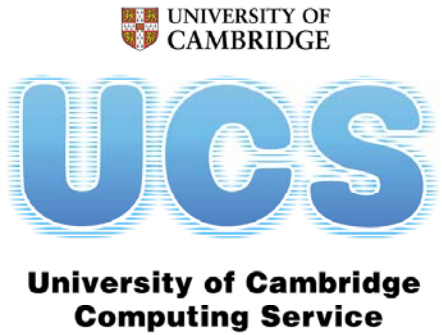
Python: 2 or 3?

Bruce Beckles

University of Cambridge Computing Service



Outline of talk

Background:

- What's so special about Python 3?

Python 2 or 3?

Main differences between Python 2 or 3

Migrating from Python 2 to 3

Python 2

Initial (2.0) release in October 2000

First release incorporating some Python 3 features (2.6) in October 2008:

- Incorporates some Python 3.0 features

Final major release (2.7) in July 2010

- Incorporates some Python 3.1 features

Most recent release (2.7.5) in May 2013

Python 2 is now in **extended maintenance**:

- Bugfix releases *only*
- Until May 2015 (see PEP 373)

Python 3

Initial (3.0) release in December 2008

First decent release (3.1) in June 2009

Most recent major release (3.3) in September 2012

Most recent release (3.3.2) in May 2013

Python 3 is the **current** version of Python:

- New features will *only* be added to Python 3, not Python 2

Python 3 evolution

Python 3.0: essentially an "*extended beta*" release

- **Do not use**
- I/O performance is rubbish

Python 3.1: First usable Python 3.x

Python 3.2: Improved version of Python 3.1

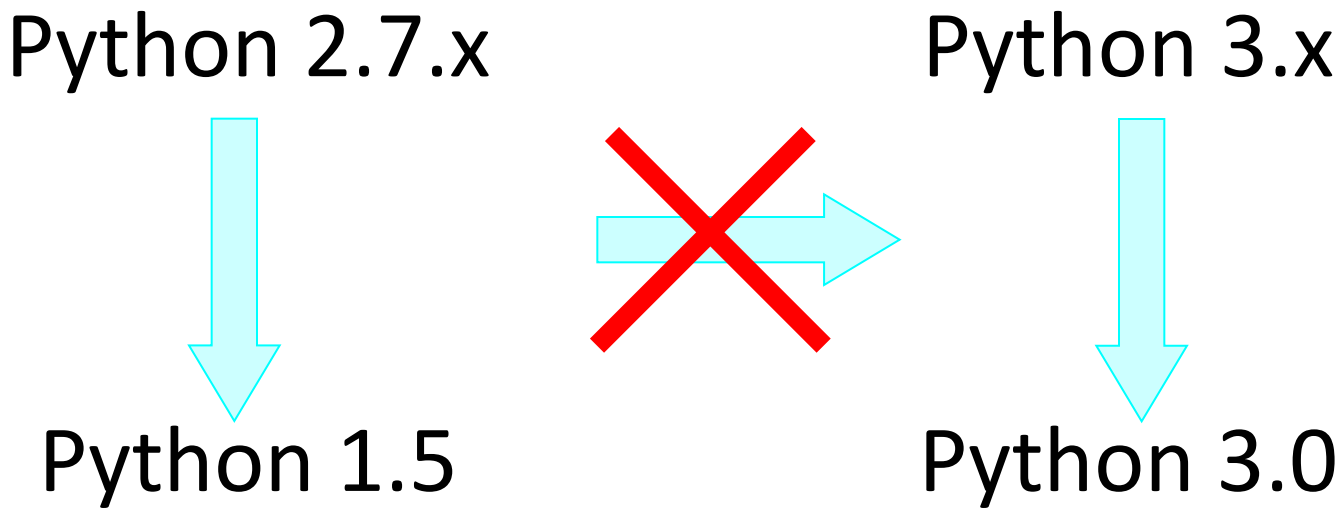
- No changes to syntax or core language (as compared to 3.1)
- More support for porting from Python 2.x

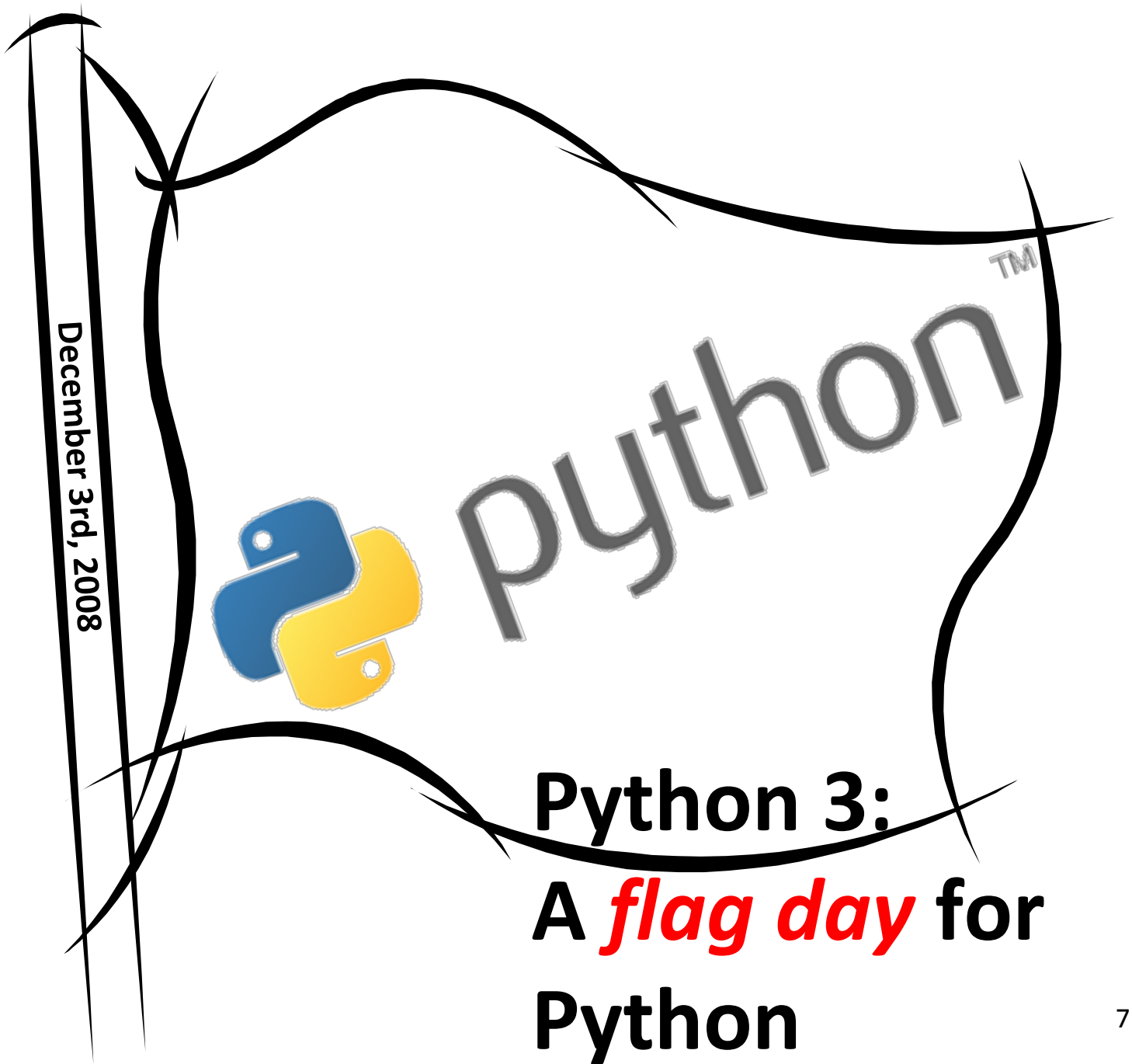
Python 3.3: Current major release of Python 3.x

- *Even more* support for porting from Python 2.x

What's so special about Python 3?

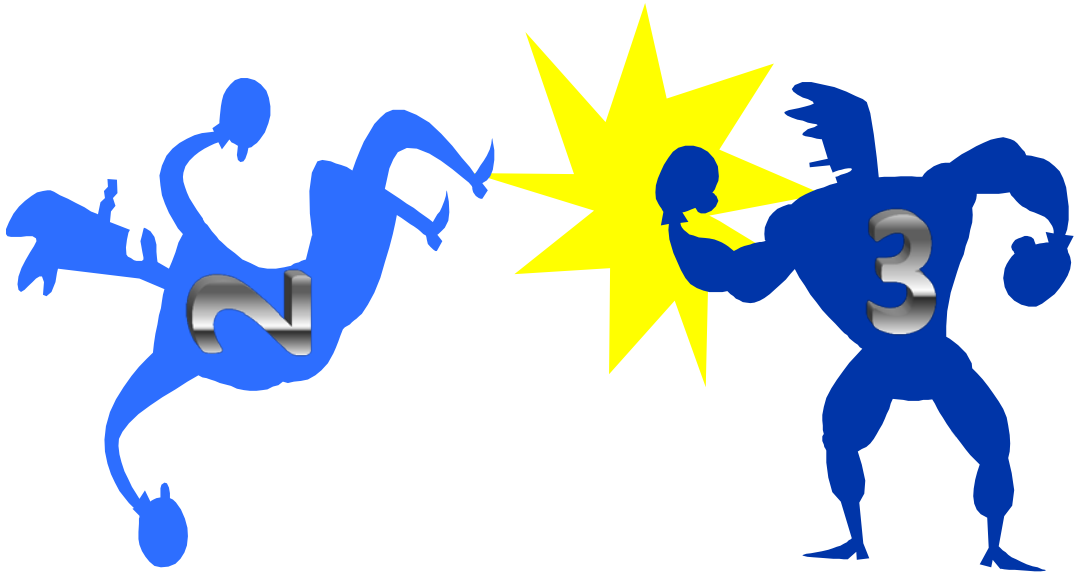
Backward compatibility





Python 3: A *flag day* for Python

Differences between Python 2 and Python 3



Why Python 3 at all?

Python 2.x	Python 3.x
Backward compatibility means very few features are ever removed (so the core language gets ‘bloated’)	Not backwardly compatible, so… Little used or obsolete features removed or demoted to modules
…and re-engineering is more-or-less impossible…	Re-engineered many of the internals and ways of doing things

Example: `print`

Python 2.x	Python 3.x
`print` is a *statement*: a special reserved word that does something "magic" (printing an object)	`print()` is a built-in *function*: behaves like any other function
Cannot be modified or overloaded	Can be modified or overloaded
Syntax not flexible or easily extensible	Can easily extend it in the future with additional options (keyword arguments) without breaking backward compatibility

Standard library

Python 2.x	Python 3.x
Over time has accumulated lots of “hacks”, obsolete functions/features, sub-optimal ways of doings	Re-designed, streamlined and improved standard library
Some similar modules	Similar/related modules rationalised
Many deprecated modules still shipped	Many deprecated modules removed

Unicode

Python 2.x

Support provided by Python module

In practice: **Aaaaaaargh!** Mixing Unicode and ASCII strings = AgONy!!!

Python 3.x

Built into core language: *all* `str`s are Unicode strings (default encoding: UTF-8)

Separate type, `bytes` (byte strings), if you really need strings where each character is stored as a single byte

String Formatting

Python 2.x

C `printf()`-style string formating via % operator on `strings`

```
"%10s %5.2f" %
(name, price)
```

Python 3.x

Built-in method of `strings` using .NET framework-style syntax

```
"{0:10} {1:5.2f}".
format(name, price)
```

C `printf()`-style string formating via % operator still supported, but discouraged

Numeric types

Python 2.x

Two types of integers:

- C-based integers (`int`)
- Python long integer (`long`)

"True" integer division, e.g.

5 / 2 = 2

Up to Python 2.6.x: Floats often **displayed** to a surprising number of decimal paces, e.g

1.0 / 10 = 0.1000000000000001

Python 3.x

All integers are Python long integers so only one integer type (`int`)

Integer division can change type, e.g.

5 / 2 = 2.5

From Python 3.1: Floats usually **displayed** to smaller number of decimal places, e.g.

1.0 / 10 = 0.1

Iterators and Sequences

"Treat it like a list and it will behave like an *iterator*"

```
for key in dictionary:
    print dictionary[key]
```

Python 2.x	Python 3.x
`next()` built-in function – backported to Python 2.6	`next()` built-in function
Several similar sequence types	Rationalised built-in sequence types
Fewer specialist container types	Many more specialist container types and subclasses

UCS

Example: `range`

Python 2.x

`range()` produces a `list` of numbers

`xrange()` produces an `xrange` object: doesn't produce an explicit list of numbers, instead knows what the current number is and can give the next one

Python 3.x

`range()` now produces an `range` object: doesn't produce an explicit list of numbers, instead knows what the current number is and can give the next one

Get an explicit list of numbers with `list(range())`

Exception Handling

Python 2.x

No proper exception hierarchy

Each type of exception can be different

- No standard behaviour
- …or information provided by exception

Python 3.x

All exceptions are derived from `BaseException`

- String exceptions are finally dead!

Exceptions cannot be treated as sequences

- Use `args` attribute instead

Exception chaining

Cleaned up APIs for raising and catching exceptions

Metaprogramming

Python 2.x

Supports function decorators

- Function wrappers

Supports metaclasses (but they can only process a class *after* the entire class has been executed)

Python 3.x

Adds support for function annotation

- Extends power of function decorators

Metaclasses can now process a class *before* any part of the class is processed and incrementally as methods are defined

Python 2 or 3?

Use Python 2.7.x for projects using:

Significant amounts of pre-existing Python 2.x code that can't easily be ported to Python 3.x

Python 2.x modules with no Python 3.x equivalent

Use Python 3 (3.1 or higher) for:

Everything else!

In particular:

Brand new projects

Code making significant use of **Unicode**

Code intended to be **used after 2016/2017**

Migrating from 2.x to 3.x

Make sure code runs under Python 2.7.x

Run Python 2.7.x with the "`-3`" command-line switch:

- Fix warnings

Use the Python-provided `2to3` tool to automatically port to Python 3.x:

- Test under Python 3.x and manually fix any remaining issues

Re-write exception handlers and any code that explicitly deals with Unicode

References (1)

Should I use Python 2 or Python 3 for my development activity?: http://wiki.python.org/moin/Python2orPython3

- Includes good set of links on differences between Python 2 and Python 3, and on porting to Python 3

What's New In Python 3.0: http://docs.python.org/3/whatsnew/3.0.html

References (2)

Python 3: the good, the bad, and the ugly (David Beazley)
http://www.ukuug.org/newsletter/18.3/#pytho_david

- Note that some of the criticisms of Python 3(.0) in this article do not apply to later versions of Python 3.x (most notably the I/O performance has been fixed in Python 3.1 and later)

Let's talk about Python 3.0 (James Bennett)
http://www.b-list.org/weblog/2008/dec/05/python-3000/

Porting tools

Porting from Python 2.x to Python 3.x: `2to3`

http://docs.python.org/3/library/2to3.html

- Use version that ships with release of Python 3.x ***to*** which you are porting

Porting from Python 3.x to Python 2.7: `3to2`

http://wiki.python.org/moin/3to2
http://pypi.python.org/pypi/3to2
http://bitbucket.org/amentajo/lib3to2
http://code.google.com/p/backport/

Questions?