Python 3, the Argument Sketch

Linux Symposium
June 14, 2011, University of Ottawa

Ian Ward

http://genericconsulting.ca/
http://excess.org/
Overview

- Language Basics
- Significant Libraries
- Development of Python Itself
- Alternative Implementations
- Examples
What is Python?

Python is an interpreted, general-purpose high-level programming language whose design philosophy emphasizes code readability

http://en.wikipedia.org/wiki/Python_(programming_language)

• 20 years old
• imperative, functional, OO w/multiple inheritance
• duck, dynamic, strong typing
Uncluttered Appearance

- ~30 keywords
- ~130 buitins (classes, functions, constants)

Less Punctuation

- Many operators are keywords: `and`, `or`, `not`, `is`, `in`
- Ending statements with `;` is optional
- No end-block keyword or symbol
Documentation

http://docs.python.org/

- every version, every module
- easy to read
- reStructuredText (.rst)
- Sphinx
  http://sphinx.pocoo.org/
- rst2pdf
  http://code.google.com/p/rst2pdf/
Being Pythonic

Zen of Python:

```python
>>> import this
```

http://www.python.org/dev/peps/pep-0020/
Python Popularity

Significant Whitespace

```python
def greet(name):
    print('hello', name)

greet('Jack')
greet('Jill')
```

Blocks preceeded by a trailing:

No closing marker

Doesn't apply to code within (), [], {}, """"
Python's Not the Only One

- ABC
- CoffeeScript
- Curry
- F# (if #light "off" is not specified)
- Haskell (only for where, let, do, or of clauses when braces are omitted)
- ISWIM, the abstract language that introduced the rule
- YAML

The Future

It seems clear that languages somewhat different from those in existence today would enhance the preparation of structured programs. We will perhaps eventually be writing only small modules which are identified by name as they are used to build larger ones, so that devices like indentation, rather than delimiters, might become feasible for expressing local structure in the source language.

PEP 8 - Style Guide for Python Code

Use 4 spaces per indentation level.

Never mix tabs and spaces.

http://www.python.org/dev/peps/pep-0008/
Configure Your Editor

Tab Stops
- Tab width: 4
- Insert spaces instead of tabs

Automatic Indentation
- Enable automatic indentation

File Saving
- Create a backup copy of files before saving
- Autosave files every 10 minutes

Help Close
Automated Clean Up

In the python source/installation:

```
Tools/scripts/reindent.py
```

On Debian or Ubuntu:

```
apt-get install python-examples
/usr/share/doc/python*/examples/Tools/scripts/reindent.py
```
Killer Features
"Batteries Included"

http://xkcd.com/353/

http://excess.org
Science

NumPy and SciPy

- N-dimensional Array manipulations
- statistics
- optimization
- numerical integration
- linear algebra
- Fourier transforms
- signal processing
- image processing
- ODE solvers
- sophisticated random number capabilities
Games

Stackless

http://www.stackless.com/

Microthreads, Channels, Scheduling

http://www.eveonline.com/
The 'Net

Twisted

An event-driven networking engine

http://twistedmatrix.com/

- Servers and clients for HTTP, NNTP, IMAP, SSH, IRC, FTP and others
- With support for TCP, UDP, SSL/TLS, multicast and Unix sockets
Python Development

- BDFL
- Mailing Lists
- PEPs
  - rationale, explanation, related work
  - influenced Tcl, Erlang
Python 3

a.k.a. Python 3000, py3k

PEP 3000: "guidelines for Python 3000 development" - April 2006
http://www.python.org/dev/peps/pep-3000/

- intentionally break backwards compatibility
- clean up many long-standing language warts
- tools and plan for migration of external code from 2.x to 3.x
Python 3

The Plan

- expect a 5 year transition (we're 2 years in now)
- all new development in 3.x
- continue to support 2.x and backport some features
- moratorium on language changes (just expired)
## Python 3

<table>
<thead>
<tr>
<th>Project</th>
<th>Python 3 support</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumPy</td>
<td>YES</td>
</tr>
<tr>
<td>SciPy</td>
<td>YES</td>
</tr>
<tr>
<td>Stackless</td>
<td>YES</td>
</tr>
<tr>
<td>Twisted</td>
<td>no</td>
</tr>
<tr>
<td>Django</td>
<td>no (promised this year)</td>
</tr>
<tr>
<td>PIL</td>
<td>no</td>
</tr>
<tr>
<td>Jython</td>
<td>no</td>
</tr>
<tr>
<td>Iron Python</td>
<td>no</td>
</tr>
<tr>
<td>PyPy</td>
<td>no</td>
</tr>
</tbody>
</table>

[http://pypi.python.org/pypi?:action=browse&c=533](http://pypi.python.org/pypi?:action=browse&c=533)
Alternative Implementations

Jython

- compiles to Java bytecode
- use Java libraries
- use JVM's garbage collector and JIT

Iron Python

- run code on the .Net CLI
- use .Net libraries

Shed Skin

Cython
Alternative Implementations

PyPy

- 8 years old
- fast without code modification (~3.4x faster)
- sandboxing
- stackless mode
- frontends for Python, JavaScript, Prolog, Smalltalk
  
  https://lwn.net/Articles/436970/

- backends for x86 32+64 bit, CLI (.Net), JVM (Java)
  
  http://pypy.org/
Alternative Implementations

How fast is PyPy?

Plot 1: The above plot represents PyPy trunk (with JIT) benchmark times normalized to CPython. Smaller is better.

http://speed.pypy.org/
Strings

Immutable

```python
a = "these are the same"  # (8-bit in 2.x, Unicode in 3.x)
b = 'these are the same'

c = b'binary data'

d = u'Unicode'  # (2.x)

e = r'c:\strings\with\backslashes'
f = r'\[.\]+\s(\w+)'

# ur'\' and br'\' work too

g = """Triple-quoted strings
may span multiple lines
and include the newlines ^^ they contain"""
```
Strings

```python
>>> a
'these are the same'
>>> len(a)
18
>>> a[0]
't'
>>> a[-1]
'e'
>>> a[6:9]
'are'
>>> a.startswith('the')
True
>>> 'he' in a
True
>>> a.count('he')
2
```
Lists

Mutable, implemented as arrays, average O(1) append

```python
>>> i = a.split()
>>> i
['these', 'are', 'the', 'same']
>>> len(i)
4
>>> i[0]
'these'
>>> i[-1]
'same'
>>> i[1:3]
['are', 'the']
>>> 'he' in i
False
>>> i.count('the')
1
```
Tuples

Immutable lists

```python
>>> t = tuple(i)
>>> t
('these', 'are', 'the', 'same')
>>> t[1:3]
('are', 'the')
>>> t[1] = "aren't"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```
def greet(family_name, given_name=None, salutation=None):
    g = 'Hello '
    if salutation:
        g = g + salutation + ' '
    if given_name:
        g = g + given_name + ' '
    print(g + family_name)

greet('Weiner')
greet('Doe', 'John')
greet('Smith', salutation='Mr.')

Hello Weiner
Hello John Doe
Hello Mr. Smith
Truthiness

Not Truthy:

• False
• None
• 0

Empty things are also not Truthy:

• ''
• []
• ()
• {}
• set([])

Almost everything else is Truthy
Print Statement (2.x)

```python
print 'Hello, world!'
Hello, world!
print 'one and', 'two and',
print 'three and four'
on one and two and three and four
```
Print Statement (2.x)

```python
import sys
print >>sys.stderr, 'Error, world!'
```
It Gets Worse

input()

```python
>>> raw_input('Your name? ')
Your name? Arthur
'Arthur'
>>> input('Eight times two? ')
Eight times two? 8*2
16
```

`raw_input()` becomes `input()` in 3.x
Print Function (3.x)

```python
from __future__ import print_function  # for 2.x

print('Hello, world!')

print('one and', 'two and', sep=' ', end=' ')
print('three and four')

import sys
print('Error, world!', file=sys.stderr)
```
Numbers

```python
x = 42
y = 10000000000000000000  # int and long merged in 3.x
z = 0x42ab  # hex
a = 0o775  # octal, 2.x supports C-style octal, eg. 0775
b = 0b1011  # binary

p = 2.934

q = 3+1j  # complex

import decimal
m = decimal.Decimal("3.14")  # exact decimal values

import fractions
n = fractions.Fraction(1, 3)  # exact rational values
```
Math

```python
>>> 1 / 2  # floor division in 2.x, normal division in 3.x
0.5
>>> 1 // 2
0
>>> 1.0 / 2
0.5
>>> abs(-3)
3
>>> round(0.375, 2)
0.38
>>> sum([29, 36, 12, 98])
175
```

Note: Floor division always rounds towards -Inf
Dictionaries

Implemented with hash tables, fast path for string keys

```python
>>> d = {'count': 3, 'product': 'coconuts'}
>>> len(d)
2
>>> d['product']
'coconuts'
>>> d['count']
3
>>> 'count' in d
True
>>> 3 in d
False
```

```python
>>> d = dict(count=3, product='coconuts')
```
String Formatting

```python
>>> count = 3
>>> product = 'coconuts'
>>> s = '%d %s remaining.' % (count, product)
>>> print(s)
3 coconuts remaining.
>>> t = '%04d %-10s remaining.' % (count, product)
>>> print(t)
0003 coconuts remaining.
```
Positional String Formatting

```python
>>> d = {'count': 3, 'product': 'coconuts'}
>>> s = '%%(count)d %%(product)s remaining.' % d
>>> print(s)
3 coconuts remaining.
>>> t = '%%(count)04d %%(product)-10s remaining.' % d
>>> print(t)
0003 coconuts remaining.
```
Advanced String Formatting

>>> fmt = '{count} {product} remaining.
>>> fmt.format(count=3, product='coconuts')
'3 coconuts remaining.'

>>> fmt = '{count:<04d} {product:>10s} remaining
>>> fmt.format(count=3, product='coconuts')
'0003 coconuts   remaining.'

http://www.python.org/dev/peps/pep-3101/
Advanced String Formatting

```python
>>> stock = {'count': 3,
...    'product': 'coconuts',
...    'stores': ['east', 'west']}
>>> fmt = '{s[product]} available in {s[stores][0]}.'
>>> fmt.format(s=stock)
'coconuts available in east.'

>>> import sys
>>> fmt = 'major: {mod.version_info.major}
... minor: {mod.version_info.minor}''
>>> print(fmt.format(mod=sys))
major: 3
minor: 2
```
def greet(*args, **kwargs):
    print('args is', args)
    print('kwargs is', kwargs)

greet('Weiner')
greet('Doe', 'John')
greet('Smith', salutation='Mr.')

args is ('Weiner',)
kwargs is {}
args is ('Doe', 'John')
kwargs is {}
args is ('Smith',)
kwargs is {'salutation': 'Mr.'}
Scoping

1. Local / function scope
2. Enclosing function(s)
3. Global / module scope (the .py file)
4. Builtins
Scoping

Where the Assignment Happens

\[ a = 1 \]

```python
import twisted
def quest():
class FavouriteColour(object):
```
Scoping

Choosing a Different Scope

```python
global vache
```

```python
nonlocal vache  # 3.x
```
Closures

```python
def fetchez_la_vache(v):
    """
    Return a vache-launching function.
    """
    def launch():
        v.moo = True
        v.crush_knight()
    return launch
```
Statements and Expressions

Statements:

```python
a = 1
if a:
    #...
assert a
print a  # 2.x only
```

Expressions:

```python
a == 1
2 / 4
a is b
9 < x < 13
"number thou shalt count" if a == 3 else "right out"
lambda z: z ** 2
```
While Loop

```python
parents = babies = 1
while babies < 100:
    print('This generation has %d babies' % babies)
    parents, babies = (babies, parents + babies)

This generation has 1 babies
This generation has 2 babies
This generation has 3 babies
This generation has 5 babies
This generation has 8 babies
This generation has 13 babies
This generation has 21 babies
This generation has 34 babies
This generation has 55 babies
This generation has 89 babies
```
Tuple Assignment

```python
a, b = 1, 2
a, b = b, a
a, (b, c) = (1, (2, (3, 4)))
```

Tuple unpacking in 3.x:

```python
*a, b = 1, 2, 3
*a, (b, *c) = (1, 2, (3, 4, 5))
```

Same code in 2.x:

```plaintext
z = 1, 2, 3
a, b = z[:-1], z[-1]

x = (1, 2, (3, 4, 5))
a, b, c = x[:-1], x[-1][0], x[-1][1:]
```
For Loop

```python
>>> people = ['john', 'pat', 'gary', 'michael']
>>> i = 0
>>> for name in people:
...     print("iteration %d -> %s" % (i, name))
...     i = i + 1
iteration 0 -> john
iteration 1 -> pat
iteration 2 -> gary
iteration 3 -> michael
```
For Loop with Unpacking

```python
>>> people = ['john', 'pat', 'gary', 'michael']
>>> for i, name in enumerate(people):
...    print("iteration \%d -> %s" % (i, name))
iteration 0 -> john
iteration 1 -> pat
iteration 2 -> gary
iteration 3 -> michael
```

```python
>>> enumerate(people)
<enumerate object at 0x7f502799aeb0>
>>> list(enumerate(people))
[(0, 'john'), (1, 'pat'), (2, 'gary'), (3, 'michael')]
```
The Fun Stuff
Generator Expression

```python
generator_expression = 

prices = {'apple': 0.40, 'banana': 0.50}
purchase = {'apple': 1, 'banana': 6}

line_items = (prices[f] * purchase[f] for f in purchase)
grocery_bill = sum(line_items)
print('I owe the grocer $%.2f' % grocery_bill)

I owe the grocer $3.40
```
List Comprehensions

Like generator expressions with [] instead of ()

```python
line_items = [prices[f] * purchase[f] for f in purchase]
```

same as

```python
line_items = list(prices[f] * purchase[f] for f in purchase)
```

same as

```python
line_items = []
for f in purchase:
    line_items.append(prices[f] * purchase[f])
```
Generator Functions

def fibonacci():
    a = b = 1
    while True:
        yield a
        a, b = b, a+b

for n in fibonacci():
    print(n)
    if n > 7: break

1 1
2 3
5 8
Generator Functions

```python
def fibonacci_until(top):
    a = b = 1
    while True:
        yield a
        if a > top: return
        a, b = b, a+b

for n in fibonacci_until(7):
    print(n)
```

1
1
2
3
5
8
Generators using Generators

```python
def double_it(seq):
    return (x * 2 for x in seq)

for n in double_it(fibonacci()):
    print(n)
    if n > 14: break
```

2
2
4
6
10
16
Generators as Iterators

```python
>>> f = fibonacci_until(2)
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
>>> next(f)
3
>>> next(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```
Generators as Coroutines

def fielder(data):
    pos = 0
    while True:
        end_pos = data.find('::', end_pos + 1)
        if end_pos == -1: break
        yield data[pos:end_pos]
        pos = end_pos + 1
    yield data[pos:]
```python
def fielder(data):
    end_pos = -1
    rewind = False
    while True:
        if rewind:
            if pos == 0:
                break
            end_pos = pos - 1
            pos = data.rfind(':', 0, end_pos) + 1
        else:
            if end_pos == len(data):
                break
            pos = end_pos + 1
            end_pos = data.find(':', end_pos + 1)
            if end_pos == -1: end_pos = len(data)
            rewind = yield data[pos:end_pos]
```
Generators as Coroutines

```python
>>> f = fielder("single:248:between:19:double:29:42")
>>> next(f)
'single'
>>> next(f)
'248'
>>> next(f)
'between'
>>> f.send(True)
'248'
>>> f.send(False)
'between'
>>> f.send(False)
'19'
```
Exception Handling

```python
import sys
try:
    total = sum(int(arg) for arg in sys.argv[1:])
except ValueError:
    print('Please supply integer arguments')
else:
    print('sum =', total)
finally:
    print('Goodbye!')
```
Fun with Finally

```python
def ones():
    try:
        while (True):
            yield 1
    finally:
        print('And finally, monsieur, a wafer-thin mint. ')

def ill_have_two():
    dinner = ones()
    print(next(dinner))
    print(next(dinner))

ill_have_two()
```

```
1
1
And finally, monsieur, a wafer-thin mint.
```
with open('cleaners.txt', 'wt', encoding='utf-8') as c:
    c.write('quit en masse, for some reason\n')

with shared_data_lock:
    modify_shared_data()
Function Decorators

```python
def show_enter_and_exit(fn):
    def inner(*args, **kwargs):
        print('entering')
        try:
            return fn(*args, **kwargs)
        finally:
            print('exiting')
    return inner

@show_enter_and_exit
def say_hello():
    print('hello')

say_hello()
```

entering
hello
exiting
Classes

class LineItem(object):
    
    """
    An item on my shopping list
    """
    def __init__(self, fruit, price, qty):
        self.fruit = fruit
        self.price = price
        self.qty = qty

    def total(self):
        return self.qty * self.price

    def __str__(self):
        return "%d x %s at %.2f each" % (self.qty, self.fruit, self.price)
Special Method Names

What's with all the underscores?

- don't clash with user attribute names
- indicate interaction with core language concepts
- allow operator overloading
- allow emulation of built-in types

- Collections
- Iterators
- Context Managers
- Functions
- Missing Attributes
- ...
An Instance

```python
>>> item = LineItem('orange', 0.75, 3)
>>> item.fruit
'orange'
>>> item.total()
2.25
>>> item.qty = 5
>>> item.total()
3.75
>>> print(item)
5 x orange at 0.75 each
```
Many Instances

```python
>>> cart = [LineItem(*columns) for columns in [
...     ('apple', 0.40, 1),
...     ('banana', 0.50, 6),
...     ('orange', 0.75, 3),]]
>>> for item in cart:
...     print(item)
1 x apple at 0.40 each
6 x banana at 0.50 each
3 x orange at 0.75 each
>>> print(sum(item.total() for item in cart))
5.65
```
Customizing Attribute Access

Get:

1. if there's a `__getattribute__(self, name)`, call it
2. look for the attribute in the instance, class and superclasses
3. if there's a `__getattr__(self, name)`, call it

Set/Delete:

1. if there's a `__setattr__(self, name, value)` or a `__delattr__(self, name)`, call it
2. set/delete the attribute in the instance
OO Features

• `classmethod`
  take `cls` as first argument instead of `self`
• `staticmethod`
  no `self` first argument
• `property`
  defines an attribute that can call a function when it is accessed, set and/or deleted
Descriptor Protocol

- `__get__(self, instance, owner)`
  - `owner` is the class
  - `instance` is the instance containing this attribute, or None for class attributes
- `__set__(self, instance, value)`
- `__delete__(self, instance)`
Descriptor Protocol

class Descr(object):
    def __get__(self, instance, owner):
        print(self, instance, owner)

class A(object):
    b = Descr()

a = A()
a.b
A.b

<Descr object at ...> <A object at ...> <class 'A'>
<Descr object at ...> None <class 'A'>
Questions