What's the deal with Python 3?

Peter Hill

Outline

- A very brief history of Python
- Why Python 3?
- The main differences
- Cool features
- Maintaining compatibility

A very brief history of Python

- Guido van Rossum started work on Python in 1989
  - van Rossum is the *Benevolent Dictator for Life*
- First version released in 1990
- Python 2.0 released in 2000
  - Where Python really came into its own
  - Introduced features like list comprehension and fancy garbage collection
- Python 2.6 and 3.0 released in 2008
  - Fixed lots of problems, but in a backwards incompatible fashion
  - Many features were simultaneously backported to 2.6
- Python 3.1 released in 2009
  - Fixed some glaring performance problems in 3.0
- Python 2.7 released in 2010
  - Many features from 3.1 backported to 2.7
  - 2.7 last release in 2.x series
- Python 3.3 released in 2012
  - Official end of Python 2 feature releases
- Python 2.7 end of life in 2020

The Zen of Python

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts
- There should be one – and preferably only one – obvious way to do it
- Although that way may not be obvious at first unless you're Dutch

Why Python 3?

What's wrong with Python 2?

- Various design decisions hindered improvements
- Some features cause of subtle bugs, e.g.
  - Unicode 1: little distinction between text and bytes
  - Unicode 2: `ê = 1` *might* work in the interpreter but not in scripts
  - `0777` interpreted as an octal number due to leading `0`
  - `input` automatically `eval`uated what the user typed
- Some names didn't follow convention
- Fixing these things required changing the meaning of existing code
- Python 3 now has a ton more features!

Major differences

- `bytes` need to be `decoded` into `str` (text)
- `str` are Unicode by default
- No distinction between `int` (machine precision) and `long` (arbitrary precision)
- `print` is a function rather than a statement
- Division of two integers returns a `float` by default
- One type of `class` rather than two (!)
- Removed some synonyms for functions: e.g. Python 2 has both `!=` and `<>` for "not equals"
- Many functions now return *iterators* rather than `lists`
- Relative imports must be explicit
- New keywords: `with`, `as`, `True`, `False`, `None`
- Better exception handling

Major differences

- `bytes` need to be `decoded` into `str` (text)
- `str` are Unicode by default
- No distinction between `int` (machine precision) and `long` (arbitrary precision)
- **`print` is a function rather than a statement**
- **Division of two integers returns a `float` by default**
- One type of `class` rather than two (!)
- Removed some synonyms for functions: e.g. Python 2 has both `!=` and `<>` for "not equals"
- **Many functions now return *iterators* rather than `lists`**
- **Relative imports must be explicit**
- New keywords: `with`, `as`, `True`, `False`, `None`
- **Better exception handling**

`print` function

Why?!

- Weird syntax for no newline: `print a,`
- Weird syntax for printing somewhere else: `print >> output, a`
- Difficult to add new syntax: how to change the separator?
- Not very flexible

`print` function

Useful things

- Change the separator/newline: `print("a", "b", sep="...", end="")`
- Can be used in new contexts: `map(print, "abc")`
- Can be overridden: (don't do this!)

```
log = []
old_print = print
def print(*args, **kwargs):
    log.append(' '.join(args))
    old_print(*args, **kwargs)
```

- or specialised:

```
my_print = lambda *args, **kwargs: print(*args, **kwargs, sep=":")
```

Digression: expression vs statement

Expression

- Has a value: `2 / 3`
- Can have a name: `cats = ["Garfield", "Maru", "Ziggy"]`
- Can pass them to functions: `is_square(2 + 2)`
- Limited to: identifiers (`cats`), literals (`2`, `"Ziggy"`) and operators (`+`, `()`)

Statement

- Made of expressions and syntax
- Makes up an executable line: `if cats is not None:`
- Can't be given a name: `assign_f = (f = 1)`
- Can't be passed to functions: `is_square(if)`

Division

The first major trip hazard

- In Python 2: `2/3 == 0` and `3/2 == 1`
  - Sometimes surprising, but sometimes what you want
- In Python 3: `2/3 == 0.666...` and `3/2 == 1.5`
  - Less surprising, unless you are a C programmer
- If you want integer division, use `//` instead:
  - `2//3 == 0` and `3//2 == 1` for Python 2 and 3
  - Rounding is down towards negative infinity

Iterators and views

The other major trip hazard

- Many functions now return iterators or views
- These are lightweight, memory-efficient objects
- Iterators only get evaluated when you try to use them and become empty afterwards

```
>>> cats = {"Garfield": False, "Maru": True, "Ziggy": True}
>>> real_cats = filter(is_real, cats)
# <filter at 0x7fc05689bba8> in Python 3
# [('Maru', True), ('Ziggy', True)] in Python 2
>>> list(real_cats)   # Convert to a list
[('Maru', True), ('Ziggy', True)]
>>> list(real_cats)   # We've already "consumed" the filter
[]
```

Iterators and views

Views

- Dynamic view into an object
- Reduces memory footprint

```
>>> cat_names = cats.keys()
>>> print(cat_names)
["Garfield", "Maru", "Ziggy"]
>>> cats["Pink Panther"] = False
>>> print(cat_names)
["Garfield", "Maru", "Ziggy", "Pink Panther"]
# ["Garfield", "Maru", "Ziggy"] in Python 2
```

Relative imports

A problem for packages

- Take a simple package like this:

```
blackholes/
|- __init__.py
|- relativity.py
```

- . . . use one file from another:

```
# __init__.py
import relativity
```

- And now try to use the package:

```
import blackholes
```

- In Python 2 this works due to the implicit relative import
- But this caused all sorts of headaches, like what if there's another `relativity.py` in your `PYTHONPATH`?

Relative imports

Python 3 is more explicit

- In Python 3 we get an error:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/tmp/blackholes/__init__.py", line 1, in <module>
    import relativity
ModuleNotFoundError: No module named 'relativity'
```

- To fix this for Python 3, we need to be explicit about which `relativity` we want to import:

```
# __init__.py
from . import relativity
```

- This will then work properly

Exceptional exceptions

Catching multiple exceptions

```
>>> try:
...   1/0
... except TypeError, ZeroDivisionError:
...     print("Exception suppressed")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:
...     1/0
... except (TypeError, ZeroDivisionError):
...     print("Exception suppressed")
...
Exception suppressed
```

Exceptional exceptions

Catching multiple exceptions

```
>>> try:
...     1/0
... except TypeError, ZeroDivisionError:
  File "<stdin>", line 3
    except TypeError, ZeroDivisionError:
                    ^
SyntaxError: invalid syntax
```

Exceptional exceptions

Exceptions during exceptions (inceptions)

```
>>> try:
...     1/0
... except Exception:
...     logging.exception("Something went wrong")
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'logging' is not defined
```

Exceptional exceptions

Exceptions during exceptions (inceptions)

```
>>> try:
...     1/0
... except Exception:
...     logging.exception("Something went wrong")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'logging' is not defined
```

Features in 3 not in 2

f-strings (3.6)

```
print("Hello {}".format(name))  # Python 2.7 and 3.5
print(f"Hello {name}")          # Python 3.6
```

Dictionaries remember insertion order (3.6)

- Used to be an implementation detail
- Now part of the specification
- Also improved memory usage and speed

Infix matrix multiplication operator (3.5)

```
S = dot((dot(H, beta) - r).T,
        dot(inv(dot(dot(H, V), H.T)), dot(H, beta) - r))
# becomes
S = (H @ beta - r).T @ inv(H @ V @ H.T) @ (H @ beta - r)
```

Features in 3 not in 2

More general unpacking (3.5)

```
>>> print(*[1], *[2], 3, *[4, 5])
1 2 3 4 5

>>> def fn(a, b, c, d):
...     print(a, b, c, d)
...

>>> fn(**{'a': 1, 'c': 3}, **{'b': 2, 'd': 4})
1 2 3 4
```

Features in 3 not in 2

Underscores in numeric literals (3.6)

```
>>> 1_000_000_000_000_000
1000000000000000
>>> 0x_FF_FF_FF_FF
4294967295
```

Easier debugging (3.7)

- `breakpoint()` drops you into a debugger

Type hinting (since 3.0, but useful in 3.5)

- Specify types of function arguments and return values
- Still need an external tool to verify

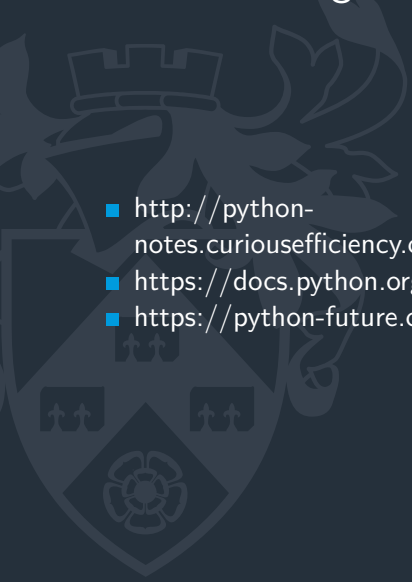
```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Advice

- Write new projects in Python 3.5+!
- Don't bother trying to be backwards compatible
- But if you need to (official advice):
  1. Only worry about supporting Python 2.7
  2. Make sure you have good test coverage (`coverage.py` can help; `pip install coverage`)
  3. Learn the differences between Python 2 & 3
  4. Use `Futurize` (or Modernize) to update your code (e.g. `pip install future`)
- If you really can't move to 3,

```
from __future__ import print_function
from __future__ import division
```

Further reading



- http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html
- https://docs.python.org/3/howto/pyporting.html
- https://python-future.org/