

# Maths

Week 3, day 4

# Numerical types in Python

- Integers: `12`, `int()`, limited to  $2^b$  or less
- Floats: `12.0`, `float()`
- Longs: `12L`, `long()`, limitless

# Numerical operations in Python

- Arithmetic operators: + - \* / % \*\*
- Inplace operators:
  - $x += 3 \Rightarrow x = x + 3$
  - += -= \*= /= etc.
- When in doubt about order (use brackets)

# Integer division

```
>>> 1 / 2  
0
```

```
>>> 1 / 2.0  
0.5
```

```
>>> 1. / 2  
0.5
```

# The imprecision of floats

```
# equality & floats don't mix
```

```
>>> x = 1.0000000000000000000000000001
```

```
>>> x  
1.0
```

```
>>> x = 0.1
```

```
>>> while (x != 0.2):  
    x += 0.1
```

```
>>> print x  
2.0000000001
```

# Standard numerical modules

- `math`: standard
- `cmath`: math for complex numbers
- `decimal`: for precise representation of floats
- `random`: generating random numbers

# math

- Constants: `pi`, `e`
- Trigonometric functions: `cos`, `sin`, `atan`,  
`degrees` and `radians` etc.
- Rounding: `ceil`, `floor`
- Logarithms: `log`, `log10`, `exp` ( $e^x$ )
- And `sqrt`, `pow`, `fabs` (absolute value) etc.

# decimal

- Like long except for floats
- Class Decimal allows arbitrary precision floats
- Express numbers as normal integers or strings or a tuple form **BUT NOT FLOATS**
- Also recognises 'Infinity', '-Infinity', 'NaN'

# decimal in use

```
from decimal import *

# construction
d1 = Decimal (32)
d2 = Decimal ("32.0")
d3 = Decimal ((1, (3, 2, 0, 0), -2))
d4 = Decimal ('Infinity')

>>> d1+d2
Decimal ("64")
>>> float (d2/d3)
1.0
```

# decimal

- Obey usual operators
- Many useful methods
- Immutable (doesn't matter)
- Calculations are controlled by a `Context` class, either in environment or passed to the `Decimal` constructor or `Decimal` methods ...

# decimal & contexts

```
>>> theCont = getcontext()
>>> theCont
Context(prec=28, rounding=ROUND_HALF_EVEN,
Emin=-999999999, Emax=999999999, capitals=1,
flags=[], traps=[InvalidOperation,
DivisionByZero, Overflow])
>>> theCont.prec = 4
>>> Decimal("123.456") + Decimal("123.456")
Decimal("246.9")

>>> big = Decimal("1.23456")
>>> big.quantize(Decimal("0.01"))
Decimal("1.23")
>>> big.quantize(Decimal("0.01"), ROUND_UP)
Decimal("1.24")
```

# random

- `import random`
- random number generators
- random choices
- random lists
- shuffling

# random: Generators

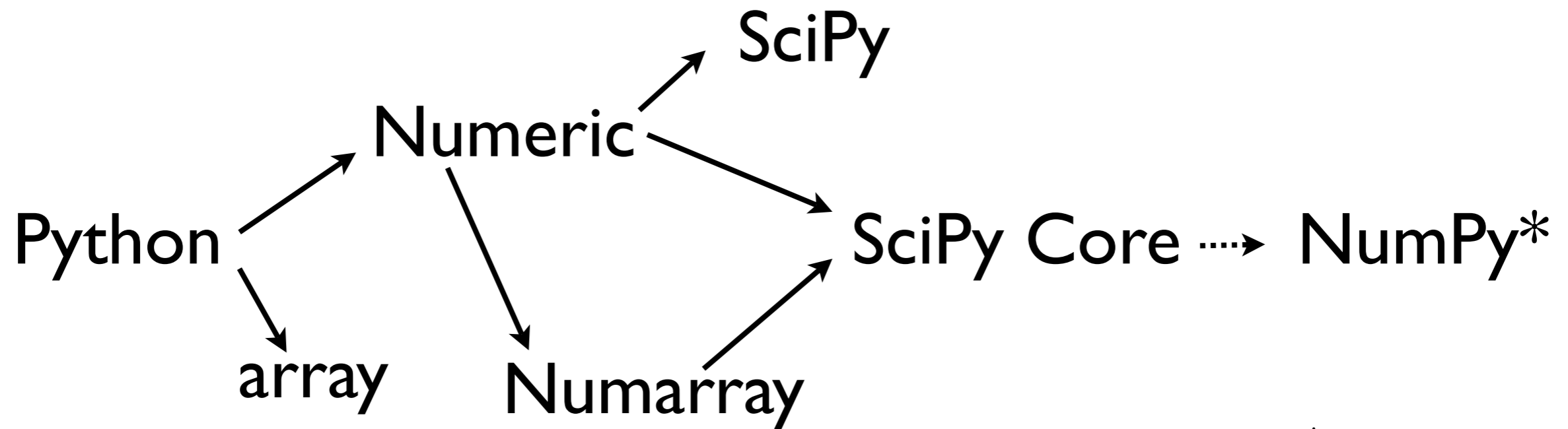
- `random ()`: random number 0.0 to 1.0.  
Doesn't include upper bound!
- `randint (a, b)`: pick an int from *a* to *b*.  
Inclusive bounds!
- `uniform (a, b)`: float from *a* to *b*. Doesn't include upper bound!
- Different RNGs: normal, gaussian ...
- `seed (x)`: Probably good practice ...

# random: List functions

- `choice (seq)`: pick one member of *seq*
- `randrange (start, stop [,step])`: pick one member of `range (start, stop [, step])`
- `sample (seq, len)`: randomly select a new sequence of length *len*
- `shuffle (seq)`: mix *seq* in place. Must be mutable!

# The long and sorry history of Python numerical libraries

A Tragedy in Four Acts



\*You are here

# Numpy contents

- `import numpy`
- Arrays and matrices, functors for operating on arrays & matrices
- Linear algebra, fourier transforms, random number generators
- Scripts to convert `numeric` & `numarray`
- More arrays and matrices

# Numpy matrices

- `ndarray`
- Homogenous type
- N-dimensional
- Fast
- Fiddly to change sizing

# Creating ndarrays

```
#construction with dimensions
```

```
>>> myMat = ndarray ((4,3))
```

```
>>> myMat
```

```
array([[ 6.76425276e-320,  1.02353805e-306,  1.89263714e-301],
       [ 1.89214586e-301,  6.76425276e-320,  1.39067116e-309],
       [ 4.45652510e-313,  1.02948779e-306,  1.01627641e-306],
       [ 4.50451010e+257,  6.76425276e-320,  8.34402697e-309]])
```

```
# construction with a matrix-like object
```

```
>>> myMat2 = array ((1, 2, 3), (4, 5, 6))
```

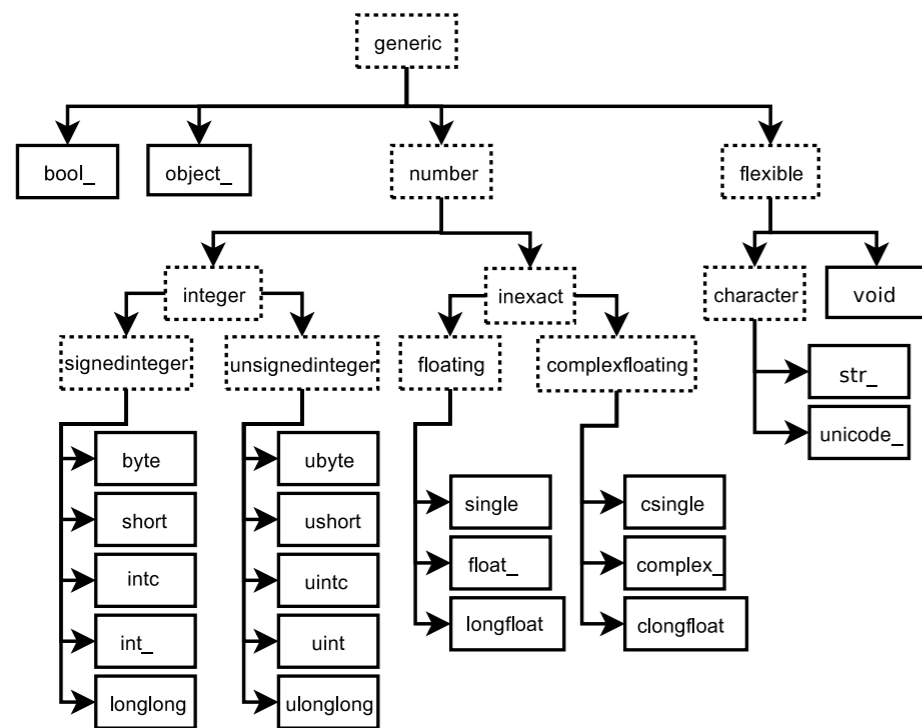
```
>>> myMat2
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
# and with arange
```

```
>>> arange (2, 5)
```

```
array([2, 3, 4])
```



Type	Bit-Width	Character
<b>bool_</b>	boolXX	'?'
byte	intXX	'b'
short		'h'
intc		'i'
<b>int_</b>		'l'
longlong		'q'
intp		'p'
ubyte	uintXX	'B'
ushort		'H'
uintc		'I'
uint		'L'
ulonglong		'Q'
uintp		'P'
single	floatXX	'f'
<b>float_</b>		'd'
longfloat		'g'
csingle	complexXX	'F'
<b>complex_</b>		'D'
clongfloat		'G'
<b>object_</b>		'O'
str_		'S#'
<b>unicode_</b>		'U#'
void		'V#'

# Creating ndarrays with types

```
>>> arange (2, 5, dtype=intc)  
array([2, 3, 4])
```

```
>>> arange (2, 5, dtype=int32)  
array([2, 3, 4])
```

```
>>> arange (2, 5, dtype=int64)  
array([2, 3, 4])
```

```
>>> arange (2, 5, dtype='i')  
array([2, 3, 4])
```

```
>>> arange (2, 5, dtype='O')  
array([2, 3, 4], dtype='object')
```

```
>>> arange (2, 5, dtype='f')  
array([2., 3., 4.], dtype='float32')
```

```
>>> arange (2, 5, dtype=float32)  
array([2., 3., 4.], dtype=float32)
```

# Indexing ndarrays

```
>>> myMat = ndarray ((4,3))
>>> myMat
array([[ 6.76425276e-320,  1.02353805e-306,  1.89263714e-301],
       [ 1.89214586e-301,  6.76425276e-320,  1.39067116e-309],
       [ 4.45652510e-313,  1.02948779e-306,  1.01627641e-306],
       [ 4.50451010e+257,  6.76425276e-320,  8.34402697e-309]])

>>> myMat[1,2]    # row 1, col 2
1.39067116e-309
>>> myMat[1]      # row 1, same as [1,:]
array ([ 1.89214586e-301,  6.76425276e-320,  1.39067116e-309])
>>> myMat[:,2]   # col 2
array ([ 1.89263714e-301,  1.39067116e-309,  1.01627641e-306,
        8.34402697e-309])
```

# Slice notation

- Multiple dimensions: e.g. [1:2,3:4]
- Can use steps: e.g. [::2,3] every 2nd row in the 3rd column
- Can use for set & get
- BUT cannot use to change array size

# unfuncs

- Universal functors
- Add on arrays
- Input & output all of the same size

1	7	8
9	0	1
3	2	2

 + 

0	7	1
9	3	1
0	4	7



1	14	9
18	3	2
3	6	9