Computing with Numbers

Theresa Migler-VonDollen
CMPS 5P
The information that is stored and manipulated by computer programs is referred to as *data*.

There are two different kinds of numbers:

- (5, 4, 3, 6) are whole numbers - they don’t have a fractional part
- (.25, .10, .05, .01) are decimal fractions

Inside the computer, whole numbers and decimal fractions are represented quite differently.

We say that decimal fractions and whole numbers are two different data types.

The data type of an object determines what values it can have and what operations can be performed on it.
The data type of an object determines what values it can have and what operations can be performed on it.

Whole numbers are represented using the integer (int for short) data type.

These values can be positive or negative whole numbers.

Numbers that can have fractional parts are represented as floating point (or float) values.
How do we tell the two data types apart?

- A numeric literal without a decimal point produces an int value.
- A literal that has a decimal point is represented by a float (even if the fractional part is 0).

Python’s `type` function tell us which is which.
Why do we need two data types for numbers?

- Values that represent counts can’t be fractional, you can’t loop 4.5 times.
- Most mathematical algorithms are very efficient with integers.
- The float type stores only an approximation to the real number being represented.
- Since floats aren’t exact, use an int whenever possible.
- Operations on ints produce ints, operations on floats produce floats (except for division).
Numeric Data Types

- Integer division produces a whole number.
- That’s why $10 \div 3 = 3$
- Think of it as how many times 3 goes into 10 where $10 \div 3 = 3$ because 3 goes into 10 3 times with a remainder 1.
- $10 \% 3 = 1$ is the remainder of the integer division of 10 by 3.
- $a = (a/b)(b) + (a\%b)$
Besides the usual arithmetic functions, there are many other math functions available in the math library.

A library is a module with some useful definitions/functions.

Suppose we wanted to compute the roots of a quadratic equation:

\[ ax^2 + bx + c = 0 \]

The only part of this we don’t know how to do is find a square root... but it’s in the math library.
To use a library, we need to make sure this line is in our program:
import math

Importing a library makes whatever functions are defined within it available to the program.

To access the sqrt library routine, we need to access it as `math.sqrt(x)`.

Using this dot notation tells Python to use the sqrt function found in the math library module.

To calculate the root:
\[ \text{discRoot} = \text{math.sqrt}(b \times b - 4 \times a \times c) \]
Write a program that asks the user for the coefficients of a quadratic equation and returns the real roots of the equation.

- What if the roots are imaginary?
Accumulating Results

- Say you are waiting in a line with five other people. How many ways are there to arrange the six people?
- 720 – 720 is the factorial of 6 (abbreviated 6!).
- Factorial is defined as: \( n! = n(n - 1)(n - 2)\ldots(1) \)
- So, \( 6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720 \)
- How could we write a program to do this?
- Input: number to take factorial of, \( n \).
  Compute factorial of \( n \), \( \text{fact} \)
- Output: \( \text{fact} \)
How did we calculate $6!$?

Using repeated multiplications, and keeping track of the running product.

This algorithm is known as an accumulator, because we’re building up or accumulating the answer in a variable, known as the accumulator variable.

The general form of an accumulator algorithm looks like this:
1. Initialize the accumulator variable
2. Loop until final result is reached
3. Update the value of accumulator variable
We will need a loop:

```python
fact = 1
for factor in [6, 5, 4, 3, 2, 1]:
    fact = fact * factor
```

Why did we need to initialize fact to 1?

- Each time through the loop, the previous value of fact is used to calculate the next value of fact. By doing the initialization, you know fact will have a value the first time through.
- If you use fact without assigning it a value, what does Python do?
We could have also written our program as:

```python
fact = 1
for factor in [2, 3, 4, 5, 6]:
    fact = fact * factor
```

This is because multiplication is associative and commutative.
What if we want to generalize our factorial program?

We need to have a list like $[2, 3, 4, \ldots, n]$ or $[n, n-1, \ldots, 3, 2, 1]$. The `range(n)` function can do this.

`range` has optional parameters:
- `range(start, n)`
- `range(start, n, step)`
Write a program that computes the factorial of any number.

- What is 100!?
- That’s huge, Python 3 can handle it, but previous versions and other languages cannot.
While there are an infinite number of integers, there is a finite range of ints that can be represented.

This range depends on the number of bits a particular CPU uses to represent an integer value.

Typical PCs use 32 bits.

That means there are 2^32 possible values, centered at 0.

This range then is -2^31 to 2^31-1. We need to subtract one from the top end to account for 0.

But our 100! is much larger than this. How does it work?
Does switching to float data types get us around the limitations of ints?

If we initialize the accumulator to 1.0, what do we get?

We no longer get an exact answer!

Very large and very small numbers are expressed in scientific or exponential notation.

\[ 1.307674368e + 012 = 1.307674368 \times 10^{12} \]

Here the decimal needs to be moved right 12 decimal places to get the original number, but there are only 9 digits, so 3 digits of precision have been lost.
Handling Large Numbers

- Floats are approximations.
- Floats allow us to represent a larger range of values, but with lower precision.
- Python has a solution, expanding ints.
- Python ints are not a fixed size and expand to handle whatever value it holds.
- Newer versions of Python automatically convert your ints to expanded form when they grow so large as to overflow.
- We get indefinitely large values (e.g., 100!) at the cost of speed and memory.
We know that combining an int with an int produces an int, and combining a float with a float produces a float.

What happens when you mix an int and float in an expression?

\[ x = 5.0 + 2 \]

What do you think should happen?

For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer addition, or convert 2 to 2.0 and do a floating point addition.
Type Conversions

- Converting a float to an int will lose information.
- ints can be converted to floats by adding ".0"
- In mixed-typed expressions Python will convert ints to floats.
- Sometimes we want to control the type conversion. This is called explicit typing.