Python:
The New Stuff

Steve Johnson
CWRU Hacker Society
9/14/2011

Whaaat?

Timeline: 2.x

2006
* 2.5: Let's start thinking about py3k

2008
* 2.6: A couple of features backported from newly-released 3.0

2010
* 2.7: Released simultaneously with 3.2, many features backported, many performance improvements

Timeline: 3.x

- 2008 * 3.0: More design statement than practical platform; for library maintainers
- 2009 * 3.1: Bug fixes and performance (language feature moratorium)
- 2010 * 3.2: Hey, this is actually kind of cool!
- 2012 * 3.3: Coming next year

Topics For Today

* String formatting
* Dict and set comprehensions
* Class decorators
* Unicode
* Functions (annotations, args)
* Grab bag: argparse, OrderedDict, abc

Brief Commentary: Things Not to Write

* Logging implementations
* Testing frameworks
* Argument parsers
* SQL-generating functions
* Date calculations
* Cryptography

We have libraries for these, and you will probably screw it up.

String Formatting

```
"""String formatting examples for Python 2.6 through 3.2."""

a = 'Tom'
b = 'Dick'
d = 'Harry'

# old:
print("The story of %s, %s, and %s" % (a, b, d))
print("The story of %(a)s, %(b)s, and %(d)s" % locals())

# new:
# http://www.python.org/dev/peps/pep-3101/
print("The story of {0}, {1}, and {c}".format(a, b, c=d))
print("The story of {a}, {b}, and {d}".format(**locals()))

some_dict = dict(a=1, b=2)

print("Item 'a' in {0.__class__.__name__}: {0[a]}".format(some_dict))
```

2.5+

Dict and Set Comprehensions

```
  # You used to only have these:
squared_evens = [x**2 for x in range(8) if x % 2 == 0]
print('squared evens: {0}'.format(squared_evens))

# to get a dict you would do this:
# (makes {0: 'A', 1: 'B', ...})
letter_for_int = dict((i, chr(65+i)) for i in range(10))

# to get a set you would do this:
s = set(i for i in range(10))
```

2.4+

Dict and Set Comprehensions

```
def invert(d):
    """Swap keys and values on d"""
    return {v: k for k, v in d.items()}
```

```
def even_keys(d):
    """Return a set of all even keys from d"""
    return {k for k in d.keys() if k % 2 == 0}

d = {i : chr(65+i) for i in range(10)}
print('dictionary:  {0}'.format(d))
print('inverted:    {0}'.format(invert(d)))
print('evens only:  {0}'.format(even_keys(d)))
print('set literal: {0}'.format({0, 1, 2, 3, 4}))
```

2.7+

Class Decorators

```
commands = {}

def command(command_string):
    def make_command(cls):
        commands[command_string] = cls
        return cls
    return make_command

@command('fight')
class FightCommand(object):

    def fight(self):
        print("I'M FIGHTING")

if __name__ == '__main__':
    commands['fight']().fight()
```

2.6+

Now it starts to get weird.

Unicode

* For Python 2, read this: http://docs.python.org/howto/unicode.html
* For Python 3, read http://docs.python.org/py3k/howto/unicode.html

Most of this is just me talking. No one wants to see complicated Unicode pet tricks.

* In Python 3, everything is Unicode unless explicitly converted to bytes
* I don't want to spend any more time on this. **Go read!**

New Function Signatures

```
def old_func(positional_arg, arg_with_default=None,
             *varargs, **kwargs):
    """Simple demonstration of old function signatures"""
    # ...
    print('Required arg:    {}'.format(positional_arg))
    print('Optional arg:    {}'.format(arg_with_default))
    print('Unnamed varargs: {}'.format(varargs))
    print('Keyword varargs: {}'.format(kwargs))

> old_func(1, 2, 3, 4, a=5, b=6):

Required arg:    1
Optional arg:    2
Unnamed varargs: (3, 4)
Keyword varargs: {'a': 5, 'b': 6}
```

2.7-

New Function Signatures

* But what if we want this:

```
make_list(*args, reverse=False)
```

* You can have it, but...

New Function Signatures

```
def make_list(*args, **kwargs):
    """If we want to both capture positional varargs and have optional
    keyword args, we have to do this crap
    """
    bad_kwargs = set(kwargs.keys()) - {'reverse'}
    if bad_kwargs:
        fmt = ("varargs_and_optional_kwargs() got an unexpected keyword"
               " argument '{}'")
        raise TypeError(fmt.format(bad_kwargs.pop()))
    if kwargs.get('reverse', False):
        return list(reversed(args))
    else:
        return list(args)

print('Non-reversed: {}'.format(make_list(1, 2, 3)))
print('Reversed:     {}'.format(make_list(1, 2, 3, reverse=True)))
```

2.7-

New Function Signatures

```
# http://www.python.org/dev/peps/pep-3102/

def make_list_3(*args, reverse=False):
    """Much better!"""
    if reverse:
        return list(reversed(args))
    else:
        return list(args)


def all_kw_only(*, a=None, b):
    print(a, b)

> all_kw_only(a=1, b=2)

  1 2
```

3.0+

Function Annotations

```
# http://www.python.org/dev/peps/pep-3107/

import inspect

def cat(a:str, b:str) -> str:
    return a + b

spec = inspect.getfullargspec(cat)

> print(spec.annotations)
  {'a': <class 'str'>, 'b': <class 'str
<class 'str'>}

> print(cat(1, 2))
```

Q: Will cat(1, 2) throw an error?
A: No

Q: Then what to function annotations do, functionally?

3.0+

NOTHING

3.0+

argparse

The rest of the slides up to "Help I'm stuck on Python 2.x!" are of the if-people-are-still-awake sort.

* getopt is usually too primitive
* optparse is better but doesn't support subcommands or positional arguments
* argparse is a rewrite that **does more, better**

2.7, 3.2+

argparse

Not going to walk you through the API, just show you this, which you get pretty much for free:

```
> ct clockin --help
usage: ct clockin [-h] [--config] [--away] [-t TIME] [project]

Start logging hours to a project

positional arguments:
  project

optional arguments:
  -h, --help            show this help message and exit
  --config              update configuration options
  --away                set Adium status to Away in addition to
                        changing the message
  -t TIME, --time TIME  time to log for checkin
```

2.7, 3.2+

OrderedDict

* Config files may be key-value, but humans care about ordering
* OrderedDict remembers insertion order, so you can programmatically modify config files without making users sad
* Also used for namedtuple._asdict()

2.7+

Abstract Base Classes

"ABCs are simply Python classes that are added into an object's inheritance tree to signal certain features of that object to an external inspector. Tests are done using isinstance(), and the presence of a particular ABC means that the test has passed."

2.6+

Abstract Base Classes

* You can confirm that an object satisfies certain properties (e.g. "acts like a list") without lots of verbose tests
* Defensive duck typing
* Not universally loved

2.6+

Abstract Base Classes

```
# http://www.python.org/dev/peps/pep-3119/

from abc import ABCMeta, abstractmethod

from class_decorators import command, commands

class Command(metaclass=ABCMeta):

    @abstractmethod
    def fight(self):
        pass


@command('fight')
class FightCommand(Command):

    def fight(self):
        print("I AM ALSO FIGHTING")


if __name__ == '__main__':
    commands['fight'].fight()
```

imports from earlier example!

2.6+

concurrent.futures

* These could probably fill an entire talk on their own
* http://www.python.org/dev/peps/pep-3148/
* "Do work asynchronously and get back to me"

Help, I'm stuck on 2.x!

from __future__ import...

* absolute_import (2.5)
* with_statement (2.5)
* print_function (2.6)
* unicode_literals (2.6)

Use these whenever possible!

Help, I'm stuck on 2.x!

* 2to3 is a pseudo-magical tool that converts your code from 2.x to 3.x
* Runs multiple "fixers" that can be enabled and disabled

3.3?

* A bunch of internal stuff you don't care about

* Syntax for delegating to a subgenerator

* August 2012

```
def complex_generator():
    subgen = complicated_expr
    yield from subgen()
```

Resources

* http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/
* http://lucumr.pocoo.org/2010/2/11/porting-to-python-3-a-guide/
* http://python3porting.com/

Thanks!

Have fun!