Computing with numbers

Read: *Chapter 3 in the textbook*

- **Numeric data types:** The numbers we are going to be using come in two different kinds (*types*)
  1. **Integers:** 3, 4, -15, 1000
  2. **Floating point (floats):** 3.0, -15.0, .25, 4.3333333

  The built-in function `type()` will tell you the type of a number:

  ```
  >>> type(3)
  >>> type(3.0)
  >>> type(-13.3333)
  ```

  (The function will tell you that, e.g., 3 is `<class 'int'>` and not `<type 'int'>`. Pretend that it is the latter, for now.) Just to see that type() works for all kinds (types) of data, try also

  ```
  >>> type("I am a string")
  ```

- As you probably suspect, operations on numbers produce results that depend on the type of the operands. For example, `+, -, *` work on integers and floats as expected:

  ```
  >>> 3+4
  >>> 3.0+4.0
  ```

  The first is an integer addition and the second is a floating point addition (and they produce an integer and a float respectively). But there is a separate symbol for float division (`/`) and for integer division (`//`):

  ```
  >>> 10.0/3.0
  >>> 10//3
  ```

  Things are not as simple as they seem though. First, notice that the result of `10.0/3.0` is
  `3.3333333333333335`
  instead of `3.3333333333333333`. Where did the last digit 5 come from? This is a result of rounding the infinite sequence of 3's that should follow (in pure Math) to a finite sequence of digits. More about rounding in a little while. Second, the result of `10//3` is the integer number 3. This is calculated according to the well known formula:

  `<dividend> = <quotient> * <divisor> + <remainder>`

  In our example <dividend>=10, <divisor>=3, <quotient>=3, and <remainder>=1. The result of `/` is the <quotient>; if you want to calculate the <remainder>, use operation `%`:

  ```
  >>> 10%3
  ```

- **Type conversion:** One would expect that the following should produce a 'wrong type' kind of error, because you apply operations to the wrong type of operands:

  ```
  >>> 10/3
  >>> 10.0//3.0
  ```

  In some programming languages, you indeed get an error. Not in Python! The first operation returns `3.3333333333333335` and the second returns `3.0`. What is going on? Python realizes that you try to apply a float operation on integral operands (first line) or vice versa (second line). Instead of producing an error message to warn you (as it should!), it **converts** the types the operands in order to make them fit. There are many ways that this can be done. Suppose you type:

  ```
  >>> x = 5.0*2
  ```

  There are two different operand types (float and int), so what should the result be?
  - x = 5*2 = 10 (an integer)
  - x = 5.0*2.0 = 10.0 (a float)

  Since floats contain integers as a subset, Python in general changes the type of operands from the particular (int) to the more general (float). The end result will be the second option (x is a float). Apply this rule yourself to the two examples above:

  ```
  >>> 10/3
  >>> 10.0//3.0
  ```

  In the second example, the operation is an *integer* one, but the result returned is a float (although Python converted the operands to int before the operation, and the result back to float). Pretty messy! In general, a programmer s**houldn't rely on the programming language to do the conversion**. Most bugs in programs (and even in hardware, like processors) are due to exactly this point: someone was expecting an integer result (for example), while the operation was producing a float. So here is a golden rule: **always do the type conversion (when needed) yourself!** Python provides conversion functions for doing exactly this. Try

  ```
  >>> int(4.9)
  >>> float(4)
  ```

  What do you get? The first one *truncates* the part after the decimal point, and the second *adds* a decimal point part. They seem rather silly though, because you can write 5 or 5.0 directly depending on whether you want your number five to be an int or a float, right? Well, what about the following line:

  ```
  >>> y = (x//2)%3
  ```

  What is y? Here the programmer intends for y to be the result of integer operations; but what if x has the value 11.66 (a float!)? Then Python will do its own conversion. Try it yourself to see what y will be:

```
>>> x = 11.66
>>> y = (x//2)%3
>>> y
```

Now try it with x = 11 (an int). In the first case y is a float (y=2.0) and in the second it is an int (y=2), just like the programmer was expecting. How can you protect yourself from Python's conversions? Force the conversion yourself to the direction **you** want!

```
>>> y = (int(x)//2)%3
```

Now, no matter what x was before you used it, you force Python to use its value as an integer!

```
>>> x = 11.66
>>> y = (int(x)//2)%3
>>> y
```

Note that `int(x)` just affects how you treat x's value in your expression, not x itself. see for yourself:

```
>>> x
```

x is still 11.66! If you want your type conversion to change x itself, try assignment:

```
>>> x = int(x)
>>> x
```

x must be an integer now (x=11).

- **Rounding:** The way `int()` works is not satisfactory if we want rounding to the closest integer. Many times we want 11.76 to be rounded to 12 and not to 11. This can be achieved with function `round()`. Its general form is `round(<value>, n)`, where `n` is the number of digits after the decimal point of `<value>` you want to keep; the rest of the digits are rounded up or down, depending on which value is closer. Try the following:

  ```
  >>> round(123.45678)
  >>> round(123.45678, 2)
  >>> round(123.45456, 2)
  ```

  Notice the difference in the result of the last two examples to see the different way 123.45xxx is rounded depending on whether xxx=678 or xxx=456.
- **Math Library:** Many of the mathematical functions that are heavily used are not part of the built-in Python machinery. We need to import the `math` library in order to have access to functions and constants such as the square root of a number or pi:

  ```
  >>> import math
  >>> math.sqrt(16.0)
  ```

```
>>> math.sqrt(-16.0)
>>> math.pi
```

Table 3.2 in the text has a list of common math functions/values. Since we use them often, and we don't want to continuously specify that are from the math library, we can import everything from the math library into our own "environment" using the `from – import` trick:

```
>>> from math import *
>>> sqrt(16.0)
>>> x = pi
>>> round(x,2)
```

Recall that the asterisk (*) in the first line means "everything". An example of using the `math` library in a program is `quadratic.py` in the text. What is missing from the book implementation? *(Hint: We expect the `a,b,c` numbers provided by the user to be floats...should we rely on the user's good will?)*

- **An example:** To put everything together, let's look at the computation of the *factorial* function, defined for every positive integer n as follows:

  `fact(n) = 1*2*3*...*n`.

  Suppose that you want to compute fact(5) = 1*2*3*4*5 = 120. The book proposes building the final result by incorporating one number from 1,2,3,4,5 at a time; this suggests starting with an initial value of 1, followed by a definite loop on the sequence 2,3,4,5:

```
fact = 1
for factor in range(2,6):
    fact = fact * factor
```

  Recall that `range(2,6)` produces a sequence of integers starting from 2 and ending at 6 *without 6*. Or, alternatively, you can replace counting 2,3,4,5 with counting 5,4,3,2 by using `range(5,1,-1)`. The full example (extended to getting `n` as input from the user) is `factorial.py` in the text.
- **Recursion:** Here we mention that there is another way to define the factorial function:

```
fact(1) = 1
fact(n) = fact(n-1) * n,  if n>1
```

  Note that this definition looks a lot like the piece of code we wrote above, except that it doesn't use a loop. Instead, the function is defined in terms of itself! In other words, the definition of the function uses the function itself, except for the case of n=1 (which we call the **base case**). This sounds cyclical, but it isn't: Notice that fact(n) is defined using fact(n-1). We don't know the value of fact(n-1) yet, but we know that if we follow the new definition of fact() and by setting n to be n-1, it is defined in terms of fact(n-2):

  `fact(n-1) = fact(n-2) * (n-1)`

  Of course, now we don't know the value of fact(n-2), but we can calculate it using the general

definition setting n to be n-2. And so on, until at some point we need the value of fact(2), which is defined to be

```
fact(2) = fact(1) * 2
```

But now we know fact(1)=1! It is our base case! So we can now calculate fact(2), and use it to calculate fact(3),..., all the way to fact(n). This way of defining the factorial function is called a **recursive definition**. Recursion is probably the most used term in Computer Science.

Practice problems

1. Do all problems in the textbook.
2. Verify that Python calculation satisfies the well-known division equation:

   dividend = divisor * quotient + remainder

   Do that by looking at the division of 32 by 11. In other words, calculate the quotients by performing 32//11 and the remainder by performing 32%11, and verify the equation. Try the same with the division of -32 by 11, as well as -32 by -11, and 32 by -11.
3. Figure out the order (i.e., precedence) of arithmetic operations by trying different combinations like the following:

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

   Verify that the order of evaluation is:
   parenthesis
   **
   *,/,//,%
   +,-

4. Print out the value of π (use the math module). Then round it to the closest integer, and then to 2, 3, 30 digits after the decimal point.
5. Take quadratic.py from the book and make all type conversions explicit, i.e., when you see a number or a variable in there that must be of a certain type (say, float) but it is not (or you suspect it may not be), then use the type conversion functions (don't change the numbers, variables themselves; keep them in the form they are in the book).
6. Go through the math functions in Table 3.2 and figure out what they do.
7. Calculate the following: math.sqrt(3)**2. What did you expect as the result, and what did you get? Why?
8. Write a program that asks for a float from the user, and then applies to it sqrt() 10 consequtive times. Calculate the result in two different ways. *(Hint: sqrt() is actually an exponentiation.)*
9. Try to solve as many of the math problems as you can writing programs that use what you've learned in this chapter.