Numerical Programming in Python

Part I: The Basic Facilities

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

February 2006

Overview of Course

Basic facilities – i.e. using Python
Integers, floating–point, complex etc.

Arithmetic details and exception handling
What we need to know, but don't want to

Applications of Python for numerics
Some important ways of using it

Practicals etc.

Many examples – to see what happens
Code is in directory Demos

Please run them and check for surprises
Ask questions if you are puzzled

There are a few, simple, real practicals
Assume that you already program in Python

Beyond the Course

Email escience–support@ucs for advice

http://www–uxsup.csx.cam.ac.uk/courses/...
.../NumericalPython

http://www.scipy.org/

Let's Start Simply

Python makes an excellent desk calculator
Non–trivial work is a pain in most (e.g. dc)
Excel is better, but still can be painful

Not as powerful as Matlab, in that respect
But is much more powerful in others

Very useful for one–off calculations
No ‘‘cliff’’ between them and complex program

Trivial Practical

What diameter circle has area of 10 cm^2?

vol. = πr^2 $\Rightarrow$ diam. = $2\sqrt{10/\pi}$

python

```
from math import pi, sqrt
print 2.0*sqrt(10.0/pi)
```

Try that and check result is about 3.568

Python Output

```
3.56824823231
```

Python's Facilities

Will now go through all of built–in numerics
At each stage, will try out facilities

- What they DO, not just how to use

Python is very standard computer language
Most things apply to other ones, too

- Key factor is how to map mathematics

Simple use is not hard, if approached right

Python's Integers

No limit on size, except memory
Definite errors (e.g. 123/0) raise exceptions
Exceptions can be trapped – see later

Very big integers (e.g. > 10^{1000}) can be slow
Multiply, divide, remainder, conversion, etc.

- Most things just work as you would expect

Integer Operations

'+', '−', '∗', '/' (used for $\div$) ops, as usual

'/' $\Rightarrow -\infty$ – can also be written '//'

x%y is remainder, same sign as 'y' – note!

Built–in functions:

abs – absolute (positive) value

Type conversion functions – int $\equiv$ long

divmod(x, y) $\Rightarrow$ (x/y, x%y)

pow(x, y) (or x∗∗y) $\Rightarrow$ x^y

Examples

```
x = divmod(+123, -45)
print +123/-45, +123%-45, x
print x[0]*-45+x[1]
```

Then try other combinations of signs

```
print 100+23, abs(-123), abs(+123)
print pow(2, 10), pow(-5, 3), pow(5, 0)
```

Will return to exception handling later

Python Output

```
-3 -12 (-3, -12)
123

-3 12 (-3, 12)
-123
2 -33 (2, -33)
-123
2 33 (2, 33)
123

123 123 123
```

Formatted Output

Formatted output based on C
Simple case: %d or %<width>d
If width too small, uses minimum necessary

```
print "%d %d " % (123, 1234567890)
print "%7d %7d" % (123, 1234567890)
```

Many more options, but can be ignored

Python Output

123 1234567890
    123 1234567890

Logical (Bitwise) Operations

Dubiously numeric, so will gloss over
See documentation for more details

Treats number as binary, twos complement
Can input/output as hex. or octal
Usual selection of logical operations

Shifts scale by a power of two (useful)
a<<b $\equiv$ a*2^b,   a>>b $\equiv$ a/2^b

Python's Floating-Point (1)

The type is called float and is numeric

- Does most things you learnt at A–level

Will return to numerical properties later

±<digits>.<digits>[<exponent>]
<exponent> is [e|E]±<digits>

Anything non–critical can be omitted
1.23, –0.00123, 1.23e5, +1e–5, 123.4E+5 etc.
Avoid unclear .23, 123., but will work

Floating-Point Operations

Includes everything you can do with integers
'/' is floating–point division

'//', '%', divmod use integer quotient

- But all results remain as float

Also fmod, modf from math (see later)

Mixing integers and reals works as expected

- Result is almost always floating–point

pow(<int>, –<int>) $\Rightarrow$ float

Examples

```
print +12.3/–3.4, +12.3//–3.4, +12.3%–3.4,  \
      divmod(+12.3,–3.4)
```

Other combinations of signs are similar

```
print abs(–123.4), pow(1.2345, 10)
print 123.0/34, 123/34.0, 5*2.34567+98
x = –3
print pow(5, –3), pow(5, x), pow(5, –x)
```

Will return to exception handling later

Python Output

–3.61764705882 –4.0 –1.3
(–4.0, –1.2999...99989)

–3.61764705882 –4.0 1.3 (–4.0, 1.29...989)
3.61764705882 3.0 –2.1 (3.0, –2.100...001)
3.61764705882 3.0 2.1 (3.0, 2.1000...0001)

123.4 8.22074056463
3.61764705882 3.61764705882 109.72835
0.008 0.008 125

Floating-Point Formatting (1)

Very like integer formatting, for same reason
%<width>.<prec>f is fixed–point form
%<width>.<prec>e is scientific form

Lots of variations, but can ignore most

- Provide a precision – default is poor

A precision of zero prints in integer form

- Can trust only 15 sig. figs
- Need 18 sig. figs to guarantee reinput

Floating-Point Formatting (2)

Try:

```
x = 100.0/7.0
print "%.3f %.5e" % (x, x)
print "%10.5f %20.3e" % (x, x)
print "%.0f %.0e" % (x, x)

print "%.30f %.30e" % (9.1, 9.1)
print "%.30f" % 1.0e–15
```

See where the numbers start to go wrong

Python Output

```
14.286 1.42857e+01
 14.28571            1.429e+01
14 1e+01

9.099999999999999644728632119950
    9.099999999999999644728632119950e+00
0.000000000000001000000000000000
```

Floating-Point Formatting (3)

Results almost always round correctly:

```
x = (1.234567890125, 1.23456789012501)
print "%.20f %.20f " % x
print x[0], x[1]
print "%.11f %.11f " % x
```

Default is a bit odd, but still rounds:

```
print x[0], x[1], x
```

Python Output

```
1.2345678901249999044
    1.2345678901250098924
1.23456789012 1.23456789013

1.23456789012 1.23456789013
 (1.2345678901249999, 1.2345678901250099)
```

Integers In Reals

Up to > $\pm\ 10^{15}$ in float are exact
Conversion to int or long uses C's rule
This almost always truncates towards zero

Alternatively, floor, ceil, from math
Towards $-\infty$ and $+\infty$, as float

Except for NaNs (see later), few problems
'Reasonable' behaviour OR raises exception

Examples

Try:

```
x = 1.0
for i in xrange(1,30) :
    x = x*5.0
    print "%2d: %.0f %.0f %.0f %.0f" %  \
        (i, x, pow(5,i), x-1, x+1)
```

Now look at line 23 – notice anything?
There are TWO things to notice

Output

1:	5	5	4	6
2:	25	25	24	26
3:	125	125	124	126
4:	625	625	624	626
	. . .			
21:	...125	...125	...124	...126
22:	...625	...625	...624	...626
23:	...124	...124	...124	...124
24:	...624	...624	...624	...624

The %d Descriptor

Watch out for %d with float data
It converts to an integer before formatting

- Use not recommended, as might change

```
x = 12345.6
y = −x
print "%.0f %.0f" % (x, y)
print "%d %d" % (x, y)
```

Python Output

12346 –12346
12345 –12345

Standard Modules

Module math includes functions, pi and e
sqrt, exp, log, log10 etc.
Normal and inverse trig. and hyperbolic
Plus those mentioned above and some others

Calls the C library directly – see later

- Watch out for exception handling!
- Use built–in pow, NOT from math

Module random includes reasonable generators

Examples

Try:

```
from math import sqrt, cos, log, atan, pi, e
print sqrt(10), log(10), cos(4)
print log(pow(e,3)), cos(pi/4)
print 4*atan(1.0), atan(1.0e6)

from random import random, gauss
for i in xrange(0,10) :
    print random(), gauss(100.0,20.0)
```

Python Output

```
3.16227766017 2.30258509299
    -0.653643620864
3.0 0.707106781187
3.14159265359 1.57079532679

0.774001216879 102.136112561
0.68237930206 105.101301637
0.28760594402 139.895961878
  . . .
```

Practical

Calculate 'e' by summing series

1 + 1/1 + 1/2 + 1/6 + 1/24 + . . . + 1/(n!) . . .

Use floating–point, add until no change

Print e, exp(1) from math and your result

They should all be the same!

Sample Code

```
from math import e, exp
total = 0.0
fact = 1.0
n = 1
while total+fact > total :
    total = total+fact
    fact = fact/n
    n = n+1

print e, exp(1), total
```

Decimal Floating-Point

Included in new IEEE 754R standard
Unclear when (and if!) hardware will have it
Python has it in the decimal module

NOT a panacea – or significantly worse
The exactness claims are propaganda
Try π, 1.0/3.0, 1.01^{25}, scientific code

Experiment with it if you are interested
Not yet recommended for real work

Complex Numbers (1)

Imaginary parts are <number>J (or 'j')
1.23+4.56j or –1.0j ≡ –1j are complex
complex(x,y) ≡ x+y∗1j even if 'y' is complex

- Most things just work as you would expect

Assuming that you use complex numbers!

- Convert to float for formatted I/O

Default I/O (e.g. print 1.23+4.56j) is fine

Complex Numbers (1)

All the built–ins that float has

- divmod, ‘//’ and ‘%’ are deprecated

Built–in real, imag attributes
Built–in conjugate method

Module cmath is analogue of math
It doesn’t have pow, but that is good

Complex Examples

```
from cmath import sqrt, cos, exp, pi, e
x = complex(12.3,3.4)
y = 5.67+8.9j
print x, y, x+y, x*y, x/y, cos(x)
print x*x, pow(x,2), sqrt(-1)
print exp(x), pow(e,x)

print x.real, x.imag, x.conjugate()
print pow(abs(x),2), x*x.conjugate()
```

Python Output

```
(12.3+3.4j) (5.67+8.9j) (17.97+12.3j)
    (39.481+128.748j)
    (0.898006356025–0.809921793409j)
    (14.4697704817+3.93935941325j)
(139.73+83.64j) (139.73+83.64j) 1j
(–212401.684765–56141.3550562j)
    (–212401.684765–56141.3550562j)

12.3 3.4 (12.3–3.4j)
162.85 (162.85+0j)
```

Where Are We?

The basics of all Python built–in numerics

- Many people can go on and write code

Provided that nothing goes wrong!

- But, in real life, things do go wrong

Will now describe the arithmetic model
Including basics of exceptions

- Need to understand this to avoid pitfalls

Get right answers, not just plausible ones