What's New in Python?

"Not your usual list of new features"

Stanford CSL Colloquium, October 29, 2003

Guido van Rossum

Elemental Security

guido@python.org
guido@elementalsecurity.com

Talk Overview

- About me
- About Python
- Case study 1: iterators and generators
- Case study 2: new classes and descriptors
- Question Time

About Me

- Age 4: first Lego kit
- Age 10: first electronics kit (with two transistors)
- Age 18: first computer program (on punched cards)
- Age 21: first girlfriend :-)
- 1982: "drs" math degree; joined CWI in Amsterdam
- 1987: first worldwide open source release
- 1989: started work on Python in spare time
- 1995: moved to Virginia, USA to join CNRI
- 2000: got married
- 2001: became a father
- 2003: moved to California to join Elemental Security

About Elemental Security

- Enterprise security software
- Early stage startup in stealth mode
- Using lots of Python
- We're hiring!
- See http://www.elementalsecurity.com

About Python

"The promotional package"

Executive Summary

- Dynamically typed object-oriented language
- Python programs look like executable pseudo-code
- Supports multiple paradigms:
  - procedural, object-oriented, some functional
- Extensible in C, C++, Fortran, ...
- Used by:
  - Google, ILM, NASA, Red Hat, RealNetworks, ...
- Written in portable ANSI C (mostly...)
- Runs on:
  - Unix, Windows, Mac, Palm, VxWorks, PlayStation 2, ...
- **Jython**: Java version, translates to Java byte code

Why Use Python?

- Dynamic languages are more productive
- Python code is more readable
- Python code is more maintainable
- Python has fast built-in very high-level data types
- Developer time is more expensive than CPU time

When Should You Not Use Python (Yet)?

- Things like packet filters, MP3 codecs, etc.
- Instead, write in C/C++ and wrap Python around it

Example Function

- ```
  def gcd(a, b):
      "Greatest common divisor of two integers"
      while b != 0:
          a, b = b, a%b
      return a
  ```

- Note:
  - no declarations
  - indentation+colon for statement grouping
  - doc string part of function syntax
  - parallel assignment (to swap a and b: "a, b = b, a")
  ```

Sample Use Areas

- Server-side web programming (CGI, app servers)
- Client-side web programming (HTML, HTTP, ...)
- XML processing (including XML-RPC and SOAP)
- Databases (Oracle, MySQL, PostgreSQL, ODBC, ...)
- GUI programming (Qt, GTK+, Tcl/Tk, wxPython, ...)
- Scientific/numeric computing (e.g. LLNL)
- Testing (popular area for Jython)
- Scripting Unix and Windows
- Rapid prototyping (e.g. at Google)
- Programming education (e.g. Oxford physics)
  - from middle school to college

Standard Library

- File I/O, socket I/O, web protocols (HTTP, CGI, ...)
- XML, HTML parsing (DOM, SAX, Expat)
- Regular expressions (using standard Perl re syntax)
- compression (gzip/zlib, bz2), archiving (zip, tar)
- math, random, checksums, algorithms, data types
- date/time/calendar
- threads, signals, low-level system calls
- Python introspection, profiling, debugging, testing
- email handling
- and much, much more!
  - and 10x more in 3rd party packages (e.g. databases)

Python Community

- Python is Open Source software; freely distributable
- Code is owned by Python Software Foundation
  - 501(c)(3) non-profit taking tax-deductible donations
  - merit-based closed membership (includes sponsors)
- License is BSD-ish (no "viral" GPL-like clause)
- Users meet:
  - on Usenet (comp.lang.python)
  - on IRC (#python at irc.freenode.net)
  - at local user groups (e.g. www.baypiggies.net)
  - at conferences (PyCon, EuroPython, OSCON)
- Website: www.python.org (downloads, docs, devel)

Python Development Process

- Nobody gets paid to work full-time on core Python
  - Though some folks get paid for some of their time
    - their employers use Python and need enhancements
- The development team never sleeps
  - For example, for the most recent release:
    - release manager in Australia
    - key contributors in UK and Germany
    - doc manager and Windows expert in Virginia
    - etc.
- Key tools: email, web, CVS, SourceForge trackers
  - IRC not so popular, due to the time zone differences

Python Enhancement Proposals (PEP)

- RFC-like documents proposing new or changed:
  - language features
  - library modules
  - even development processes
- Discussion usually starts in python-dev mailing list
- Wider community discussion on Usenet
- BDFL approval required to go forward
  - BDFL = "Benevolent Dictator For Life" (that's me :-)
  - this is not a democracy; let Python have *my* quirks
  - we don't want design by committee or majority rule
  - the PEP system ensures everybody gets *input* though

Python Release Philosophy

- "Major releases": 2.0 -> 2.1 -> 2.2 -> **2.3**
  - 12-18 month cycle
  - Focus on new features
  - Limited backward incompatibilities acceptable
    - usually requires deprecation in previous major release
- "Minor releases": e.g. 2.3 -> 2.3.1 -> **2.3.2**
  - 3-9 month cycle
  - Focus on stability; zero backward incompatibilities
  - One previous major release still maintained
- "Super release": 3.0 (a.k.a. Python 3000 :-)
  - Fix language design bugs (but nothing like Perl 6.0 :-)
  - Don't hold your breath (I'll need to take a sabbatical)

Case Study 1:
Iterators and Generators

"Loops generalized and turned inside out"

Evolution of the 'For' Loop

- Pascal: for i := 0 to 9 do ...
- C: for (i = 0; i < 10; i++) ...
- Python: for i in range(10): ...
- General form in Python:

  for <*variable*> in <*sequence*>:
      <*statements*>

- Q: What are the possibilities for <*sequence*>?

Evolution of Python's Sequence

- Oldest: *built-in* sequence types: list, tuple, string
  - indexed with integers 0, 1, 2, ... through len(seq)-1
    - for c in "hello world": print c
- Soon after: *user-defined* sequence types
  - class defining __len__(self) and __getitem__(self, i)
- Later: lazy sequences: *indeterminate length*
  - change to for loop: try 0, 1, 2, ... until IndexError
- Result: *pseudo-sequences* became popular
  - these work only in for-loop, not for random access

Python 1.0 For Loop Semantics

- for <*variable*> in <*sequence*>:
    <*statements*>
- Equivalent to:
- seq = <*sequence*>
ind = 0
while ind < len(seq):
    <*variable*> = seq[ind]
    <*statements*>
    ind = ind + 1

Python 1.1...2.1 For Loop Semantics

- for <*variable*> in <*sequence*>:
    <*statements*>
- Equivalent to:
-
```
seq = <sequence>
ind = 0
while True:
    try:
        <variable> = seq[ind]
    except IndexError:
        break
    <statements>
    ind = ind + 1
```

Example Pseudo-Sequence

- class FileSeq:

```
    def __init__(self, filename):       # constructor
        self.fp = open(filename, "r")

    def __getitem__(self, i):           # i is ignored
        line = self.fp.readline()
        if line == "":
            raise IndexError
        else:
            return line.rstrip("\n")
```

- for line in FileSeq("/etc/passwd"):

```
    print line
```

Problems With Pseudo-Sequences

- The __getitem__ method invites to random access
  - which doesn't work of course
  - class authors feel guilty about this
    - and attempt to make it work via buffering
    - or raise errors upon out-of-sequence access
    - both of which waste resources

- The for loop wastes time
  - passing an argument to __getitem__ that isn't used
  - producing successive integer objects 0, 1, 2, ...
    - (yes, Python's integers are real objects)
      - (no, encoding small integers as pseudo-pointers isn't faster)
        - » (no, I haven't actually tried this, but it was a nightmare in ABC)

Solution: The Iterator Protocol (2.2)

- for <*variable*> in <*iterable*>:
      <*statements*>
- Equivalent to:
- ```
  it = iter(<iterable>)
  while True:
      try:
          <variable> = it.next()
      except StopIteration:
          break
      <statements>
      # There's no index to increment!
  ```
  ```

Iterator Protocol Design

- Many alternatives were considered and rejected
- Can't use sentinel value (list can contain any value)
- while it.more():
  <*variable*> = it.next()
  <*statements*>
  - Two calls are twice as expensive as one
    - catching an exception is much cheaper than a call
  - May require buffering next value in iterator object
- while True:
  (more, <*variable*>) = it.next()
  if not more: break
  <*statements*>
  - Tuple pack+unpack is more expensive than exception

Iterator FAQ

- Q: Why isn't next() a method on <*iterable*>?

  A: So you can nest loops over the same <*iterable*>.
- Q: Is this faster than the old way?

  A: You bet! Looping over a builtin list is 33% faster. This is because the index is now a C int.
- Q: Are there incompatibilities?

  A: No. If <*iterable*> doesn't support the iterator protocol natively, a wrapper is created that calls __getitem__ just like before.
- Q: Are there new possibilities?

  A: You bet! dict and file iterators, and generators.

Dictionary Iterators

- To loop over all keys in a dictionary in Python 2.1:
  - ```
    for key in d.keys():
        print key, "->", d[key]
    ```
- The same loop in Python 2.2:
  - ```
    for key in d:
        print key, "->", d[key]
    ```
- Savings: the 2.1 version copies the keys into a list
- Downside: can't mutate the dictionary while looping
- Additional benefit: you can now write "if x in d:" too instead of "if d.has_key(x):"
- Other dictionary iterators:
  - d.iterkeys(), d.itervalues(), d.iteritems()

File Iterators

- To loop over all lines of a file in Python 2.1:
  - ```
    line = fp.readline()
    while line:
        <statements>
        line = fp.readline()
    ```
- And in Python 2.2:
  - ```
    for line in fp:
        <statements>
    ```
  - 40% faster than the 'while' loop
    - (which itself is 10% faster compared to Python 2.1)
    - most of the savings due to streamlined buffering
    - using iterators cuts down on overhead *and* looks better

Generator Functions

- Remember coroutines?
- Or, think of a parser and a tokenizer:
  - the parser would like to sit in a loop and occasionally ask the tokenizer for the next token...
  - but the tokenizer would like to sit in a loop and occasionally give the parser the next token
- How can we make both sides happy?
  - threads are way too expensive to solve this!
- Traditionally, one of the loops is coded "inside-out" (turned into a state machine):
  - code is often hard to understand (feels "inside-out")
  - saving and restoring state can be expensive

Two Communicating Loops

- Generator functions let you write *both* sides (consumer *and* producer) as a loop, for example:

  - ```
    def tokenizer():                # producer (a generator)
        while True:
            ...
            yield token
            ...
    ```

  - ```
    def parser(tokenStream):  # consumer
        while True:
            ...
            token = tokenStream.next()
            ...
    ```

Joining Consumer and Producer

- tokenStream = tokenizer(); parser(tokenStream)

- The presence of *yield* makes a function a generator
- The tokenStream object is an *iterator*
- The generator's stack frame is prepared, but it is *suspended* after storing the arguments
- Each time its next() is called, the generator is *resumed* and allowed to run until the next *yield*
- The caller is *suspended (*that's what a call does!)
- The yielded value is returned by next()
- If the generator *returns*, next() raises StopIteration
- "You're not supposed to understand this"

Back To Planet Earth

- Generator functions are useful iterator filters
- Example: double items: A B C D -> A A B B C C D D
  - ```
    def double(it):
        while True:
            item = it.next()
            yield item
            yield item
    ```
- Example: only even items: A B C D E F -> A C E
  - ```
    def even(it):
            while True:
                yield it.next()
                xx = it.next()          # thrown away
    ```
- Termination: StopIteration exception passed thru

Generators in the Standard Library

- tokenize module (a tokenizer for Python code)
  - old API required user to define a callback function to handle each token as it was recognized
  - new API is a generator that yields each token as it is recognized; much easier to use
  - program transformation was trivial:
    - replaced each call to "callback(token)" with "yield token"
- difflib module (a generalized diff library)
  - uses yield extensively to avoid incarnating long lists
- os.walk() (directory tree walker)
  - generates all directories reachable from given root
  - replaces os.path.walk() which required a callback

Stop Press! New Feature Spotted!

- Consider list comprehensions:
  - [x**2 for x in range(5)] -> [0, 1, 4, 9, 16]

- Python 2.4 will have generator expressions:
  - (x**2 for x in range(5)) -> "iter([0, 1, 4, 9, 16])"

- Why is this cool?
  - sum(x**2 for x in range(5)) -> 30
    - computes the sum without creating a list
    - hence faster
  - can use infinite generators (if accumulator truncates)

Case Study 2: Descriptors

"Less dangerous than metaclasses"

Bound and Unbound Methods

- As you may know, Python requires 'self' as the first argument to method definitions:
  - ```
    class C:                    # define a class...
        def meth(self, arg):    # ...which defines a method
            print arg**2
    ```
  - `x = C()` # create an instance...
  - `x.meth(5)` # ...and call its method
- A lot goes on behind the scenes...
- **NB:** classes and methods are runtime objects!
    ```

Method Definition Time

- A method defined like this:
  - `def meth(self, arg):`
    `...`
- is *really* just a function of two arguments
- You can play tricks with this:

```
def f(a, b):          # function of two arguments
    print b
class C:              # define an empty class
    pass
x = C()               # create an instance of the class
C.f = f               # put the function in the class
x.f(42)               # and voila! magic :-)
```

Method Call Time

- The magic happens at method call time
- Actually, mostly at method *lookup* time
  - these are not the same, you can separate them:
    - "xf = x.f; xf(42)" does the same as "x.f(42)"
    - "x.f" is the lookup and "xf(42)" is the call
- If x is an instance of C, "x.f" is an *attribute lookup*
  - this looks in x's *instance variable dict* (x.__dict__)
  - then in C's *class variable dict* (C.__dict__)
  - then searches C's base classes (if any), etc.
- *Magic happens* if:
  - f is found in a class (not instance) dict, *and*
  - what is found is a *Python function*

Binding a Function To an Instance

- Recap:
  - we're doing a lookup of x.f, where x is a C instance
  - we've found a function f in C.__dict__
- The value of x.f is a *bound method object*, xf:
  - xf holds references to instance x and function f
  - when xf is called with arguments (y, z, ...), xf turns around and calls f(x, y, z, ...)
- This object is called a bound method
  - it can be passed around, renamed, etc. like any object
  - it can be called as often as you want
  - yes, this is a currying primitive! xf == "curry(x, f)"

Magic Is Bad!

- Why should Python functions be treated special?
- Why should they *always* be treated special?

Magic Revealed: Descriptors

- In Python 2.2, the class machinery was redesigned to unify (user-defined) classes with (built-in) types
  - The old machinery is still kept around too (until 3.0)
  - To define a new-style class, write "class C(object): ..."
- Instead of "if it's a function, do this magic dance", the new machinery asks itself:
  - if it supports the descriptor protocol, invoke that
- The descriptor protocol is a method named __get__
- __get__ on a function returns a bound method

Putting Descriptors To Work

- Static methods (that don't bind to an instance)
  - a wrapper around a function whose __get__ returns the function unchanged (and hence unbound)
- Class methods (that bind to the class instead)
  - returns curry(f, C) instead of curry(f, x)
    - to do this, __get__ takes *three* arguments: (f, x, C)
- Properties (computed attributes done right)
  - __get__ returns f(x) rather than curry(f, x)
  - __set__ method invoked by *attribute assignment*
  - __delete__ method invoked by *attribute deletion*
  - (__set__, __delete__ map to different functions)

Properties in Practice

- If you take *one* thing away from this talk,
  it should be how to create simple properties:

  - ```
    class C(object):                # new-style class!
        __x = 0                     # private variable

        def getx(self):             # getter function
            return self.__x

        def setx(self, newx):       # setter function
            if newx < 0:            # guard
                raise ValueError
            self.__x = newx

        x = property(getx, setx)    # property definition
    ```
    ```

Useful Standard Descriptors

- Static methods:
  - class C(object):
```
class C(object):
    def foo(a, b):          # called without instance
        ...
    foo = staticmethod(foo)
```

- Class methods:
```
class C(object):
    def bar(cls, a, b):     # called with class
        ...
    bar = classmethod(bar)
```

- See: http://www.python.org/2.2.3/descrintro.html

A Schizophrenic Property

- Challenge: define a descriptor which acts as a class method when called on the class (C.f) and as an instance method when called on an instance (C().f)
  - class SchizoProp(object):

```
def __init__(self, classmethod, instmethod):
    self.classmethod = classmethod
    self.instmethod = instmethod

def __get__(self, obj, cls):
    if obj is None:
        return curry(self.classmethod, cls)
    else:
        return curry(self.instmethod, obj)
```

- Do Not Try This At Home! :-)

Question Time

"If there's any time left :-)"