see that the representation is accurate to about 17 significant figures.

Checking for 0.5, we see that it appears to be represented exactly:

[35]:
```
print('{0:.30f}'.format(0.5))
```

```
0.500000000000000000000000000000
```

The round-off error for the 0.1 case is small, and in many cases will not present a problem. However, sometimes round-off errors can accumulate and destroy accuracy.

8.2.1 Example: inexact representation

It is trivial that

$$x = 11x - 10x$$

If $x = 0.1$, we can write

$$x = 11x - 1$$

Now, starting with $x = 0.1$ we evaluate the right-hand side to get a 'new' x, and use this new x to then evaluate the right-hand side again. The arithmetic is trivial: x should remain equal to 0.1. We test this in a program that repeats this process 20 times:

[36]:
```
x = 0.1
for i in range(20):
    x = x*11 - 1
    print(x)
```

```
0.10000000000000009
0.10000000000000098
0.10000000000001075
0.10000000000011822
0.10000000000130038
0.1000000000143042
0.10000000015734622
0.10000000173080847
0.10000001903889322
0.10000020942782539
0.10000230370607932
0.10002534076687253
0.10027874843559781
0.1030662327915759
0.13372856070733485
0.4710141677806834
4.181155845587517
44.992714301462684
493.9198573160895
5432.118430476985
```

The solution blows up and deviates widely from $x = 0.1$. Round-off errors are amplified at each step, leading to a completely wrong answer. The computer representation of 0.1 is not exact, and every time we multiply 0.1 by 11, we increase the error by around a factor of 10 (we can see above that we lose a digit of accuracy in each step). You can observe the same issue using spreadsheet programs.

If we use $x = 0.5$, which can be represented exactly in binary:

[37]:
```
x = 0.5
for i in range(20):
    x = x*11 - 5
    print(x)
```

```
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
0.5
```

The result is exact in this case.

By default, Python uses 64 bits to store a float. We can use the module NumPy to create a float that uses only 32 bits. Testing this for the $x = 0.1$ case:

[38]:
```
x = np.float32(0.1)
for i in range(20):
    x = x*11 - 1
    print(x)
```

```
0.10000001639127731
0.10000018030405045
0.1000019833445549
0.10002181679010391
0.10023998469114304
0.1026398316025734
```

```
0.12903814762830734
0.41941962391138077
3.6136158630251884
38.74977449327707
425.2475194260478
4676.722713686526
51442.949850551784
565871.4483560696
6224584.931916766
68470433.25108442
753174764.7619286
8284922411.381214
91134146524.19336
1002475611765.127
```

The error blows up faster in this case compared to the 64 bit case - using 32 bits leads to a poorer approximation of 0.1 than when using 64 bits.

Note: Some languages have special tools for performing decimal (base 10) arithmetic (e.g., https://docs.python.org/3/library/decimal.html). This would, for example, allow 0.1 to be represented exactly. However, decimal is not the 'natural' arithmetic of computers so operations in decimal could be expected to be much slower.

8.3 Patriot Missile Failure

The inexact representation of 0.1 was the cause of the software error in the Patriot missile system (see preamble to this notebook). The missile system tracked time from boot (system start) using an integer counter that was incremented every 1/10 of a second. To get the time in seconds, the missile software multiplied the counter by the float representation of 0.1. The control software used 24 bits to store floats. The round-off error due to the inexact representation of 0.1 lead to an error of 0.32 s after 100 hours of operation (time since boot), which due to the high velocity of the missile was enough to cause failure to intercept the incoming Scud.

We don't have 24-bit floats in Python, but we can test with 16, 32 and 64 bit floats. We first compute what the system counter (an integer) would be after 100 hours:

[39]:
```
# Compute internal system counter after 100 hours (counter increments every 1/
↪10 s)
num_hours = 100
num_seconds = num_hours*60*60
system_counter = num_seconds*10  # system clock counter
```

Converting the system counter to seconds using different representations of 0.1:

[40]:
```
# Test with 16 bit float
dt = np.float16(0.1)
time = dt*system_counter
print("Time error after 100 hours using 16 bit float (s):", abs(time -␣
↪num_seconds))
```

```
# Test with 32 bit float
dt = np.float32(0.1)
time = dt*system_counter
print("Time error after 100 hours using 32 bit float (s):", abs(time -␣
 ↪num_seconds))

# Test with 64 bit float
dt = np.float64(0.1)
time = dt*system_counter
print("Time error after 100 hours using 64 bit float (s):", abs(time -␣
 ↪num_seconds))
```

```
Time error after 100 hours using 16 bit float (s): 87.890625
Time error after 100 hours using 32 bit float (s): 0.005364418029785156
Time error after 100 hours using 64 bit float (s): 0.0
```

The time computation with 16-bit floats is more than a minute off after 100 hours! The stop-gap measure for the Patriot missiles at the time was to reboot the missile systems frequently, thereby resetting the system counter and reducing the time error.

9 Summary

The key points from this activity are:

- The size of an integer that a computer can store is determined by the number of bits used to represent the integer.
- Computers do not perform exact arithmetic with non-integer numbers. This does not usually cause a problem, but it can in cases. Problems can often be avoided with careful programming.
- Be thoughtful when converting between types - undesirable consequences can follow when careless with conversions.

10 Exercises

Complete now the 03 Exercises notebook.