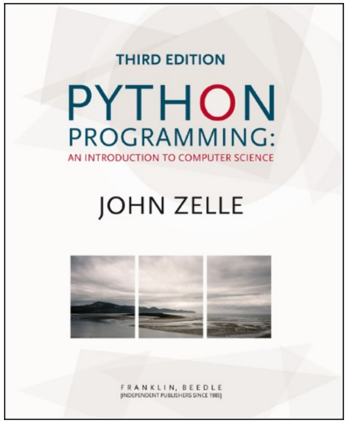
Python Programming: An Introduction to Computer Science



Chapter 3
Computing with Numbers

Objectives

- To understand the concept of data types.
- To be familiar with the basic numeric data types in Python.
- To understand the fundamental principles of how numbers are represented on a computer.

Objectives (cont.)

- To be able to use the Python math library.
- To understand the accumulator program pattern.
- To be able to read and write programs that process numerical data.

Numeric Data Types

- 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

Numeric Data Types

- 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.

Numeric Data Types

- Whole numbers are represented using the *integer* (*int* for short) data type.
- These values can be positive or negative whole numbers.

Numeric Data Types

- Numbers that can have fractional parts are represented as *floating point* (or *float*) values.
- How can we tell which is which?
  - 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)

Numeric Data Types

- Python has a special function to tell us the data type of any value.

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
>>>
```

Numeric Data Types

- Why do we need two number types?
  - Values that represent counts can’t be fractional (you can’t have 3 ½ quarters)
  - 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!

Numeric Data Types

- Operations on ints produce ints, operations on floats produce floats (except for /).

```
>>> 3.0+4.0
7.0
>>> 3+4
7
>>> 3.0*4.0
12.0
>>> 3*4
12
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
```

Numeric Data Types

- Integer division produces a whole number.
- That's why 10//3 = 3!
- Think of it as 'gozinta', where 10//3 = 3 since 3 gozinta (goes into) 10 3 times (with a remainder of 1)
- 10%3 = 1 is the remainder of the integer division of 10 by 3.
- a = (a//b)(b) + (a%b)

Type Conversions & Rounding

- 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?

Type Conversions & Rounding

- For Python to evaluate this expression, it must either convert 5.0 to 5 and do an integer multiplication, or convert 2 to 2.0 and do a floating point multiplication.
- Converting a float to an int will lose information
- Ints can be converted to floats by adding ".0"

Type Conversion & Rounding

- In *mixed-typed expressions* Python will convert ints to floats.
- Sometimes we want to control the type conversion. This is called *explicit typing*.
- Converting to an `int` simply discards the fractional part of a `float` – the value is truncated, not rounded.

Type Conversion & Rounding

- To round off numbers, use the built-in `round` function which rounds to the nearest whole value.
- If you want to round a float into another float value, you can supply a second parameter that specifies the number of digits after the decimal point.

Type Conversions & Rounding

```
>>> float(22//5)
4.0
>>> int(4.5)
4
>>> int(3.9)
3
>>> round(3.9)
4
>>> round(3)
3
>>> round(3.1415926, 2)
3.14
```

Type Conversions & Rounding

```
>>> int("32")
32
>>> float("32")
32.0
```

- This is useful as a secure alternative to the use of `eval` for getting numeric data from the user.

Type Conversions & Rounding

- Using `int` instead of `eval` ensures the user can only enter valid whole numbers – illegal (non-int) inputs will cause the program to crash with an error message.
- One downside – this method does not accommodate simultaneous input.

Type Conversions & Rounding

```
# change.py
#   A program to calculate the value of some change in dollars

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = quarters * .25 + dimes * .10 + nickels * .05 + pennies * .01
    print()
    print("The total value of your change is", total)
```

Using the Math Library

- Besides (+, -, *, /, //, **, %, abs), we have lots of other math functions available in a *math library*.
- A *library* is a module with some useful definitions/functions.

Using the Math Library

- Let's write a program to compute the roots of a quadratic equation!

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- The only part of this we don't know how to do is find a square root… but it's in the math library!

Using 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.

Using the Math Library

- 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, you can do discRoot = math.sqrt(b*b – 4*a*c)

Using the Math Library

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a, b, c = eval(input("Please enter the coefficients (a, b, c): "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )
```

Using the Math Library

```
This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 3, 4, -1

The solutions are: 0.215250437022 -1.54858377035
```

- What do you suppose this means?

```
This program finds the real solutions to a quadratic

Please enter the coefficients (a, b, c): 1, 2, 3

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in -toplevel-
    main()
  File "C:\Documents and Settings\Terry\My Documents\Teaching\W04\CS 120\Textbook\code\chapter3\quadratic.py", line 14, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
>>>
```

Using the Math Library

- If a = 1, b = 2, c = 3, then we are trying to take the square root of a negative number!
- Using the sqrt function is more efficient than using **. How could you use ** to calculate a square root?

Using the Math Library

Python	Mathematics	English
pi	π	An approximation of pi
e	e	An approximation of e
sqrt(x)	$\sqrt{x}$	The square root of x
sin(x)	$\sin x$	The sine of x
cos(x)	$\cos x$	The cosine of x
tan(x)	$\tan x$	The tangent of x
asin(x)	$\arcsin x$	The inverse of sine x
acos(x)	$\arccos x$	The inverse of cosine x
atan(x)	$\arctan x$	The inverse of tangent x

Using the Math Library

Python	Mathematics	English
log(x)	$\ln x$	The natural (base *e*) logarithm of *x*
log10(x)	$\log_{10} x$	The common (base 10) logarithm of *x*
exp(x)	e^x	The exponential of x
ceil(x)	$\lceil x \rceil$	The smallest whole number >= *x*
floor(x)	$\lfloor x \rfloor$	The largest whole number <= *x*

Accumulating Results: Factorial

- 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*5*4*3*2*1 = 720

Accumulating Results: Factorial

- How we could we write a program to do this?
- `Input number to take factorial of, n`
  `Compute factorial of n, fact`
  `Output fact`

Accumulating Results: Factorial

- How did we calculate 6!?
- 6*5 = 30
- Take that 30, and 30 * 4 = 120
- Take that 120, and 120 * 3 = 360
- Take that 360, and 360 * 2 = 720
- Take that 720, and 720 * 1 = 720

Accumulating Results: Factorial

- What's really going on?
- We're doing repeated multiplications, and we're 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*.

Accumulating Results: Factorial

- The general form of an accumulator algorithm looks like this:

```
Initialize the accumulator variable
Loop until final result is reached
  update the value of accumulator
  variable
```

Accumulating Results: Factorial

- It looks like we'll need a loop!

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

- Let's trace through it to verify that this works!

Accumulating Results: Factorial

- Why did we need to initialize fact to 1? There are a couple reasons…
  - 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?

Accumulating Results: Factorial

- Since multiplication is associative and commutative, we can rewrite our program as:

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

- Great! But what if we want to find the factorial of some other number??

Accumulating Results: Factorial

- What does *range(n)* return?
  0, 1, 2, 3, …, n-1
- range has another optional parameter!
  range(start, n) returns
  start, start + 1, …, n-1
- But wait! There's more!
  range(start, n, step)
  start, start+step, …, n-1
- `list(<sequence>)` to make a list

Accumulating Results: Factorial

- Let’s try some examples!

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
```

Accumulating Results: Factorial

- Using this souped-up *range* statement, we can do the range for our loop a couple different ways.
  - We can count up from 2 to n:
    range(2, n+1)
    (Why did we have to use n+1?)
  - We can count down from n to 2:
    range(n, 1, -1)

Accumulating Results: Factorial

- Our completed factorial program:

```
# factorial.py
#    Program to compute the factorial of a number
#    Illustrates for loop with an accumulator

def main():
    n = eval(input("Please enter a whole number: "))
    fact = 1
    for factor in range(n,1,-1):
        fact = fact * factor
    print("The factorial of", n, "is", fact)

main()
```

The Limits of Int

- What is 100!?

```
>>> main()
Please enter a whole number: 100
The factorial of 100 is
   9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000
```

- Wow! That's a pretty big number!

The Limits of Int

- Newer versions of Python can handle it, but…

```
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import fact
>>> fact.main()
Please enter a whole number: 13
13
12
11
10
9
8
7
6
5
4
Traceback (innermost last):
  File "<pyshell#1>", line 1, in ?
    fact.main()
  File "C:\PROGRA~1\PYTHON~1.2\fact.py", line 5, in main
    fact=fact*factor
OverflowError: integer multiplication
```

The Limits of Int

- What's going on?
  - 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.

The Limits of Int

- Typical PCs use 32 bits or 64.
- 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?

Handling Large Numbers

- Does switching to *float* data types get us around the limitations of *int*s?
- If we initialize the accumulator to 1.0, we get

```
>>> main()
Please enter a whole number: 30
The factorial of 30 is 2.652528598121911e+32
```

- We no longer get an exact answer!

Handling Large Numbers: Long Int

- Very large and very small numbers are expressed in *scientific* or *exponential notation*.
- 2.6525285981219111e+32 means 2.6525285981219111 * 10^{32}
- Here the decimal needs to be moved right 32 decimal places to get the original number, but there are only 16 digits, so 16 digits of precision have been lost.

Handling Large Numbers

- Floats are approximations
- Floats allow us to represent a larger range of values, but with fixed precision.
- Python has a solution, expanding ints!
- Python ints are not a fixed size and expand to handle whatever value it holds.

Handling Large Numbers

- 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