Python types, modules and the standard library

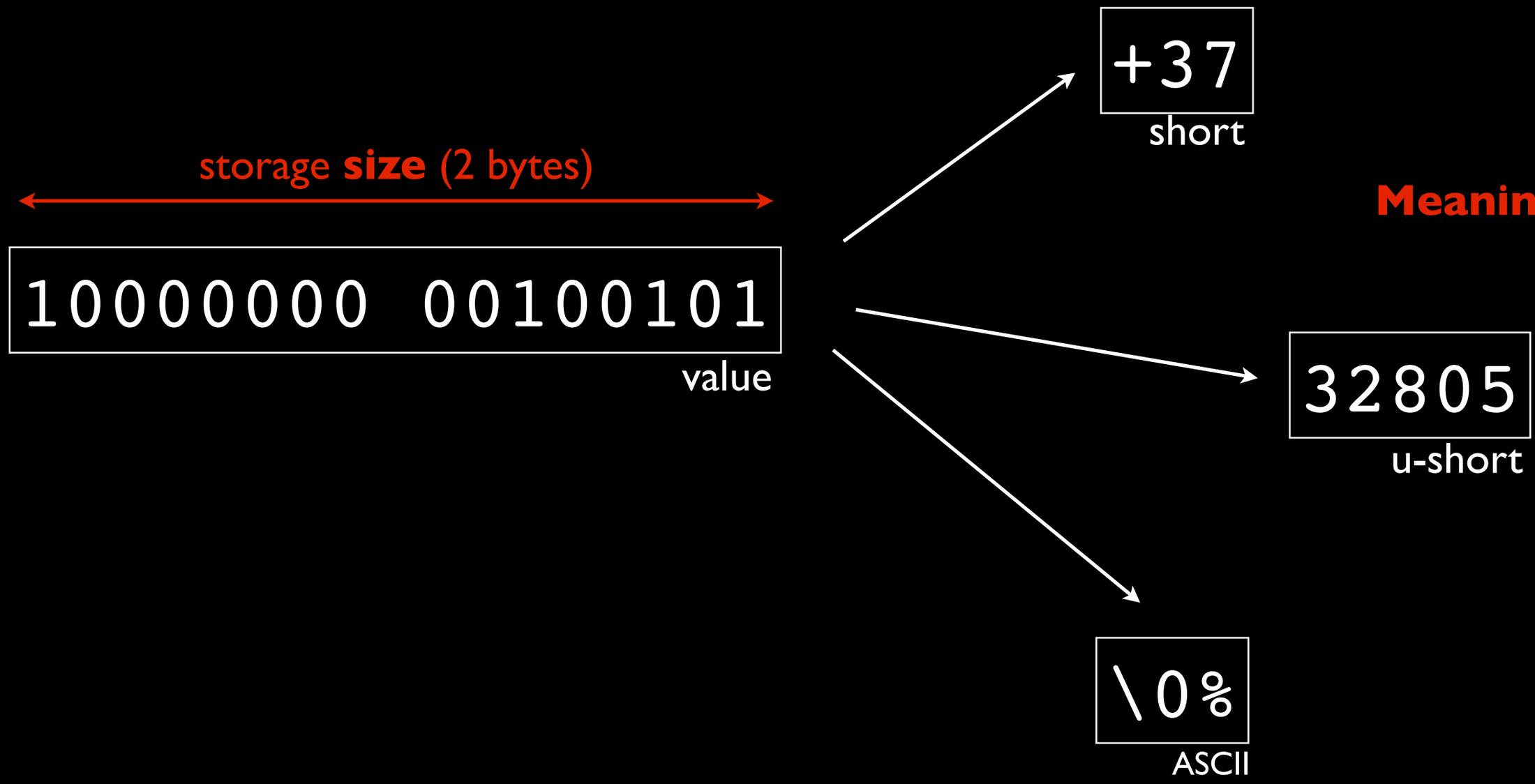
Andrew Walker

andrew.walker@bris.ac.uk

for Earth Scientists:
27 & 29 Sept. 2011

In general, what is a “type”?

In general, what is a "type"?



In general, what is a "type"?

Objects are Python's abstraction for data. All data in a Python program is represented by objects [...]

Every object has an identity, a type and a value. [...] An object's type determines the operations that the object supports (e.g., "does it have a length?") and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself).

Data model, from the *Python Language Reference*

Why?

Types allow *abstraction*: a language without types is a language where the programmer needs to explicitly convert between binary and whatever it is that the binary represents. This is present in all type systems.

Types *document* the program: for static typing the fact that type declarations exist allows somebody reading the program to know what you mean.

Safety: a strong type system can (and does) catch programming errors. Try passing a double precision real into a fortran function expecting two single precision reals with type checking turned off - then find the bug.

Performance: static typing allows a compiler to make optimisations; dynamic typing implies a run time overhead in time and memory.

The choice of a type system is a fundamental one involving trade offs when designing a language

Python is strongly but dynamically typed

for Earth Scientists:
27 & 29 Sept. 2011

~36 built in types in Python, but mostly you won't care about most of them

- Integer
- Real
- Complex
- String
- List
- Tuple
- Dictionary
- Set
- Frozenset
- File
- None
- Function

Float

Real number like 3.14159, -74.2, 34×10^{97} or 2.0.
Implemented using `double` in C so precision is system dependant (see `sys.float_info`).
Binary operators with integers "widen".

```
real = 34E97
pi = 3.14159
r = 22.0
area = pi * r**2
```

Operations with real numbers

Complex

Complex or imaginary number like 7+15*i* or $-34\times10^{97}i$. Works as two floats (`c.imag` and `c.real`). Integers and floats are widened.

```
cplx = 10+17E32j
pi = 3.14159265
e = 2.718281828
i = 1.0j
error = e**(pi*i)-1
```

Operations with complex numbers

Integer

A positive or negative whole number (1, -6, 432, etc.) Come in three sizes boolean, short integer and long integer. Conversion between these is automatic.

```
a = 1
b = -7L
c = True
print a + b + c
# prints -5
```

Operations with integers

String

A series of characters. Cannot change a string in place (immutable).

```
a = “abc”
b = ‘def’
c = a+b
print c
# prints abcdef
```

String concatenation

```
a = “abc”
print a[1]
# prints b
a[1] = ‘d’
# error
a = a[0]+‘d’+a[2]
# OK
```

Basic slicing and immutability

List

Like Matlab's cell arrays. A sequence of other types kept in order. Like strings, lists are zero based. Unlike strings, lists are mutable.

```
l = [“a”, 10, “abc”, 555.3]
for i in l:
    print i # a, 10, abc, 555.3
print l[2] # abc
l[2] = ‘cde’ # OK
len(l) # 4
l = [] # New empty list
```

Tuple

Tuples are immutable lists. Can do “tuple assignment” - useful for returning the results of functions

```
t = (“a”, 10, “abc”, 555.3)
for i in t:
    print i # a, 10, abc, 555.3
print t[2] # abc
t[2] = ‘cde’ # ERROR
len(t) # 4
t = () # New empty tuple
a, b = (‘abc’, 54) # a=‘abc’; b=54
```

tuples

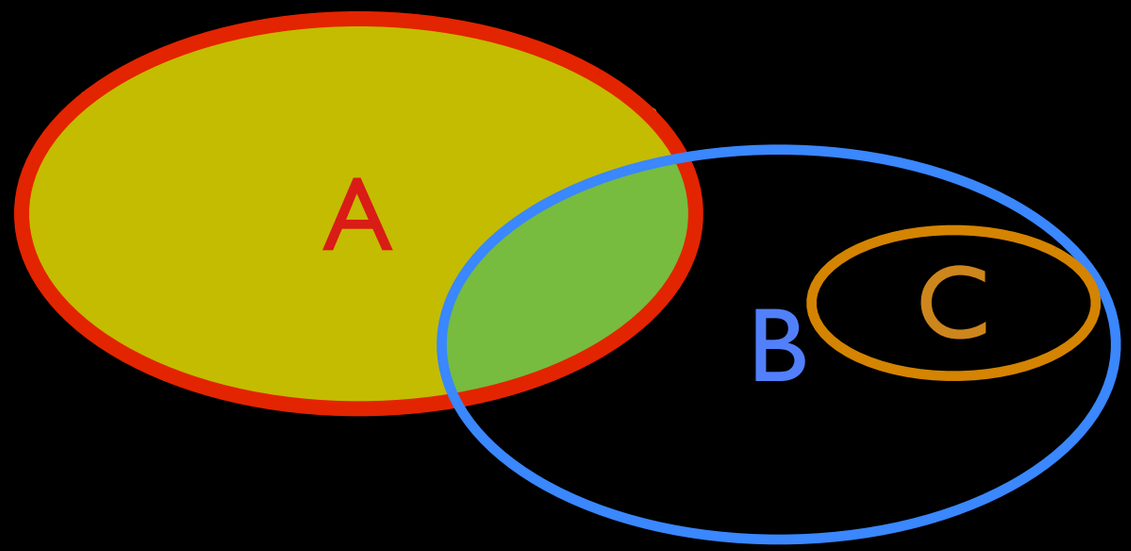
Set and Frozenset

Difference

`A - B`

Intersection

`A & B`



Union

`A|B == A|B|C`

superset

`B > C`

Set and Frozenset

Sets and frozensets behave in the same way, but sets are mutable and frozensets are not.

```
s1 = set()
s1.add(“hello”)
s2 = set([‘hello’])
s2.add(“goodby”)
s2 - s1 # Gives “goodby”
s2 | s1 # Gives set([“hello”, “goodby”])
s2 & s1 # Gives set([“hello”])
s2 ^ s1 # Gives set([“goodby”])
```

Dictionary

A collection of data (values) accessed via other, immutable, data (keys). An associative array.

```
d = {“a”: 10, “abc”: 555.3]
for k, v in d.items():
    print k # a, abc
    print v # 10, 555.3

for k, v in zip(d.keys(), d.values()):
    print k # a, abc
    print v # 10, 555.3

d[“abc”] = 77.8895 # OK
d[“zzz”] = [1, 2, 3] # OK
d = {} # New empty dictionary
```

File

Type representing data stored on disk (or something that looks like data on a disk). A file must be opened, used, and closed.

```
f = open('filename', 'w')
f.write('Some data')
f.close()
```

Using the file type for output

```
f = open('filename', 'r')
for line in f:
    print line
f.close()
```

Using the file type for input

Function

Function is a type too. You can assign functions to variables, pass them to functions, and generally become confused. e.g Useful for general integration of a function.

```
def addOne(x):
    return x+1
b = addOne
print b(4)
# prints 5
```

Assign a function

None

Special value (with its own type) that represents no data. Useful as default value for an optional argument to a function.

```
if x is None:
    # default case
else:
    # use x in calculation
```

None

Digression: what is typed?

for Earth Scientists:
27 & 29 Sept. 2011

Digression: what is typed in Fortran?

```
integer(dp) :: i
```
type

```
i = 37
print*, i
```

+37

```
! compiler error:
i = “string”
```

The variable `i` carries the type information in fortran

storage **size** (2 bytes)

10000000 00100101

value

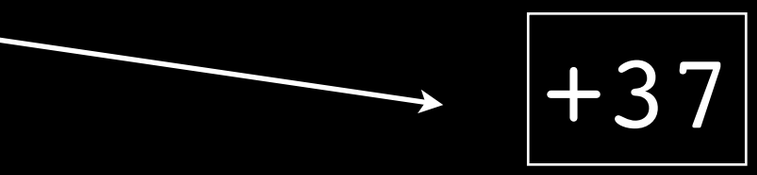
Digression: what is typed in Python?

type

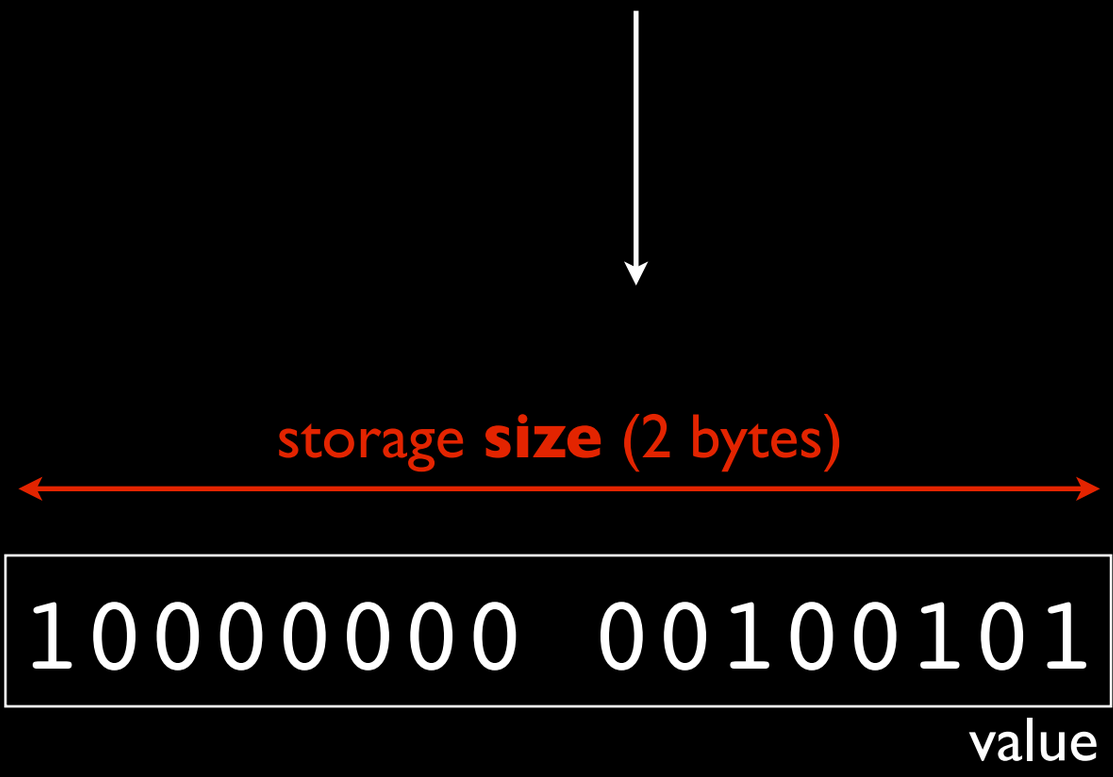
```
i = 37
print i

# This is fine:
i = "string"
```

type

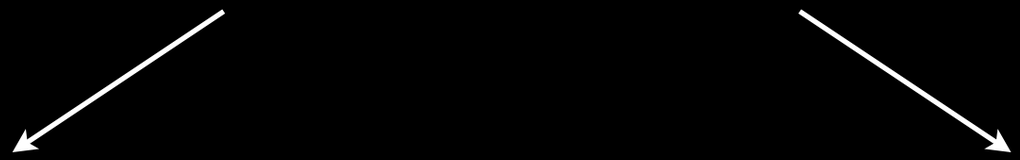


The data carries the type information in python



Namespaces and modules: Python is designed

Who is Mike?



use a namespace

use an alias

Mike from geophysics

Professor Kendall

Accidentally reusing the same name is a major problem for large pieces of code (more than one screenful) and makes code reuse difficult. Such "namespace pollution" can be avoided in Python (and Fortran) by using modules

```
geophys.mike
```

Modules

Modules are Python's containers for namespaces. They are just a file (called `name.py`) with Python code inside - i.e. they are just like the files you wrote in practical 1. Use `import` to load a module and create a namespace.

```
import foo

print foo.var
print foo.calc(10)
```

Main code

```
def calc(i)
    return i*5

var = 15
```

foo.py

Modules have names...

```
import foo

print foo.var
print foo.calc(10)

print foo.__name__

print __name__
```

Main code

```
def calc(i)
    return i*5

var = 15

print “foo loaded”
```

foo.py

... and are executed on import

`import foo`

Create a new namespace foo. Load foo.py. Access as foo.var and foo.calc(). Mangle names like _ _ internal_function.

`import foo as bar`

Just like import foo, but the namespace is bar. Access as bar.var etc. e.g. import math as m (to save keystrokes).

`from foo import *`

Load everything into your namespace. Access var and calc() directly. Dangerous. Do not use! Things names _var are not imported.

Scripts as modules and modules as scripts

```
import math
def hypot(a, b):
    return math.sqrt(a**2 + b**2)
if __name__ == “__main__”:
    import sys
    print hypot(float(sys.argv[1]),
                float(sys.argv[2]))
```

triangles.py

Can use triangles.py directly, or import triangles.

Have the OS find python

```
#!/usr/bin/env python
import math

def hypot(a, b):
    return math.sqrt(a**2 + b**2)

if __name__ == "__main__":
    import sys
    print hypot(float(sys.argv[1]),
                float(sys.argv[2]))
```

triangles.py

```
#> chmod u+x triangles.py
```

The standard library:

How Python comes with batteries included

for Earth Scientists:
27 & 29 Sept. 2011

Standard library

As well as being useful to organise your own code, Python modules and packages (modules containing other modules) are used to distribute useful code to others. ~300 modules in the standard library.

`sys`

`math`

`os.path`

`datetime`

`gzip`

`random`

http://docs.python.org/library/

math

Lots of mathematical functions. You will need use this to do anything beyond arithmetic. Look at cmath for functions that handle complex numbers properly

```
import math as m
a = m.radians(90)
m.sin(a) # ~1
m.cos(a) # ~0
```

datetime

Create variables to hold dates, times and the time difference between two dates or times. Can handle time zones.

```
import datetime
a = datetime.date(2011,9,27)
b = datetime.date(2011,9,29)
c = a - b
c.total_seconds()
# -172800.0 : two days
```

sys

This module allows you to interface with the operating system and shell environment.

```
import sys
sys.stdin # File object connected to <
sys.argv[1] # 1st command line argument
sys.argv[2] # 2nd command line argument
sys.argv[0] # script name
```

os.path

Chop up and join together file paths in a way that is aware of the convention of the computer where the script is running.

```
import os.path
os.path.join('a','b') # 'a/b'
os.path.split('a/b') # ('a', 'b')
os.path.splitext('a/b/f.o.txt')
# ('a/b/f.o', '.txt')
```

gzip

Allows you to work with compressed files as if you were using the built in file type. The ‘b’ means open in binary mode.

```
import gzip
f = gzip.open(“file.gz”, ‘rb’)
for line in f:
    print line
f.close()
```

random

A module to allow the generation of sequences of pseudo-random numbers. Based on Mersenne Twister generator.

```
import random
random.seed() # Set up PRNG
print random.randint(0, 7)
# a number between 1 and 6
```

http://docs.python.org/library/

http://docs.python.org/tutorial/

for Earth Scientists:
27 & 29 Sept. 2011