Topic: Numerical Computation

Good news: computers are really fast at arithmetic!

Bad news: Computer arithmetic is not exact!

Good enough much of the time, but not always.
Must understand shortcomings and how to cope with them.

Computer Storage

computer memory & devices (disks, etc.):
    lots of binary digits (*bits*) – each 0 or 1
integers, floats, Strings, lists, etc. all made up of bits
encoding schemes translate data to & from sequences of bits

bits grouped into sets of 8
8 bits = 1 *byte*

Most numeric types: set number of bits used.
If that's not enough bits: problems!

How Integers are Represented

integer = sequence of bits, interpreted as a binary number (base 2)

Example:

$$21_{10} = 10101_2$$

$2\text{x}10^1$ $1\text{x}10^0$ $1\text{x}2^4$ $0\text{x}2^3$ $1\text{x}2^2$ $0\text{x}2^1$ $1\text{x}2^0$

Python ints are *usually* 32 bits long
one bit used for sign
so range is -2^{31} to $2^{31}-1$

translation is exact – no accuracy lost going back & forth

Integer Overflow or Underflow

Range of values that will fit in a 32-bit **`int`**: -2^{31} to $2^{31}-1$

That's -2147483648 to 2147483647

Trying to store a value that's too big: *overflow*

a value that's too small: *underflow*

Other languages (& older versions of Python):

Overflow/underflow leads to errors and/or bad results

Python 2.1:

```
>>> 9999**8
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OverflowError: integer exponentiation
```

Python long type

Python has another integer type: **`long`**
Represents integer values with *no restrictions* on size.
(representation: list of integers or digits?)
No overflow/underflow; always stores exact values.

Since Python 2.2, if an int overflows Python converts the result to a **`long`**:

```
>>> 9999**8
99920027994400699944002799920001L
```

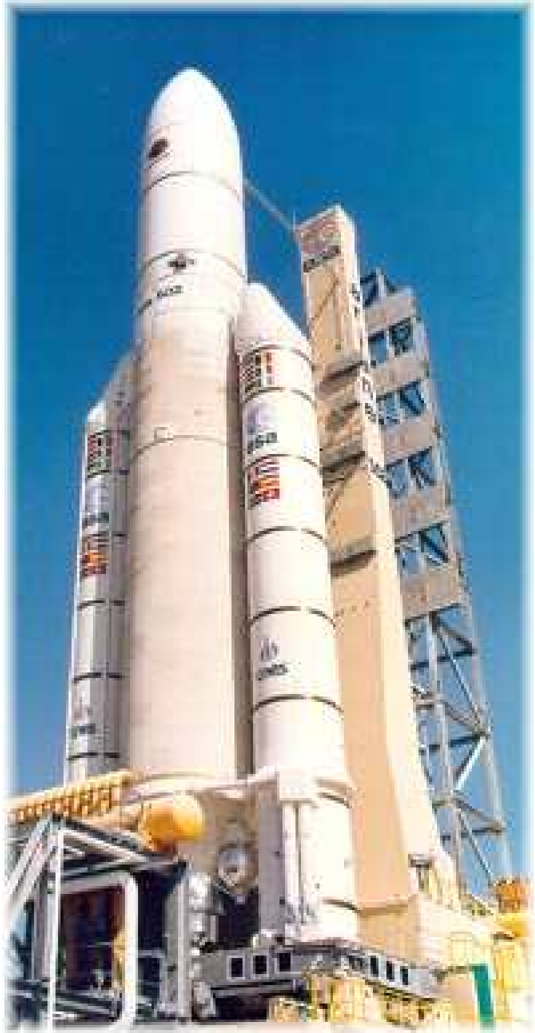
Promised in later versions of Python: No distinction between **`int`** and **`long`**.

Conclusion: Integers in Python

Arithmetic with integers in Python will always be exact.
The result may be an **`int`** or a **`long`**.

But don't get in the habit of assuming this for every language!

The Ariane 5 Explosion



June 4, 1996
http://www.around.com/ariane.html
http://youtu.be/gp_D8r-2hwk

Floating Point (Decimal)

Recall Scientific Notation (base 10):

$$120{,}000 = .\underbrace{12}_{\text{mantissa}} \times 10^{\overbrace{6}^{\text{exponent}}}$$

Conventions:

- no digits before the decimal point
- mantissa never starts with zero

If a number obeys these conventions it is *normalized*

Floating Point (Binary)

Computer uses similar binary notation to represent floating point numbers

$$21_{10} = 10101_2 = .10101_2 \text{ x } 2^{101}$$

mantissa: 10101
exponent: 5 (101 in binary)
binary convention: first digit always 1

To store floating point number on a computer:

- mantissa
- exponent
- one extra bit for sign (+/-)

The Standard Floating-Point Representation

Most Python implementations use 8 bytes to represent a floating-point number:

1 bit for sign
52 bits for mantissa = approx. 15 decimal digits
11 bits for exponent
total: 64 bits = 8 bytes

Warning: Some Python implementations may use a slightly different scheme.
Floating-point results will vary.
These slides assume the above scheme.

Example

How does a computer represent 14.375?

$$14.375 = 8+4+2+\frac{1}{4}+\frac{1}{8}$$

$$= \mathbf{1110.011_2}$$

$$= \mathbf{.1110011 \times 2^4}$$

So 14.375 represented as:

mantissa = 1110011

exponent = 100

sign = positive

Roundoff Error

Any computer has only a finite number of bits for mantissa
Rest are truncated.

Example: hypothetical small computer, 6 bits for mantissa
want to represent $14.375_{10} = .1110011 \times 2^4$

Exact representation needs 7 bits for mantissa.
We only have 6, so we truncate to $.111001 \times 2^4 = 14.25_{10}$

Categorizing Errors

Previous Example: exact answer was 14.375
Computer got 14.25

Absolute Error: | exact value - computer value |
In this example, absolute error is .125

Relative Error: $\left| \dfrac{exact\ value - computer\ value}{exact\ value} \right|$

same as: $\left| \dfrac{absolute\ error}{exact\ value} \right|$

In this example, relative error is .125 / 14.375 = .0087 = .87%

Decimal -> Binary Problems

Some decimal fractions can't be represented exactly in binary

Example: 0.1 = 1/10
As a binary number: .000 1100 1100 1100 .....

With 52 bits for mantissa, you can get very close
but it's still not exact

```
>>> 1/10.0
0.10000000000000001
```

Seems close enough, but what if you are adding many numbers,
all with very small errors like this?

Errors can accumulate and become significant

Python Examples

```
>>> sum = 0.0
>>> for i in range(0,1000):
     sum += 0.1

>>> sum
99.9999999999998593

>>> for i in range(0,10000000):
     sum += 0.1

>>> sum
1111099.9998858953
```

Patriot Missile Failure

February 25, 1991
http://www.ima.umn.edu/~arnold/disasters/patriot.html
http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

Back to Decimal

To discuss how errors accumulate, we will use a hypothetical computer that stores numbers in decimal.
Makes arithmetic much easier and principles are the same.

hypothetical computer uses 3 digits for mantissa
rounds off extra digits

to represent 12.37:
normalize so no digits before decimal: $.1237 \times 10^2$
round off so only 3 digits: $.124 \times 10^2$

Adding Numbers of Similar Size

Example: 5.63 + 6.81
Exact answer is 12.44

Representation on our hypothetical computer:

```
 .563 x 10¹
+.681 x 10¹
------------
1.244 x 10¹
```

Normalize: **.1244 x 10^2**

We only have 3 digits, so round: **.124 x 10^2 = 12.4**

Absolute Error = .04 Relative Error: .04/12.44 = 0.32%

Adding Numbers of Different Sizes

Example: **563 + 4.32** Exact answer is **567.32**

Representation on our hypothetical computer:

```
 .563 x 10³
+.432 x 10¹
------------
```

Before we can add, we must line up decimal points:

both numbers must have same exponent

Un-normalize .432 x 10 so that its exponent is 3: **.00432 x 10³**

Our computer only allows 3 digits, so round: **.004 x 10³**

```
 .563 x 10³
+.004 x 10³
------------
 .567 x 10³ = 567
```

Error is **.32**, or **.06%**

Another Example

When you add a small number to a much larger one, you lose some of the last digits of the smaller number

If the numbers are different enough, you may lose the small one altogether.

Example: **5630 + 4.32**   Exact answer is **5634.32**

$$\begin{array}{r} .563 \times 10^4 \\ +.432 \times 10^1 \\ \hline \end{array} \longrightarrow \begin{array}{r} .563 \times 10^4 \\ +.\color{red}{000} \times 10^4 \\ \hline .563 \times 10^4 \end{array}$$

Example in Python

```
>>> big = 1E30
>>> small = 1E10
>>> big+small
1e+030
>>> (big+small) == big
True
```

Order Matters

usual math fact: a + (b + c) = (a + b) + c

Not always true for computer arithmetic!

Example: 1000 + 1 + 2 + 3 + 4. Exact answer: 1010

On our hypothetical 3-digit decimal computer:

Sum left to right: 1000 + 1 = 1000
1000 + 2 = 1000
1000 + 3 = 1000
1000 + 4 = 1000 Error: 10

Sum right to left: 4 + 3 + 2 + 1 = 10
10 + 1000 = 1010. Error: 0

In general, works best to add from smallest to largest.

Infinite Series

How do computers calculate functions such as sine, cosine, exp?
One way: infinite series. Derived using calculus.

Example: $$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - .....$$

Meaning: If you add up this sum "forever", you get the exact value of sin(x).

Alternately: You can get as close to the exact value of sin(x) as you want by adding up enough terms of this series.

Both of the above are true with "real" arithmetic – not computer arithmetic.

Question: in what order should you add up the terms in this series?

Old Final Exam Question

two ways to calculate ln(2):

$$\ln(2) = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots$$

$$\ln(2) = 2\left(\frac{1}{3} + \frac{1}{3}\left(\frac{1}{3}\right)^3 + \frac{1}{5}\left(\frac{1}{3}\right)^5 + \frac{1}{7}\left(\frac{1}{3}\right)^7 + \ldots\right)$$

Both are mathematically correct.
Which would work best on a computer?

Overflow

Computer representations also have limits on size of exponents.

Let's say hypothetical computer allows exponents from –5 to 5.

Add $6 \times 10^4 + 7 \times 10^4$:

```
 .600 x 10⁵
+.700 x 10⁵
------------
1.300 x 10⁵
```

Answer normalizes to **$.130 \times 10^6$** – exponent is too big.
This is *overflow*.

Different languages handle it different ways.
In Python: **`OverflowError`**

Python Example

```
try:
    power = 100
    while True:
        x = 10.0**power
        print "power:", power, "x:", x
        power += 1
except OverflowError:
    print "overflow with power =", power
```

Output:

```
power: 100 x: 1e+100
power: 101 x: 1e+101
power: 102 x: 1e+102
   ........
power: 306 x: 1e+306
power: 307 x: 1e+307
power: 308 x: 1e+308
overflow with power = 309
```

Underflow

Similar to overflow: with hypothetical decimal computer,
exponent can't be less than –5.

$(.123 \times 10^{-5}) / 10 = .123 \times 10^{-6}$

Exponent is too small: result is zero.

In Python: no exception for underflow.
Result is zero.

Python Example

```
n = 1.0
while n > 0.0:
    n = n / 10.0
    print n
if n == 0.0:
    print "n is exactly zero"
else:
    print "n is close to zero"
```

Last lines of output:

```
9.98012604599e-322
9.88131291682e-323
9.88131291682e-324
0.0
n is exactly zero
```

Arithmetic and Portability

Many languages (Python, C) let writer of interpreter / compiler choose scheme for representing numbers.

Python programs may yield different numeric results on different computers

Advantage: speed -- can pick scheme that matches computer hardware.

Other languages (Java, Turing) specify exact encoding scheme for numeric types and rules for arithmetic

A Java program should yield the same numeric results on every computer

Advantage: portability