"*Using Python*": a Book Preview

May 13, 1995

Mark Lutz
Boulder, Colorado
lutz@kapre.com
(303) 546-8848 [work]
(303) 684-9565 [home]

INTRODUCTION.

This paper is a brief overview of the upcoming Python O'Reilly book I'm writing. I received a formal contract for the book in April. Roughly, the book will be some 400 pages, and should be ready (at least in beta form) by fall, 1995.

Although we've spent quite a bit of time developing the current outline, everything mentioned here should be considered '*pre-alpha*' material at best. In particular, the outline is prone to frequent change (it will probably change by the time you read this!), and my editor and reviewers will have a big influence on the book's final structure. So please take all this with a reasonably large 'grain-of-salt'.

Some of what gets covered in the book is still open to input; in fact, this is one reason I'm making this information public right now. If you have suggestions, I'd be happy to listen. I may also be asking for comments from experts in some of the domains the book addresses.

THE BASIC APPROACH.

The goal of the book will be to show how Python can be used for *real* software development tasks. It's not intended to be an exhaustive reference for every detail of the language. Instead, we want to show how Python can be *applied* to typical tasks faced by developers. Along the way, we'll present Python's major features, and show what it is about the language that makes it such a great tool.

In other words, this will be a *practical* book for engineers, not a programming-language *theory* book. Of course, some theory will be required, but that won't be the tone of the book. Some components of the approach:

- Organized by *common roles*, not language features
- Based on *realistic examples*, and incremental improvements
- A very *informal* presentation (this is an O'Reilly book!)
- A *practical* book: not just a list of syntax/semantic rules
- Details on a *need-to-know* basis (to understand the examples)
- Almost a "*Python-by-example*" book (but not really...)

Some of these goals may turn out to be difficult. For instance, "realistic" examples of ILU use, or WWW scripting might be too complex or large for this book. Still, we'll try to avoid using overly-artificial code samples, whenever possible.

THE BOOK'S STRUCTURE.

Essentially, the book's structure just reflects the approach we're taking. Because the book's goal is to present Python by its roles (not by its syntax or semantics), its structure reflects ways that Python can be used in practice. Here's some general notes; the outline below gives specific details.

- One section for each of Python's *major roles*
- Realistic *examples* of *Python-at-work* for each role
- *Language details* presented in the context of real examples
- A *comprehensive index* serves as a language reference
- *Appendices* for the most-common reference material
- An *optional tutorial* introduces basic language issues

THE (CURRENT) OUTLINE.

The outline (as presented to O'Reilly editors, more or less) appears below. The first 2 sections (shell and extension-language programming) are probably the most critical, and the longest, since they'll also cover much of the Python language, as an incidental topic. The parts following these 2 are roughly selected popular Python application areas. Note: the outlines of parts not yet written are still very incomplete.

Summary...

Details...

FOREWORD.
Guido; he'll also be a major reviewer for the book (in all his copious spare time :-).

PREFACE.
The book's structure, about the examples (run on Linux), acknowledgments, etc.

1. INTRODUCTION (15 pages).

Already written. A short description of Python's roles, what people like about it, and a sneak-preview of some examples we'll look at in the book. Here's the current subsections of the introduction:

- Introduction
- So what's Python?
- What's all the excitement about?
- Some of Python's features
- What's Python good for?
- A "Sneak Preview"
    - Data structures
    - System administration
    - GUI programming
    - Extension-language work
- Conclusion

2?. A PYTHON TUTORIAL (50 pages; here, or an appendix).

Already written. It's not yet clear whether the tutorial will appear first (here) or as an appendix at the end (as optional reading for beginners). There's different schools-of-thought on this; my current idea is to put this first, to avoid cluttering up the 'role' chapters that follow. On the other hand, it uses artificially small examples to illustrate language features, and may move too slow as is. Stay tuned... ;-)

- Introduction

- Getting started
    - Interactive use
    - Simple expressions and assignment

- Some useful statements
    - The 'if' selection
    - Syntax rules
    - The 'while' loop
    - The 'for' loop, part 1...

- Using built-in data structures
    - Lists
    - range()
    - Dictionaries
    - Tuples
    - Conversions
    - General features
    - Nesting
    - Object references
    - Generic operations
    - Slicing
    - Growing and shrinking

- Using Functions
    - Basic concepts
    - Functions are objects
    - Arguments
    - Recursion

  - Modular design
  - Global and local variables

- Using Modules
  - Basic concepts
  - Running code in modules
  - Modules are name-spaces
  - Built-in modules
  - Reloading modules
  - Module export rules
  - Compiled modules

- Using classes
  - Basic concepts
  - Classes are name-spaces too
  - Instances versus classes
  - Inheritance, polymorphism, and encapsulation
  - Constructors and destructors
  - Subclass specialization
  - Operator overloading
  - Inheritance versus composition
  - Metaclass protocol methods
  - Implementing 'records'
  - The stack class
  - Name scoping rules
  - Program organization

- Exception handling
  - The 'try' and 'raise' statements
  - System exceptions
  - Nesting exception handlers

- Extending and embedding in C
  - C extension module format
  - Running embedded strings
  - Wrapping C++ classes
  - Other integration ideas

- Using built-in tools
  - Python system structure
  - Pre-coded functionality: functions, types, modules

- Summary (each subsection has a summary too)
- Conclusion

3. SHELL PROGRAMMING (50 pages).

In progress. This section uses shell-scripts to get started, and show basic concepts. File processing, and system-component access appears in the first example we look at; these are critical topics to the target audience. The section starts with very simple tasks, and develops them into more sophisticated programs.

We start with a 6-line text-file packer; eventually, we wind up with a simple application class framework to encapsulate top-level components. Along the way, we introduce language-features needed to understand the examples. Composed of multiple chapters:

- Introduction

- Basic system administration
  - 'Quick-and-dirty' file packing scripts
  - Improving on the pack scripts: step 1
  - Improving on the pack scripts: a functional API
  - A 'quick-and-dirty' mail-file scanner

- Wrapping top-level components
  - The App class
  - Adding i/o stream redirection
  - Adding internal (string) files
  - Adding interactive loops and menus
  - Mixing multiple Application classes
  - Testing the framework
  - A menu-based application

- pack, unpack, and mail-tool scripts as application objects
- Optimizing the mail-tool by subclassing
- Adding file backups to unpack
- An interactive interface for our shell tools

- Summary/Conclusion

Complex topics, such as multiple inheritance, will appear in the context of real program needs: to derive a subclass that supports both *stdio* redirection, and a command menu. We'll also introduce the *debugger*, using intentional errors. Basic *profiler* use comes up here too: we improve on a mail-tool class's performance be a factor of 3. More details on this section appear below.

4. EXTENSION-LANGUAGE PROGRAMMING (50 pages).

An example of using Python as an embedded language, for on-site customization of a C/C++ libraries and binaries. Develops basic embedded call functions, and shows extension modules and types by example. This section won't be an exhaustive reference for all run-time functions; the goal is just to show some ways
that embedding/extending can be done. The section's example problems make use of both C->Python (embedding) and Python->C (extending) interfaces. Its current parts:

- Introduction
- The problem: on-site configurations

- Running embedded Python code
  - As strings: expressions and statements
  - As files: by module/object names
  - As registered callable objects
  - As object methods

  - As UNIX scripts
  - Conversions: sending inputs, getting results
  - Error-handling
  - Debugging and reloading embedded code
  - An embedded-call API
  - A Python object API

- Calling C extensions from Python
  - Extension modules
  - Extension data-types
  - Wrapping C++ classes
  - Conversions: sending inputs, getting results
  - Error-handling
  - Code generation for integrating API's ('bgen')

- Building with Python
  - Linking Python into an application
  - Linking extensions into Python
  - Dynamic and static binding
  - Shipping Python code: source, compiled, frozen

- Conclusion

This section might also present Guido's new '*bgen*' tool, which generates code needed to integrate a C API/library into Python, by scanning header files. It may also build some higher-level embedded-call functions (on top of existing low-level interfaces), unless these become a standard part of Python in the next release.

5. GUI PROGRAMMING (30 pages) (might be moved to part 4).

Using Tkinter: Python's object-oriented interface to TCL/Tk. Looks at some simple GUI programs ('hello world', a simple mail-file browser), but defers to TCL/Tk documentation for more information. This section might be moved ahead to part 4 (between scripts and embedding); it will have a powerful visual impact.

- GUI options: Tk, X, MFC, Pesto/Fresco, portability,...

- The '*Hello world*' program in Tkinter
  - Basic use: widgets as functions
  - Adding event callback handlers
  - Attaching widgets to frames
  - Making GUI's by subclassing
  - Pop-up dialog boxes
  - Menus and tool-bars
  - Spawning new windows and processes

- A simple text editor
- A GUI front-end for mail-tool (part 3)
- Using GUI tools in embedded code (part 4)
- GuiApp: a subclass of the application framework (part 3)

- RAD: changing the GUI without stopping or rebuilding
- Tkinter as an extension example (part 4)
- Optional: a GUI front-end for our shell tools (part 3)

Tkinter will be used in later chapters too. Rapid GUI development is fast becoming one of Python's strongest points. Unfortunately (fortunately?), there's a lot of options right now; since Tk seems to be dominant in the UNIX public-domain arena (O'Reilly's market), it will probably get the most exposure. But I'm open to other opinions.

6. DATA-STRUCTURE PROGRAMMING (30 pages).

Develops a data-structure class library, to demonstrate more OOP in Python. The simple framework will handle sets, and will grow to include trees, graphs, etc. Along the way, we'll look at the profiler, for optimizing sets.

- Set-processing functions
- A simple set class (with a generic superclass..)
- Handling sequences generically: reverse, permute,..
- Optimizing set performance using dictionaries
- Adding trees and graphs

- The RelSet subclass: adding relational algebra
- The GuiSet subclass: adding GUI display protocol
- The FileSet subclass: random access files as sets

- Making objects persistent: 'pickle' and 'shelve'
- SQL queries in Python
- Optional: Database interfaces

Also looks at persistent objects (a new feature of Python); if pressed for space, this is the most important topic here. May also discuss other Python database interfaces: dbm, oracle, sybase, etc. (optional).

7. WWW/INTERNET SCRIPTING TOOLS (30 pages).

Python 1.2 has a set of WWW tools bundled with it, as a standard part of the language. Given WWW's popularity, the book will devote a whole part to Python's WWW connection. Under construction...

8. LANGUAGE-BASED TOOLS (30 pages).

Builds a calculator program, to show Python as a stand-alone language. Also looks at a command-line interpreter, etc.

- 'calc': a simple calculator program (parser/evaluator)
- Adding an unparser
- Adding an interpreter: class instance trees
- A GUI front-end for 'calc'
- Moving the scanner to C

- Embedded Python + API's: versus home-grown languages
- Optional: a command-line interpreter ('psh')
- Optional: the 'gluegen' parser
- Optional: the Python parser module
- Optional: the regex/regsub modules

9. CLIENT/SERVER PROGRAMMING (30 pages).

Also known as '*distributed computing/objects*'. Looks at pipes, threads, sockets. May also build up a simple 'visual python' environment, using modules developed in earlier chapters (a python server process, a GUI client, text editor...), to tie-together some earlier concepts into a complete system.

- Forking new Python processes: fork, pipes, sockets
- The 'calc' program as client/server system (a GUI client)
- ILU: language/machine-independent interfaces
- Optional: A "Visual Python" environment

This section will also give a brief overview of the Python/ILU system: roughly, a public-domain CORBA implementation, which supports Python, Modula-3, and Common-Lisp. This is becoming a popular client/server medium.

10. KNOWLEDGE-BASED PROGRAMMING (30 pages).

Very optional. Presents a simplified version of an expert-system shell I developed ('Holmes'), as an example of stand-alone Python programming. Includes a simple forward and backward chaining inference engine, simplified pattern matcher, and a rule index-tree class. Very likely to be cut; I only include it in case I wind up with some free time/space at the end.

11. CONCLUSION (10 pages).

More on Python's roles in the development cycle, now that we've shown how to use it. Hybrid systems, the development bottleneck, the *Gilligan Factor*,...

12. APPENDICES (50 pages).

The goal here isn't to include a reference manual: that's what the index (and Python's standard documentation) is for. Instead, we'll only include some of the more critical information not presented elsewhere in the text.

- A Python tutorial (if not earlier; see details above)
- Common programming mistakes (from questions on the net)
- Common library (built-in) functions, modules, and exceptions
- Syntax definition (charts?)
- Some platform-dependencies (but not build details)
- A "peek under the hood": some design/internals notes
- Sources for more information (news, mail, ftp, WWW, PSA, etc.)
- Where to get the examples in the book
- A quick-reference (a card?)
- Index

Since the index serves as a reference, it must be complete. See above: the tutorial might appear earlier.

MORE ABOUT THE SHELL PROGRAMMING SECTION.

This section opens with a real-life scenario: packing/unpacking text files for transfer over a modem. The examples are based on scripts I actually needed to write from home; the mail-tool also represents a real-life need. The 'pack' program begins as a simple 6-line script:

```
#!/usr/local/bin/python

import sys
marker = '::::::'

for name in sys.argv[1:]:
    input = open(name, 'r')
    print marker + name
    print input.read()
```

which we'll look at line-by-line, since this is the first Python code in the book. 'pack' later evolves to include a functional API, for use from other modules:

```
#!/usr/local/bin/python
# use: pack_all('target', ['src', 'src'])
# use: % pack src src ...
# use: % pack -b target src src ...


from sys import *
from textpack import marker


def pack_file(name, output):
    input = open(name, 'r')
    output.write(marker + name + '\n')
    while 1:
        line = input.readline()
        if not line: break
        output.write(line)


def pack_all(output, sources):
    if type(output) == type("string"):
        output = open(output, 'w')
    for name in sources:
        try:
            print 'packing:', name
            pack_file(name, output)
        except:
            print 'error processing:', name;  exit(1)


if __name__ == '__main__':
    try:
        if len(argv) >= 4 and argv[1] == '-b':
            pack_all(argv[2], argv[3:])
        else:
            pack_all(stdout, argv[1:])
    except:
        print 'error opening output'; exit(1)
```

and finally, becomes an instance of the 'App' application class, inheriting input/output stream redirection, command line processing, etc., after we've looked at the framework:

```
#!/usr/local/bin/python
# use: app.appCall(PackApp, args...)
# use: % packapp.py -o target src src ...


from app import App
from textpack import marker


class PackApp(App):
    def start(self):
        if not self.args:
            self.exit('packapp.py [-o target]? src src...')
        else:
            self.setOutput()

    def run(self):
        for name in self.restargs():
            try:
                self.message('packing: ' + name)
                self.pack_file(name)
            except:
                self.exit('error processing:' + name)

    def pack_file(self, name):
        self.setInput(name)
        self.write(marker + name + '\n')
        while 1:
            line = self.readline()
            if not line: break
            self.write(line)


if __name__ == '__main__':  PackApp().main()
```

Along the way, each step in pack's evolution will improve its functionality in some way. Before this part is over, we'll have introduced much of the Python language, indirectly. New language features in the examples will be discussed as needed. For example, object-oriented-programming comes up, as a means of wrapping top-level components.

The 'unpack' and 'mail-tool' scripts similarly evolve. For instance, 'unpack' starts out simple:

```
#!/usr/local/bin/python

from sys import *
marker = '::::::'

for line in stdin.readlines():
    if line[:6] != marker:
        print line[:-1]
    else:
        stdout = open(line[6:-1], 'w')
```

and then moves to a variant that doesn't have to load the whole packed file into memory all at once:

```
#!/usr/local/bin/python

import sys
marker = ':::::'

while 1:
    line = sys.stdin.readline()
    if not line:
        break
    elif line[:6] != marker:
        print line[:-1]
    else:
        sys.stdout = open(line[6:-1], 'w')
```

and finally to an 'App' subclass. 'mail-tool' is only presented as a simple script, and then as an 'App' instance. Mail-tool extracts messages from a mail-box file, based on simple header-line matching. Other book examples are available on request; they'll all be ftp'able eventually.