"Nobody Expects the Spanish Inquisition"

An Overview of the Python Programming Language

Guido van Rossum
CWI
(Centre for Mathematics and Computer Science)
Amsterdam, The Netherlands

<guido@cwi.nl>

currently:
Invited to NIST as a Guest Researcher in the EEEL
<guido@eeel.nist.gov>

Today's Presentation...

...will dwell on the following topics:

- Hello, World!
- History
- The Design
- Doubts
- The Crystal Ball

Hello, World!

- Object-Oriented, Interpreted
- Dynamic Typing, Static Scoping
- Extensible, Embeddable

Python encourages:

- Readable code
- Reusable code

Python supports:

- Interactive use, Scripting, Prototyping
- Medium-to-large program development

Variables and Expressions...

...Look much like in C:

```
pi = 3.14159
root = (-b + sqrt(b*b - 4*a*c)) / (2*a)
```

Strings are basic data types and can be "sliced":

```
hello = "hello "
world = "world"
w = world[0]        # => 'w'
d = world[-1]       # => 'd'
hw = hello + world  # => 'hello world'
lowo = hw[3:-3]     # => 'lo wo'
len(lowo)           # => 5
```

Other Data Types

```
primes = [2, 3, 5]
primes.append(7)
primes = primes + [9, 11, 13]
primes.remove(9)    # oops!
print primes       # => [2, 3, 5, 7, 11, 13]

database = {}
database['guido'] = 4127
database['jack'] = 4099
database['mclay'] = 4099
del database['jack']
database['guido'] = 3695
database['klm'] = 3539
print database
# => {'guido': 3695, 'mclay': 4099, 'klm': 3539}
```


Flow Control

```
if -10 <= hit <= 10:
    if -1 <= hit <= 1: print 'real close'
    else: print 'close'
elif hit < 0:
    print 'far left'
else:
    print 'far right'

while x > 0 and abs(lastx-x) > FUZZ:
    lastx, x = x, (x + a/x)/2.0

for p in range(2, n):  # i in [2, 3, 4, ..., n-1]
    if i%p == 0:
        break
# note use of indentation for grouping
```


Functions

```
def factorial(n):
    result = 1
    while n > 1:
        result = result * n
        n = n - 1
    return result
```

- 'def' is an executable statement
- No argument type declarations
- Assignment is local by default
  - 'global' statement overrides

Classes

```
class Set:
    def __init__(self): self.values = {}
    def add(self, x): self.values[x] = x
    def remove(self, x):
        if self.contains(x): del self.values[x]
    def contains(self, x):
        return self.values.has_key(x)
```

- '__init__' defines constructor; 'x = Set()' calls it
- Explicit 'self' required in method definitions
- Supports multiple inheritance
- Supports operator overloading, e.g. '__add__'
- All methods are 'virtual'

Modules

```
import os, sys
import getopt
from socket import *
from math import sin, cos, pi
```

- Modules can be written in Python or built-in
- First import initializes module
- Later imports are one dictionary lookup
- Pyton module initialization executes the module text as a script
- Each module has its own global name space
- ’from’ bind local names – no alias magic

Exceptions

```
try:
    fp = open(filename, ’r’)
    while 1:
        line = fp.readline()
        if not line: break
        if line[-1] != ’\n’:
            raise IOError, ’incomplete line’
        if interesting(line): print line
    fp.close()
except IOError, message:
    print ’*** I/O Error:’, filename, message
    sys.exit(1)
```

- Also ’try’ ... ’finally’ ... (guaranteed clean-up)

Accurate Diagnostics

- Precise error location
- Stack frame trace

```
Traceback (innermost last):
  File "clock.py", line 204
    main()
  ...
  File "clock.py", line 92, in settimer
    now = getlocaltime()
  File "clock.py", line 202, in getlocaltime
    return int(time.time() - TZDIFF)
OverflowError: float to large to convert
```

- Debugger, profiler written in Python

Ever-growing Library

Perhaps Python’s real strength lies in its library...

- String operations, regular expressions
- O/S interfaces: POSIX, sockets, select
- Math library
- Graphics and User Interface libraries (e.g. X11, SGI GL, STDWIN, Tk)
- Internet, World-Wide Web
- Multimedia data types
- Threads (on selected O/S)
- Introspective capabilities (e.g. debugger, profiler, bytecode tools)

History

- Conception Xmas holidays 1989
- First Internet release February 1991
- Language remains relatively stable
- Updates every 3-6 months

Important Influences

ABC

- The ideal teaching language, yet powerful
- Alienated experienced computer users

Amoeba distributed system

- Clean client/server model, almost O-O
- Ubiquitous error handling a chore

Modula-3

- Modules
- Exceptions

A Look At Python's Design

- Conventional Syntax
- High Level Data Types
- Object-oriented Semantics
- Scopes
- Name Spaces

Conventional Syntax

Fairly conventional syntax:

- Minimalist
- No tricks required
- Little training needed
- Mostly used as a second language

For compactness:

- indentation for statement grouping
- no type declarations

High Level Data Types

Small number of powerful data types can be combined in any way (orthogonally):

- Numbers (int, arbitrary precision int, float)
- “None” (not any other type)
- Sequences (strings, tuples, lists)
- Mappings (associative arrays or hash tables)

Object-oriented Semantics

Object-oriented all the way down:

- Numbers, characters, bytes are objects
- Strings, lists, associative arrays are objects
- Functions, modules, classes, methods are objects
- (as are compiled code blocks, stack frames, tracebacks, and everything else the debugger needs...)
- User-defined classes can mimic behavior of built-in types (operator overloading)

Scopes

Two-and-a-half scopes:

- Local (within current function)
- Global (within current module)
- Built-in names (read-only)

Name assignment goes into local scope

- use `global` declaration to override

Name use tries local, global, built-in scopes in turn

- static analysis speeds up use of locals

Name Spaces

Unified name spaces contain:

- Imported objects
  (usually modules, classes, or functions)
- Locally defined objects
  (usually classes, functions, and variables)

Concluding

Python’s design is:

- Compact
- Consistent
- Orthogonal
- Open

Obtained by:

- Standing on the shoulders of Giants
- Listening to users
- Being a user!

Doubts About the Language

Maybe Python is too dynamic:

- static checks, compilation, optimization are hard/impossible

User problems:

- Two-scope system causes confusion (lambda!)
- Indentation generates flames
- No inheritance from built-in types
- Library varies in quality

Implementation Doubts

- Reference count interface inconsistencies
- Doesn’t reclaim circular references
- Uses (some) global variables
- No inheritance from built-in types
- Inefficient parser
- Embeddability as an afterthought
- No naming conventions (yet)

The Crystal Ball

- Python Workshop
- Working on Python at NIST
- A book or two on Python
- More commercial use
- Improvements to the implementation
  - C/C++ API naming conventions
  - see workshop topics

A Python Workshop

Next week at NIST (booked full). Topics:

- Interfacing to C++
- A standard GUI API?
- Persistent Objects
- Safe-Python
- Static Checks, Optimization, Compilation
- Software Management
- Python and the World-Wide Web
- **Python Steering Committee**
  (decide which direction Python will take)

Probably another workshop in late spring

A word from our Sponsor...

Author is currently visiting NIST as a Guest Researcher in the Electronics and Electrical Engineering Laboratory (EEEL)

- Python's features make it attractive for enterprise integration and engineering tasks
- Python is used by several projects in EEEL as a World Wide Web CGI language
- Some specific enhancements would make it more suitable for electrical and electronics engineering

Python Use at NIST

- World Wide Web access to databases:
  - Electronic Commerce of Component Information (ECCI) project
  - Prototype of a NIST storeroom catalog

Likely also:

- In the SEMATECH Computer Integrated Manufacturing (CIM) Framework
  - As a language for executable specifications
  - As an equipment control language

The MMACE Research and Engineering Framework

MMACE = Tri-services Microwave and Millimeter-Wave Advanced Computational Environments project

- NIST and MMACE are testing Python for use as a core element of a Research and Engineering Framework for high frequency electronics
- Python's modularity should make the MMACE Framework useful outside of the microwave tube industry

Priorities for a R+E Framework

General requirements for commercial use:

- "Safe Python"
- A Standard Python GUI API
- Object Persistence

Requirements for use by engineers:

- An Engineering Units Module
- 2D+3D Graph Plotting

Other requirements:

- Globalization