Vahé Karamian

Python Programming

CS-110

CHAPTER 3

Computing with Numbers

OBJECTIVES

- To understand the concepts 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.
- 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 generally referred to as data.
- There are two different kinds of numbers!
  - Whole Numbers: (1,2,3,…) they do not have fractional part
  - Decimal Numbers: (0.25, 0.5, 0.75,…)
- Inside the computer, whole numbers and decimal numbers are represented quite differently.
- We say that decimal fractions and whole numbers are two different data types.

NUMERIC DATA TYPES

- 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.
- Decimal numbers are represented using the *floating point* (or *float*) data type.

NUMERIC DATA TYPES

- 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 fraction part is 0)
- Python has a special function to tell us the data type of any value.

```
>>>type(3)
<type 'int'>
```

NUMERIC DATA TYPES

- Why do we need two number types?
  - Values that represent counts can't be fractional.
  - Most mathematical algorithms are very efficient with integers.
  - The float type stores only an approximation to the real number being represented!
- Operations on integers produce integers, operations on floats produce floats.

NUMERIC DATA TYPES

OPERATOR	OPERATION
+	Addition
-	Subtraction
*	Multiplication
/	Float Division
**	Exponentiation
abs()	Absolute Value
//	Integer Division
%	Remainder

NUMERIC DATA TYPES

- Integer division always produces an integer, discarding any fractional result.
- That's why 10 / 3 = 3
- And 10.0 / 3.0 = 3.333333335
- Now you can see why we had to use 9.0/5.0 rather than 9/5 in our Celsius to Fahrenheit conversion program!

USING THE MATH LIBRARY

- Besides the basic operations we just discussed, Python provides many other useful mathematical functions in a special math library.
- A library is a module with some useful definitions/functions.
- Let’s write a program to compute the roots of a quadratic equation!

USING THE MATH LIBRARY

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

- The only part of this we don't know how to do is find a square root … but thanks to Python, someone has already programmed that for us in a library!
- To use a library, we need to make sure this line is in our program:
  - *import math*
- Importing a library makes whatever functions that are defined within it available for us to use in our program!

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)	$\sin^{-1} x$	The inverse of since x
Acos(x)	$\cos^{-1} x$	The inverse of cosine x
Atan(x)	$\tan^{-1} x$	The inverse of tangent x
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

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

```
# Makes the math library available
import math

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)

    main()
```

USING THE MATH LIBRARY

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

Please enter the coefficients (a, b, c): 3,4,-2

The solutions are:  0.387425886723 -1.72075922006
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#0>", line 1, in <module>
    main()
  File "C:/Python31/quadratic.py", line 22, in main
    main()
  File "C:/Python31/quadratic.py", line 15, in main
    discRoot = math.sqrt(b*b-4*a*c)
ValueError: math domain error
```

What do you suppose this mean?

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?

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 is the factorial of 6
- Factorial is defined as:
  - n! = n(n-1)(n-2)..1
- So 6! = 6*5*4*3*2*1 = 720

ACCUMULATING RESULTS: FACTORIAL

- How could we write a program to perform a factorial?
  - Input number to take factorial of n
  - Compute factorial of n
  - Output factorial
- How would we calculate 6! ?

ACCUMULATING RESULTS: FACTORIAL

- Let's look at it closely
  - 6*5 = 30
  - 30*4 = 120
  - 120*3 = 360
  - 360*2 = 720
  - 720*1 = 720
- What is really going on here?

ACCUMULATING RESULTS: FACTORIAL

- We are doing repeated multiplications, and we're keeping track of the running product.
- This algorithm is known as an *accumulator*, because we are 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
  - Update the value of accumulator variable
- Loops can come handy in accumulators!
- How can we write a factorial program using 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?
  - 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 will Python do?

ACCUMULATING RESULTS: FACTORIAL

- Since multiplication is associate 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
- The *range()* function 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

ACCUMULATING RESULTS: FACTORIAL

- Let's try some examples!

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
for i in range(5,10):
    print(i)
```

```
5
6
7
8
9
```

```
for i in range(5,10,2):
    print(i)
```

```
5
7
9
```

ACCUMULATING RESULTS: FACTORIAL

- Using our new knowledge of the additional options for the range() function. We can formulate our loop a couple of different ways!
  - We can count up from 2 to n:
    - *range(2, n+1)* : Why we use n+1?
  - We can count down from n to 1:
    - *range(n, 1, -1)*

ACCUMULATING RESULTS: FACTORIAL

- Our completed factorial program:

```
# factorial.py
# progam 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()
```

LIMITATIONS OF COMPUTER ARITHMETICS

- Back in Chapter 1, we learned that the computer's CPU (Central Processing Unit) can perform very basic operations such as adding or multiplying two numbers.
- Better put, these operations are performed on the internal representation of numbers/data which are represented by 0s and 1s!
- Inside the computer the, data is stored in a fixed size binary representation.

LIMITATIONS OF COMPUTER ARITHMETICS

- Computer memory is composed of electrical “switches”
- These switches can be in one of two possible states, either on or off, in other words 0 or 1.
- Each switch represents a binary digit or bit of information.
- One bit can encode two possibilities, again either 0 or 1!
- A sequence of bits can be used to represent more possibilities.

LIMITATIONS OF COMPUTER ARITHMETICS

Bit 2	Bit 1
0	0
0	1
1	0
1	1

Bit 3	Bit 2	Bit 1
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

LIMITATIONS OF COMPUTER ARITHMETICS

- The limits of integer representation in computers.
  - While there are an infinite number of integers, there is a finite range of integers that can be represented in a computer.
  - This range depends on the number of bits a particular CPU uses to represent an integer value.
  - Typical PCs use 32-bits.

LIMITATIONS OF COMPUTER ARITHMETICS

- That means there are 2^{32} possible values, centered at 0.
- This range then is -2^{31} to 2^{31}. We need to subtract one from the top end to account for 0.
- We can test this using the older version of Python!

LIMITATIONS OF COMPUTER ARITHMETICS

- It blows up between 2^{30} and 2^{31} as we expected. Can we calculate 2^{31}-1?

```
>>> 2 ** 31 – 1
Traceback (innermost last):
  error message, line #, in ?
  2 ** 31 -1
OverflowError: integer pow()
```

- What happened? It tried to evaluate 2^{31} first!

LIMITATIONS OF COMPUTER ARITHMETICS

- We need to be more cleaver!
  - $2^{31} = 2^{30} + 2^{30}$
  - $2^{31} = 2^{30}-1 + 2^{30}$
  - We are subtracting one from each side
- What have we learned?
  - The largest *int* value we can represent is 2147483647

HANDLING LARGE NUMBERS

- Very large and very small numbers are expressed in scientific or exponential notation.
  - 1.307674368e+012 = $1.307674368 * 10^{12}$
- 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 large range of values, but with lower precision.
- Python has a solution.
- A Python int is not a fixed size, but expands to accommodate whatever value it holds.
- The only limit is the amount of memory the computer has available to it!

HANDLING LARGE NUMBERS

- When the number is small, Python uses the computer's underlying int representation and operations.
- When the number is large, Python automatically converts to a representation using more bits.
- In order to perform operations on larger numbers, Python has to break down the operations into smaller units that the computer hardware is able to handle.

TYPE CONVERSIONS

- We know that combining an int with an int produces and 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 CONVERSION

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

TYPE CONVERSION

- In mixed-typed expression Python will convert ints to floats.
- Sometimes we want to control the type conversion. This is called explicit typing.
- average = sum / n
- If the numbers to be averaged are 4, 5, 6, 7 then sum is 22 and n is 4, so sum / n is 5, not 5.5!

TYPE CONVERSION

- To fix this problem, tell Python to change one of the values to floating point:
  - average = float(sum) / n
- We only need to convert the numerator because now Python will automatically convert the denominator.
- Why doesn't this work?
  - average = float(sum/n)
  - Sum = 22, n = 5, sum/n = 4, float(sum/n) = 4.0!
- Python also provides *int()*, and *long()* functions to convert numbers into ints and longs.

CHAPTER SUMMARY

- The way a computer represents a particular kind of information is called a data type. The data type of an object determines what values it can have and what operations it supports.
- Python has several different data types for representing numeric values, including int and float.
- Whole numbers are generally represented using the int data type and fractional values are represented by using floats.

CHAPTER SUMMARY

- All of Pythons numeric data types support standard, built-in mathematical operations:
  - +, -, *, /, //, %, **
- Additional mathematical functions are defined in the math library. To use these functions, a program must first import the math library.
- Both ints and floats are represented on the underlying computer using a fixed-length sequence of bits. This imposes certain limits on these representations.
- Hardware ints must be in the range -2^{31} … $(2^{31}-1)$ on a 32-bit machine.
- Floats have a finite amount of precision and cannot represent most numbers exactly.

CHAPTER SUMMARY

- Python automatically converts numbers from one data type to another in certain situations. For example, in a mixed-type expression involving ints and floats, Python first converts the ints into floats and then uses float arithmetic.
- Programs may also explicitly convert one data type into another using the functions *float()*, *int()*, and *round()*

CHAPTER 3 HOMEWORK

Review Questions

True/False (ALL)

Multiple Choice (ALL)

Discussion Questions:

1, 2, 3, 4

Programming Exercises:

1, 3, 5, 6, 7, 9