# Decimals, Floats, and Floating Point Arithmetic

Floating point numbers like 12.345 are a basic type, but there are some complications due to their inexactness. This section may be deferred until you actually need numbers other than integes.

## Floats, Division, Mixed Types

As you moved on in school from your first integer division to fractions and decimals, you probably thought of 6/8 as a fraction and could convert to a decimal .75. Python can do decimal calculations, too, approximately.

Try all set-off lines in this section in the Shell:

6/8

6/3

2.3/25.7

There is more going on here than meets the eye. As you should know, decimal representations of values can be a pain. They may not be able to be expressed with a finite number of characters. Try

2/3

Also, as you may have had emphasized in science class, real number measurements are often not exact, and so the results of calculations with them are not exact. In fact there are an infinite number of real number just between 0 and 1, and a computer is finite. It cannot store all those numbers exactly! On the other hand, Python does store integers exactly (well at least far past the number of atoms in the universe - eventually even integers could get too big to store in a computer). The difference in the way integers and decimals are stored and processed leads to decimals and integers being differenttypes in Python. Try

type(3.5)

Note that 3.5 is of type ‘float’, not ‘decimal’. There are several reasons for that name having to do with the actual way the type is stored internally. “Decimal” implies base ten, our normal way for writing numbers with ten digits 0,1,2,3,4,5,6,7,8,9. Computers actually use base two, with only two symbols 0,1. (Did you note what symbols were in the machine language in [Context](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/context.html#context)?) Also floats use an encoding something like scientific notation from science class, with exponents that allow the decimal point to move or “float”, as in the decimal case: 2345.6 = (2.3456)103

Try

type(-2)

type(-2.0)

Even a number that is actually an integer can be represented in the float type if a decimal point is included.

Always be sure to remember that floats may not be exact. The use of base two makes this true even in cases where decimal numbers can be expressed exactly! More on that in [String Formats for Float Precision](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/float.html#precision-formats).

It is sometimes important to know the numeric type of the result of a binary operation. Any combination of +, -, and \* with operands of type int produces an int. If there is an operation /, or if either operand is of type float, the result is float. Try each in the Shell (and guess the resulting type): [[1]](http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/float.html#id2)

3.3 - 1.1

2.0 + 3

2\*2.5

|  |  |
| --- | --- |
|  | Python 3.1 does what you would expect mathematically with an expression like (1/2)\*6.5  Caution: This is not the case in other common languages like Java and C++ (or with versions of Python before 3.0). They treat the / operation with integers like the current Python //, so the result of the expression above is 0, since 1//2 is 0. |

## Exponentiation, Square Roots

Exponentiation is finding powers. In mathematical notation, (3)(3)(3)(3)=34. In Python there is no fancy typography with raised exponent symbols like the 4, so Python uses \*\* before a power: Try in the Shell:

3\*\*4

5\*2\*\*3

If you expected 1000 for the second expression, remember exponentiation has even higher precedence than multiplication and division: 2\*\*3 is 2\*2\*2 or 8, and 5\*8 is 40.

Exponents do not need to be integers. A useful example is the 0.5 power: it produces a square root. Try in the Shell:

9\*\*.5

2\*\*.5

The result of a power operation is of int type only if both parameters are integers and the correct result is an integer.

## String Formats for Float Precision

You generally do not want to display a floating point result of a calculation in its raw form, often with an enormous number of digits after the decimal point, like 23.457413902458498. You are likely to prefer rounding it to something like 23.46. There are two approaches.

First there is a format function (not method) with a second parameter allowed to specialize the formatting of objects as strings. Read the following example interpreter sequence showing possibilities when a float is being formatted:

**>>>** x = 23.457413902458498

**>>>** s = format(x, '.5f')

**>>>** s

'23.45741'

**>>>** format(x, '.2f')

'23.46'

**>>>** x

23.457413902458498

Note that the results are rounded not truncated: the result to two places is 23.46, not 23.45. The formatting string '.5f' means round to 5 places after the decimal point. Similarly '.2f' means round to two decimal places.

**Warning**

This format function returns the formatted string. It does not change the parameters. As a complete statement in a program format(x,'.2f'), is useless: The '23.46' gets returned and thrown away, with no effect on x.

The first version, saving the formatted value to s, will allow the formatted string to be used again (as s).

This rounding notation can also be placed after a colon inside the braces of format strings, for use with the string format method. Recall there are many ways to indicate what values to substitute into a format string. One way is just to omit any reference to the variables and substitute parameters in order. Separate from the value to substitute, and following any notation for it, you can put a colon : and the formatting information we used in the simple format method above (like .5f. but with NO quotes):

**>>>** x = 2.876543

**>>>** 'longer: {:.5f}, shorter: {:.3f}.'.format(x, x)

'longer: 2.87654, shorter: 2.877.'

The instructions for the data to insert can also be given by position index:

**>>>** x = 2.876543

**>>>** 'longer: {0:.5f}, shorter: {0:.3f}.'.format(x)

'longer: 2.87654, shorter: 2.877.'

or using dictionary keys:

**>>>** x = 2.876543

**>>>** 'longer: {x:.5f}, shorter: {x:.3f}.'.format(\*\*locals())

'longer: 2.87654, shorter: 2.877.'

In each of these approaches, the colon and formatting specification come at the end of the expression inside the braces, just before the closing }. This follows the { and symbols (if any) identifying what value to use for the substitution.

There are many more fancy formatting options for the string format method that we will not discuss.

Going to the opposite extreme, and using formatting with many digits, you can check that Python does not necessarily remember simple decimal numbers exactly:

**>>>** format(.1, '.20f')

0.10000000000000000555

**>>>** format(.2, '.20f')

'0.20000000000000001110'

**>>>** format(.1 + .2, '.20f')

'0.30000000000000004441'

**>>>** format(.3, '.20f')

'0.29999999999999998890'

Python stores the numbers correctly to about 16 or 17 digits. You may not care about such slight errors, but you will be able to check in Chapter 3 that if Python tests the expressions .1 + .2 and .3 for equality, it decides that they are not equal! In fact, as you can see above, the approximations that Python stores for the two expressions are not exactly equal.

**Warning**

Do not depend on the exactness of floating point arithmetic, even for apparently simple expressions!

The floating point formatting code in this section is also in example program floatFormat.py.

### Floating Point Exercise

Write a program, discount.py, that prompts the user for an original price and for a discount percentage and prints out the new price to the nearest cent. For example if the user enters 2.89 for the price and 20 for the discount percentage, the value would be

(1 - 20/100) \* 2.89

rounded to two decimal places, 2.31. For price .65 with a 25 percent discount, the value would be

(1 - 25/100) \* .65

rounded to two decimal places, .49.