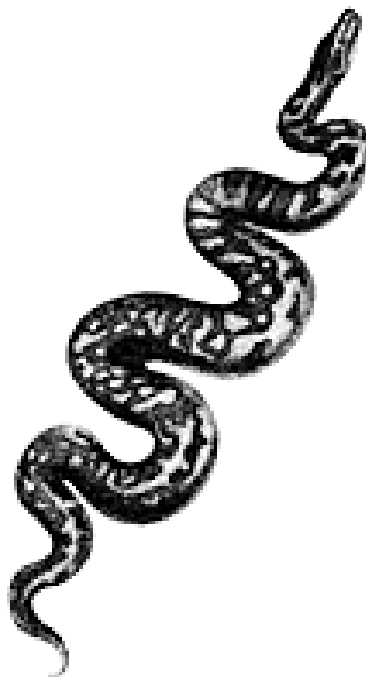


Learning Python



Student Course Workbook

October, 2006
HTML Workbook Version 2.5.0
Copyright © Mark Lutz, 1997—2006

Contents

This is the course workbook root page. It contains all the material presented during the class, source code for examples and lab exercises, and links to related information on the Net. Click on unit titles below to go to lecture unit pages. Click on the exercises title at the bottom of the list below to go to lab exercises, or use the links at the end of each lecture unit's page. Use your browser's "back" button to come back to this index, or create a shortcut to this file on your machine. See also the CD-ROM distribution's ["readme"](#) file.

The *paper workbook handouts* (if provided) are essentially the same as the electronic HTML version of the workbook on the CD. The paper workbook is simply a printed version of the HTML workbook on CD, but with minor formatting changes and some optional-reading items deleted to save space. However, the HTML workbook also contains active hyperlinks to sections and related information. In addition, the HTML workbook allows examples to be cut-and-paste into a text editor or the Python command-line (some example code is not in the CD's Examples directory). Therefore, the paper workbook is generally used only for taking notes; students are encouraged to follow along by viewing the HTML workbook in a web browser.

CONTENTS	2
PREFACE	7
ABOUT THIS CLASS	7
ABOUT THE PROGRAM EXAMPLES.....	7
SUGGESTED SUPPLEMENTAL BOOKS	8
ABOUT THE INSTRUCTOR	9
DAILY SCHEDULE	9
COURSE TOPICS	9
1. GENERAL PYTHON INTRODUCTION	11
SO WHAT'S PYTHON?.....	11
WHY DO PEOPLE USE PYTHON?	12
SOME QUOTABLE QUOTES.....	13
A PYTHON HISTORY LESSON	14
ADVOCACY NEWS	16
WHAT'S PYTHON GOOD FOR?	17
WHAT'S PYTHON NOT GOOD FOR?	18
PYTHON TECHNICAL FEATURES	19
PYTHON PORTABILITY	20
ON APPLES AND ORANGES	21
SUMMARY: WHY PYTHON?	23
2. USING THE INTERPRETER.....	26
HOW PYTHON RUNS PROGRAMS	26
HOW YOU RUN PROGRAMS.....	28
EXAMPLE PROGRAM LAUNCHES.....	30
CONFIGURATION DETAILS.....	31
MODULE FILES: A FIRST LOOK	32
THE IDLE INTERFACE (NEW IN 1.5.2)	33
OTHER PYTHON IDEs.....	35
TIME TO START CODING	35
LAB SESSION 1.....	36

3. TYPES AND OPERATORS	37
THE ‘BIG PICTURE’	37
NUMBERS	38
THE DYNAMIC TYPING INTERLUDE.....	40
STRINGS.....	42
LISTS	48
DICTIONARIES	50
TUPLES	54
FILES	55
GENERAL OBJECT PROPERTIES	56
SUMMARY: PYTHON’S TYPE HIERARCHIES.....	60
BUILT-IN TYPE GOTCHAS	61
LAB SESSION 2.....	62
4. BASIC STATEMENTS.....	63
GENERAL CONCEPTS.....	64
ASSIGNMENT	64
EXPRESSIONS.....	66
PRINT.....	66
IF SELECTIONS	67
PYTHON SYNTAX RULES	69
DOCUMENTATION SOURCES INTERLUDE	70
TRUTH TESTS REVISITED.....	71
WHILE LOOPS	73
BREAK, CONTINUE, PASS, AND THE LOOP ELSE	74
FOR LOOPS.....	75
LOOP CODING TECHNIQUES.....	76
COMPREHENSIVE EXAMPLES.....	79
BASIC CODING GOTCHAS	81
PREVIEW: PROGRAM UNIT STATEMENTS	81
LAB SESSION 3.....	82
5. FUNCTIONS.....	83
FUNCTION BASICS.....	83
SCOPE RULES IN FUNCTIONS	85
MORE ON “GLOBAL”	86
MORE ON “RETURN”	87
MORE ON ARGUMENT PASSING	87
SPECIAL ARGUMENT MATCHING MODES	88
ODDS AND ENDS	91
FUNCTION DESIGN CONCEPTS	95
FUNCTIONS ARE OBJECTS: INDIRECT CALLS	96
FUNCTION GOTCHAS	97
LAB SESSION 4.....	99
6. MODULES	101
MODULE BASICS	101
MODULE FILES ARE A NAMESPACE	102
NAME QUALIFICATION.....	103
IMPORT VARIANTS	104
RELOADING MODULES	105
PACKAGE IMPORTS	106
ODDS AND ENDS	107
MODULE DESIGN CONCEPTS	109
MODULES ARE OBJECTS: METAPROGRAMS	110

MODULE GOTCHAS	111
LAB SESSION 5.....	113
7. CLASSES.....	114
OOP: THE BIG PICTURE	115
A FIRST LOOK: CLASS BASICS.....	116
A CLOSER LOOK: CLASS TERMINOLOGY	120
USING THE CLASS STATEMENT.....	121
USING CLASS METHODS	122
CUSTOMIZATION VIA INHERITANCE	124
SPECIALIZING INHERITED METHODS	125
OPERATOR OVERLOADING IN CLASSES	127
NAMESPACE RULES: THE WHOLE STORY	130
OPTIONAL READING: OOP AND INHERITANCE	132
OPTIONAL READING: OOP AND COMPOSITION.....	134
CLASSES ARE OBJECTS: FACTORIES.....	136
METHODS ARE OBJECTS: BOUND OR UNBOUND.....	136
ODDS AND ENDS	137
CLASS GOTCHAS	142
SUMMARY: OOP IN PYTHON	143
LAB SESSION 6.....	144
8. EXCEPTIONS	145
EXCEPTION BASICS	145
FIRST EXAMPLES.....	147
EXCEPTION IDIOMS	147
EXCEPTION CATCHING MODES	148
CLASS EXCEPTIONS.....	151
EXCEPTION GOTCHAS	153
LAB SESSION 7.....	154
9. BUILT-IN TOOLS OVERVIEW	155
DEBUGGING OPTIONS.....	156
INSPECTING NAME-SPACES	158
DYNAMIC CODING TOOLS	159
TIMING AND PROFILING PYTHON PROGRAMS	161
PACKAGING PROGRAMS FOR DELIVERY	163
DEVELOPMENT TOOLS FOR LARGER PROJECTS.....	164
TESTING TOOLS IN THE STANDARD LIBRARY.....	165
SUMMARY: PYTHON TOOL-SET LAYERS.....	167
LAB SESSION 7.....	167
10. SYSTEM INTERFACES	168
SYSTEM MODULES OVERVIEW	168
ARGUMENTS, STREAMS, SHELL VARIABLES	170
FILE TOOLS	171
DIRECTORY TOOLS	171
FORKING PROCESSES	175
PYTHON THREAD MODULES	176
FORK VERSUS SPAWNV	179
LAB SESSION 8.....	179
11. GUI PROGRAMMING.....	180
PYTHON GUI OPTIONS	180
THE TKINTER ‘HELLO WORLD’ PROGRAM.....	182
ADDING BUTTONS, FRAMES, AND CALLBACKS.....	183

GETTING INPUT FROM A USER	184
MORE DETAILS	185
OOP: BUILDING GUIs BY SUBCLASSING FRAMES	185
OOP: REUSING GUIs BY SUBCLASSING, 'IS-A'	186
OOP: REUSING GUIs BY ATTACHING, 'HAS-A'	187
IMAGES	188
GRID LAYOUTS	189
CANVAS, TEXT, DIALOGS, AND TOPLEVEL	190
LARGER EXAMPLES: PROGRAMMING PYTHON, ED. 2+	191
TKINTER ODDS AND ENDS	200
LAB SESSION 8	201
12. DATABASES AND PERSISTENCE	202
OBJECT PERSISTENCE: SHELVES	202
STORING CLASS INSTANCES	204
PICKLING OBJECTS WITHOUT SHELVES	206
USING SIMPLE DBM FILES	207
SHELVE GOTCHAS	208
ZODB OBJECT-ORIENTED DATABASE	208
PYTHON SQL DATABASE API	211
PERSISTENCE ODDS AND ENDS	214
LAB SESSION 9	214
13. TEXT PROCESSING	215
STRING OBJECTS: REVIEW	215
SPLITTING AND JOINING STRINGS	216
REGULAR EXPRESSIONS	218
PARSING LANGUAGES	220
LAB SESSION 10	220
14. INTERNET SCRIPTING	221
USING SOCKETS IN PYTHON	221
THE FTP MODULE	225
EMAIL PROCESSING	228
OTHER CLIENT-SIDE TOOLS	232
BUILDING WEB SITES WITH PYTHON	233
WRITING SERVER-SIDE CGI SCRIPTS	234
BASIC CGI EXAMPLE	235
CGI CONTROLS: TEST5.HTML/.CGI (PP BOOK)	237
THE GRAIL WEB BROWSER	241
JYTHON: PYTHON FOR JAVA SYSTEMS	243
ACTIVE SCRIPTING AND COM	246
OTHER INTERNET-RELATED TOOLS	248
LAB SESSION 10	249
15. EXTENDING PYTHON IN C/C++	250
REVIEW: PYTHON TOOL-SET LAYERS	250
WHY INTEGRATION?	251
THE 'BIG PICTURE' REVISITED	252
INTEGRATION MODES	253
A SIMPLE C EXTENSION MODULE	254
C MODULE STRUCTURE	255
BINDING C EXTENSIONS TO PYTHON	256
DATA CONVERSIONS: PYTHON \leftrightarrow C	258
C EXTENSION TYPES	259
USING C EXTENSION TYPES IN PYTHON	264

WRAPPING C EXTENSIONS IN PYTHON	265
WRITING EXTENSIONS IN C++	266
SWIG EXAMPLE (PP BOOK).....	267
PYTHON AND RAPID DEVELOPMENT.....	268
LAB SESSION 11.....	270
16. EMBEDDING PYTHON IN C/C++.....	271
GENERAL EMBEDDING CONCEPTS	271
RUNNING SIMPLE CODE STRINGS	273
CALLING OBJECTS AND METHODS.....	275
RUNNING STRINGS: RESULTS & NAME-SPACES	279
OTHER CODE STRING POSSIBILITIES	280
REGISTERING PYTHON OBJECTS AND STRINGS	283
ACCESSING C VARIABLES IN PYTHON.....	287
C API EQUIVALENTS IN PYTHON	287
RUNNING CODE FILES FROM C	288
PRECOMPILING STRINGS INTO BYTE-CODE	288
EMBEDDING UNDER C++	289
MORE ON OBJECT REFERENCE COUNTS	289
INTEGRATION ERROR HANDLING.....	291
AUTOMATED INTEGRATION TOOLS	292
LAB SESSION 12.....	294
17. RESOURCES	295
INTERNET RESOURCES	295
PYTHON BOOKS	296
PYTHON CONFERENCES AND SERVICES	298
LAB SESSION 12.....	299
AND FINALLY	299
LABORATORY EXERCISES	300
LAB 1: USING THE INTERPRETER.....	301
LAB 2: TYPES AND OPERATORS.....	302
LAB 3: BASIC STATEMENTS	305
LAB 4: FUNCTIONS	306
LAB 5: MODULES.....	308
LAB 6: CLASSES.....	309
LAB 7: EXCEPTIONS AND BUILT-IN TOOLS	313
LAB 8: SYSTEM INTERFACES AND GUIs	314
LAB 9: PERSISTENCE.....	316
LAB 10: TEXT PROCESSING AND THE INTERNET.....	316
LAB 11: EXTENDING PYTHON IN C/C++	317
LAB 12: EMBEDDING PYTHON IN C/C++	319
SELECTED EXERCISE SOLUTIONS.....	321
LAB 1: USING THE INTERPRETER	321
LAB 2: TYPES AND OPERATORS	323
LAB 3: BASIC STATEMENTS	328
LAB 4: FUNCTIONS.....	330
LAB 5: MODULES.....	333
LAB 6: CLASSES.....	336
LAB 7: EXCEPTIONS AND BUILT-IN TOOLS.....	343

Preface

About this class



This class introduces students to the [Python](#) programming language, and common Python applications. Through lectures and hands-on laboratory work, students learn the basics of Python programming. We'll cover the [Python](#) language, study C/C++ integration topics, and introduce common Python application domains such as GUIs, databases, web sites, and system tools.

This class presents Python in a *bottom-up* fashion – we start with small details and build to larger and larger examples as we move along. This class is also designed to provide an *in-depth* look at Python itself, and a general and *broad* survey of Python applications – although we will study some advanced Python application domains, we won't have time to do full justice to most. Specialized domains (e.g., Tkinter GUIs, numeric programming, Internet scripting) can be more thoroughly covered in follow-up books and classes.

There are no real prerequisites for most of this course, though the last two modules on C integration will be more meaningful if you have basic C programming skills. Any prior programming or scripting experience applies, no matter how minor. For more details about this course, see my [training webpage](http://home.earthlink.net/~python-training) (<http://home.earthlink.net/~python-training>).

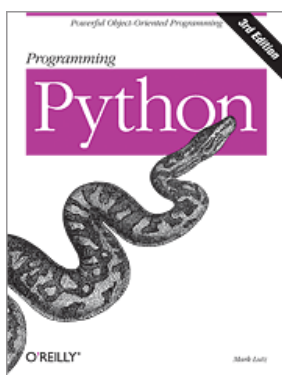
About the program examples



The examples in this course are based on Python versions 2.5 and later (at this writing). We will look at some cutting edge new features of Python along the way; but since later releases are generally backward compatible with earlier ones, most of the material here should apply earlier Python releases as well.

Source code for all the lecture examples and laboratory exercises is available on this CD. You may also cut and paste code from these web pages, and at the end of unit web pages you will find links to source-code file directories on this CD-ROM for exercise solutions and lecture examples. In fact, if you are using this HTML form of the workbook, you don't need the hardcopy paper workbook at all; it's all in this package. Simply follow the links in the [table of contents page](#).

Suggested supplemental books



Because this class workbook is a self-contained document, there are no additional required texts for this class. However, students may wish to obtain related books to serve as supplemental resources. Here are some suggestions:

Directly related books

The core language sections of this class parallel the book *Learning Python 2nd Edition*, and the more advanced applications-level material in this class parallel the book *Programming Python 3rd Edition*, both published by [O'Reilly & Associates](#). The first of these two cover language fundamentals and is largely based upon the main portion of this class; it may be available in a 3rd Edition by late 2007. The latter book is an advanced continuation of the former, and is focused on applications-level programming topics.

Some clients may wish to provide students with copies of either or both of these texts for use after the class, depending upon student interest and skill levels. If you need to obtain copies of these books, they can be purchased from most large or technical bookstores, directly from O'Reilly (see URL <http://www.oreilly.com/>), or through the Python web site (<http://www.python.org/>) or amazon.com.

Recommended books

Though not required for this class, O'Reilly's *Python Pocket Reference 3rd Edition* is also suggested as a reference supplement that will prove handy after the class. I also recommend the text *Python Essential Reference 2nd Edition* by David Beazley, as well as *Python in a Nutshell 2nd Edition* and *Python Cookbook 2nd Edition* from O'Reilly, as supplemental reference books. In addition, there are now some 50 Python books on the market that may be of interest as well, depending upon your Python applications; see the (slightly dated) books list in the [Packages and Resources](#) lecture page for a list.

About the instructor

[Mark Lutz](#) is the world leader in [Python](#) training, the author of Python's earliest and best-selling texts, and a pioneering figure in the Python community.

Mark is the author of the O'Reilly books [Programming Python](#) and [Python Pocket Reference](#), and the primary co-author of [Learning Python](#), all currently in 2nd or 3rd Editions. He has been involved with Python since 1992, began teaching [Python classes](#) in 1997, and has instructed over 170 Python training sessions as of mid 2006.

In addition, he holds BS and MS degrees in computer science from the University of Wisconsin, and over the last two decades has worked on compilers, programming tools, scripting applications, and assorted client/server systems.

Whenever Mark gets a break from spreading the Python word, he leads an ordinary, average life in Colorado. Mark can be reached by email at lutz@rmi.net, or on the web at <http://www.rmi.net/~lutz>.

Daily schedule

Each class proceeds at a slightly different rate, and our schedule is going to vary each day. Typically, there will four to six lectures sessions per day, with laboratory work time after each, and an hour for lunch. The exact session schedule depends on student needs and interests, and on how much interaction students desire; questions and comments at any time are encouraged.

Also note that we never cover all the material in this workbook in a 3-day class. In general, the workbook contains a superset of topics to be presented, and some of its examples are included for student self-study only. Material skipped is either of minor importance, or optional reading. We may also sometimes depart from the workbook to explore special topics of interest to students. This is your class – please ask about topics not listed above.

Course topics

Introducing Python

General Python introduction

Python Basics

Using the interpreter

Types and operators

Basic statements

Functions

Modules

Classes

Exceptions

Built-in tools

Python Applications

System interfaces

GUI programming

Databases and persistence

Text processing

Internet scripting

Extending Python in C/C++

Embedding Python in C/C++

Where to go from here?

Python resources

1. General Python Introduction

Topics

- ◆ So what's Python?
- ◆ Why do people use Python?
- ◆ A Python history lesson
- ◆ Advocacy news
- ◆ What's Python good for?
- ◆ What's Python not good for?
- ◆ Python technical features
- ◆ Python portability

So What's Python?

“An open source, object-oriented, scripting language”

- ◆ An “open source” software project

A BDFL, plus a cast of thousands

You are not held hostage by a vendor (VB!)

- ◆ An “object-oriented” language

OOP is an option, but a nice one

Supports code reuse

♦ A “scripting” language

But not just for shell tools—general purpose

Control language: C libs, Com, .NET, Java

Easy to use: 3x ~ 4x less code than Java, C++

Plus...

- General purpose
- Tactical or strategic
- Stand alone or embedded
- Very high-level, dynamic

Why Do People Use Python?

Software Quality

- *Readable syntax: maintainable*
- *Coherent design, fewer interactions*
- *Simple enough to remember*
- *Art versus Engineering → maintenance, reuse*

Developer Productivity

- *Smaller programs, flexible code (“agile”)*
- *Rapid turnaround, code reuse*
- *Tactical and strategic roles*
- *Good in boom and bust times*

And other reasons...

- *Program portability*
- *Component integration*
- *Vast application libraries*
- *Open source*

Some Quotable Quotes

- ◆ “Python looks like it was designed, not accumulated.”
- ◆ “It bridges the gap between scripting languages and C.”
- ◆ “It’s as easy or as powerful as you want it to be.”
- ◆ “Python: less filling, tastes great. :-)”
- ◆ “Python fits your brain.”

A more formal definition?

Seen on comp.lang.python...

*“python, (Gr. Myth. An enormous serpent that lurked in the cave of Mount Parnassus and was slain by Apollo) **1.** any of a genus of large, non-poisonous snakes of Asia, Africa and Australia that crush their prey to death. **2.** popularly, any large snake that crushes its prey. **3.** totally awesome, bitchin’ language invented by that rad computer geek Guido van Rossum that will someday crush the \$’s out of certain *other* so-called VHLLs ;-)”*

A Python History Lesson

- ◆ Created by [Guido van Rossum](#) in Amsterdam, 1990



Guido Van Rossum

- ◆ USENET newsgroup started in 1994
comp.lang.python, www.python.org
3rd party add-ons: [Vaults of Parnassus](#), [PyPI](#)



- ◆ Python Software Foundation (PSF): O'Reilly, ActiveState, Zope



- ◆ First Python books appeared Fall, 1996, over 50 available by 2003



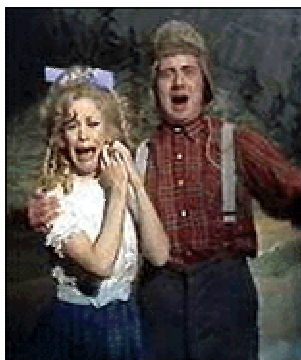
- ◆ International following: US, Europe, Asia, Australia



- ◆ Community: 750K to 1M users worldwide (guesstimate), user groups, two annual US conferences (PyCon, OSCON), European conferences (EuroPy, PythonUK)



- ◆ Named after 70s BBC comedy group “*Monty Python's Flying Circus*”



Advocacy News

Noteworthy users

Web services: *Google, Yahoo*

Animation: *Industrial Light & Magic, ImageWorks, Disney, Pixar*

Financial: *JP Morgan Chase, UBS, Getco, NYSE*

Hardware Testing: *Intel, Hewlett-Packard, Seagate, Qualcomm, Micron*

Numerics: *National Weather Service, Los Alamos, NASA*

P2P downloads: *BitTorrent (32M DLs, 1/3 of all inet traffic!)*

Other: *ESRI, NSA, IronPort, Red Hat, Jet Propulsion Lab, Eve Online, ...*

Domains

IronPython for .NET/Mono (Msoft),

Windows COM

Jython Java port

Zope & Plone web site frameworks

Mac OS X Cocoa integration

Sun's Coyote project?

Cellphone ports

Compilers

IronPython (Microsoft), Python.net for C#/.NET"

Jython for Java

Standard C-Python

PyPy & Psyco

Parrot

Group therapy

Python user groups: Oregon, Bay Area, DC, Colorado, Italy, England, Korea,...

Books

Over 50 Python books available, 40+ English language, a dozen non-English books, more on the way (see Resources)

Press

O'Reilly's [Python Success Stories](#); Pythonology [users list](#); Guido on the cover of *Linux Journal*, *Dr. Dobbs Journal*, in *Washington Post*

Education

Guido's Computer Programming For Everybody (CP4E), edu-sig, tutors list

Services

Commercial support, training; pre-packaged distributions; standard on Linux and Mac OS X

Jobs

Python job board; hundreds of hits on monster and dice

Other

30% increase in python.org traffic for year ended March '05

PyCon conference attendance increase: 400-500 '05 vs 200-300 '04

Jolt productivity award given to Python 2.4 early '05

InfoWorld: 6% gain in popularity '05 (14% use rate '05 vs 8% '04)

Google open source site, Python projects

What's Python Good For?

General purpose:

=> Almost anything computers can do

◆ System programming: shell tools, test scripts

sockets, regex, POSIX calls, threads, streams

◆ Graphical user interfaces

Tk, wxPython, Qt, Gtk, MFC, X11, Swing (Jython)

◆ Internet scripting

CGI, email, FTP, Telnet, Jython applets, XML-RPC, SOAP, ActiveScripting, mod_Python (Apache)

◆ Database programming

Persistent objects, ZODB, Oracle, Informix, Sybase, MySQL,...

- ◆ **Component integration**

Product customization and testing, embedded scripting, system front-ends

- ◆ **Rapid Application Development**

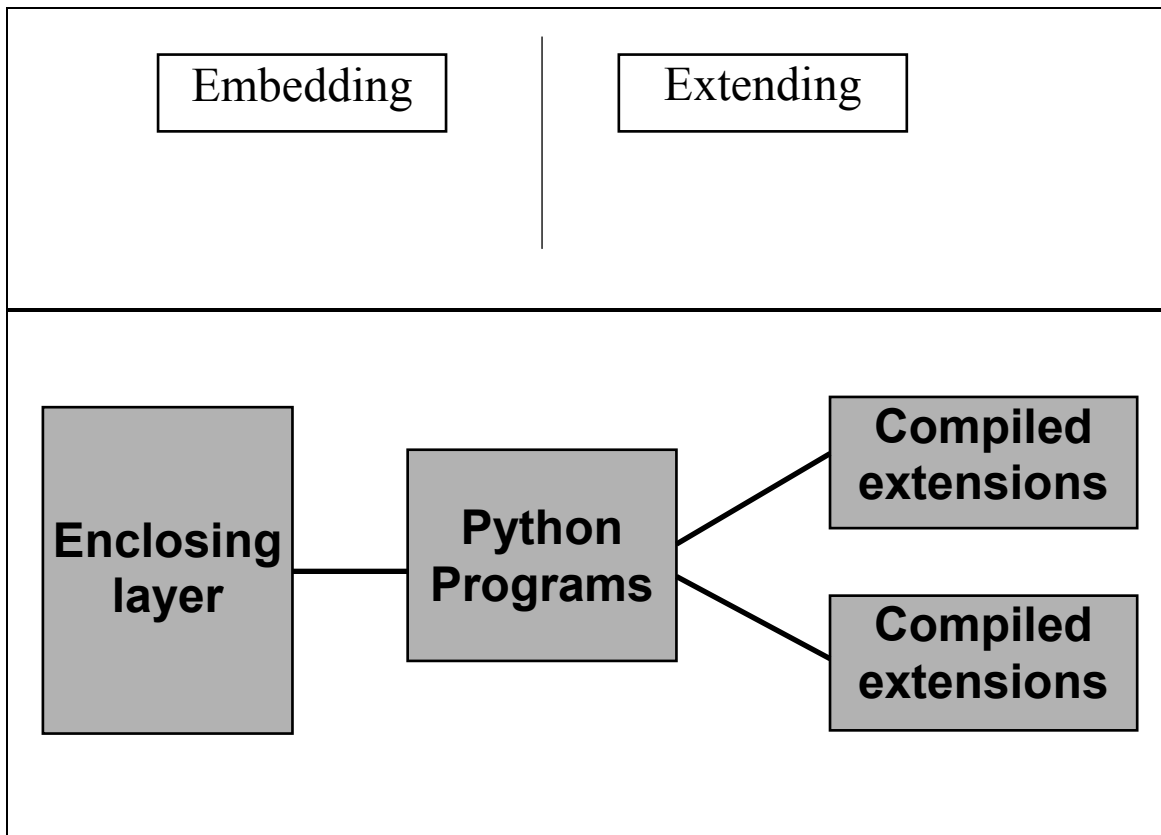
Prototype-and-migrate, fast turnaround, deliverable prototypes

- ◆ **And more specific domains: general purpose**

COM (PyWin32), Numeric programming (NumPy), Gaming (PyGame), graphics (OpenGL, Blender), AI, CORBA,...

What's Python Not Good For?

- ◆ **Fast enough for most tasks as is**
- ◆ **Most real tasks run linked-in C code**
- ◆ **Exception truly speed-critical components**
- ◆ **Solution: implement in C and export to Python**
- ◆ **Python is optimized for speed-of-development**
- ◆ **Python is designed for multi-language systems**
- ◆ **Example: Python Numeric Programming**
- ◆ **Psyco JIT may improve the speed story**



Python Technical Features

- ◆ **No compile or link steps**
Rapid development-cycle turnaround
- ◆ **No type declarations**
Programs are simpler, shorter, and flexible
- ◆ **Automatic memory management**
Garbage collection avoids bookkeeping code
- ◆ **High-level datatypes and operations**
Fast development using built-in object types
- ◆ **Object-oriented programming**
Code structuring and reuse, C++ integration

- ♦ **Extending and embedding in C**
Optimization, customization, system 'glue'
- ♦ **Classes, modules, exceptions**
Modular 'programming-in-the-large' support
- ♦ **A simple, clear syntax and design**
Readability, maintainability, ease of learning
- ♦ **Dynamic loading of C modules**
Simplified extensions, smaller binary files
- ♦ **Dynamic reloading of Python modules**
Programs can be modified without stopping
- ♦ **Universal 'first-class' object model**
Fewer restrictions and special-case rules
- ♦ **Interactive, dynamic nature**
Incremental testing, runtime program coding/construction
- ♦ **Access to interpreter information**
Metaprogramming, introspective objects
- ♦ **Wide interpreter portability**
Cross-platform systems without ports
- ♦ **Compilation to portable byte-code**
Execution speed, protecting source-code
- ♦ **Built-in interfaces to external services**
O/S, GUI, persistence, DBMS, regular expressions...
- ♦ **True 'freeware': *Open Source* software**
May be embedded/shipped without copyright restrictions

Python Portability

- ♦ **Core Language + Standard Library**
 - ♦ *Unix, Linux, Windows, Macs*
 - ♦ *Cray supers, IBM mainframes, VxWorks realtime*
 - ♦ *PDA's: PalmOS, PocketPC, Zaurus*

- ♦ *OS/2, VMS, Next, BeOS, QNX, Itanium*
 - ♦ *Amiga, AtariST*
 - ♦ *PlayStation, XBox, Gamecube*
 - ♦ *Nokia Series 60 cellphones*
 - ♦ *Ipods*
 - ♦ *...*
- ♦ **Platform-specific Extensions**
 - ♦ *COM (Windows—PyWin32 extension)*
- ♦ **General Portability**
 - ♦ *Bytecode is platform-neutral*
 - ♦ *Tkinter GUI library: X (Unix), Windows, Macs*
 - ♦ *Standard library system calls (module “os”)*

On Apples and Oranges

<i>Versus</i>	<i>Python advantage</i>	<i>Description</i>
Tcl	Power	Python better at “programming in the large”: module, OOP, exceptions, etc.
Perl	Coherence	Python has a readable, maintainable syntax, fewer special variables, etc.
Java	Simplicity Turnaround	Built-in objects, dynamic typing, etc.; can be freely shipped with products.

C++	Simplicity Turnaround	Interpreted language turnaround; avoids C++ language complexity.
Smalltalk	Conventional	In Python, “if” statements are not message-receiver objects.
Ruby	Flexibility, maturity	Python more readable, more like C++: both procedural and OO (optional)
Scheme, Lisp	Conventional	Python’s syntax is closer to traditional languages like C and Pascal.
Visual Basic	Power, Portability	Python is powerful, cross-platform, and not controlled by one company (Python cannot be discontinued!)

- ◆ **But your mileage may vary**
- ◆ *Different language design goals*
- ◆ *Programmers matter too*
- ◆ *Many languages are a Good Thing*
- ◆ *Python coding may be too easy: design and brains still matter*

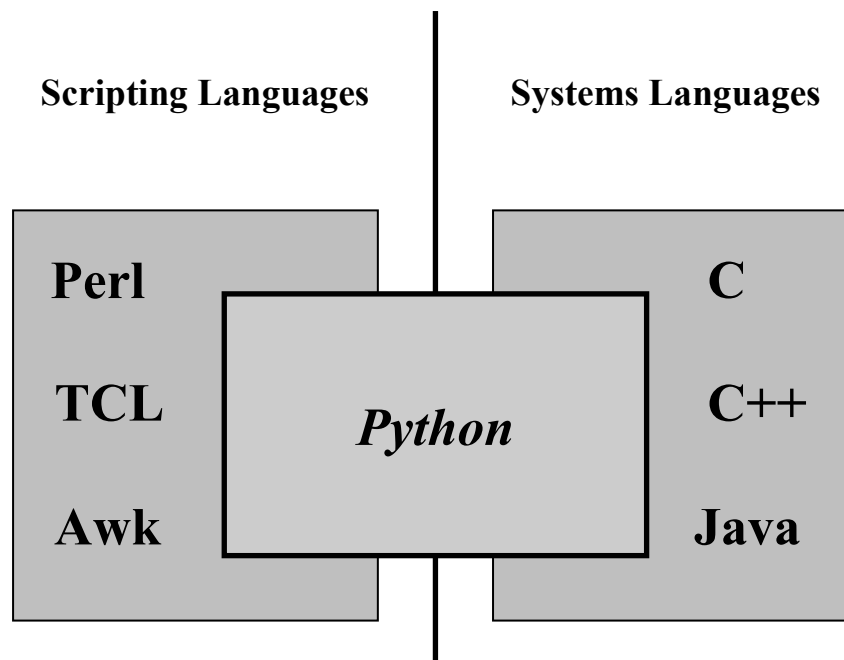


Summary: Why Python?

- ◆ It's object-oriented
 - ◆ *Powerful OO support*
 - ◆ *But OO is an option*
- ◆ It's free
 - ◆ *Can freely embed and ship in products*
 - ◆ *Can even sell the source-code!*
- ◆ It's portable
 - ◆ *Runs everywhere: Unix, Windows, Mac,...*
 - ◆ *Portable byte-code, portable Tkinter GUI interface*
- ◆ It's powerful
 - ◆ *Built-in types and operations*
 - ◆ *Dynamic typing, libraries, modules, garbage collection, ...*
- ◆ It's mixable
 - ◆ *Python/C, Python/C++, Python/Java, COM*
- ◆ It's easy to use
 - ◆ *Fast turnaround after changes*
 - ◆ *A simple language and syntax*
- ◆ It's easy to learn
 - ◆ *For developers and product customers*

=> *Quality and Productivity*

A scripting language doesn't have to look like one



A morality tale of Perl versus Python

(The following was posted recently to the rec.humor.funny USENET newsgroup, by Larry Hastings, and is reprinted here with the original author's permission. I don't necessarily condone language wars.)

This has been percolating in the back of my mind for a while. It's a scene from *The Empire Strikes Back*, reinterpreted to serve a valuable moral lesson for aspiring programmers.

EXTERIOR: DAGOBAH--DAY

With Yoda strapped to his back, Luke climbs up one of the many thick vines that grow in the swamp until he reaches the Dagobah statistics lab. Panting heavily, he continues his exercises--grepping, installing new packages, logging in as root, and writing replacements for two-year-old shell scripts in Python.

YODA: Code! Yes. A programmer's strength flows from code maintainability. But beware of Perl. Terse syntax... more than one way to do it... default variables. The dark side of code maintainability are they. Easily they flow, quick to join you when code you write. If once you start down the dark path, forever will it dominate your destiny, consume you it will.

LUKE: Is Perl better than Python?

YODA: No... no... no. Quicker, easier, more seductive.

LUKE: But how will I know why Python is better than Perl?

YODA: You will know. When your code you try to read six months from now.

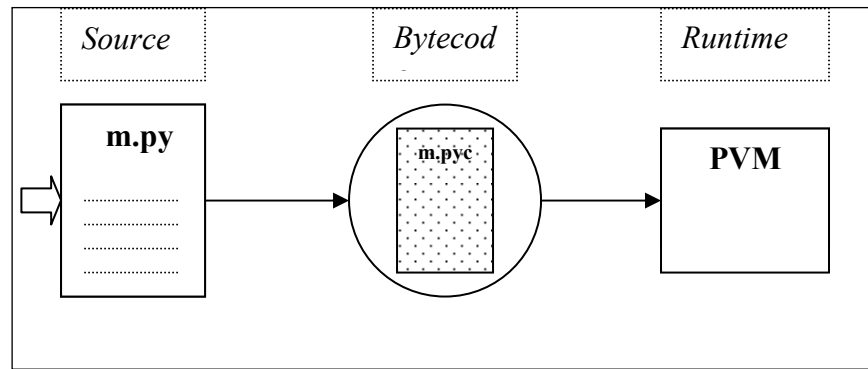
2. Using the interpreter

Topics

- ◆ How Python runs programs
- ◆ How you run programs
- ◆ Example program launches
- ◆ Python configuration details
- ◆ A first look at module files
- ◆ The IDLE interface
- ◆ Other Python IDEs

How Python runs programs

- **Execution Model**
 - A “script”: text file of statements, “mod.py”
 - Mod.py (sourcecode) =>
 Mod.pyc (bytecode) =>
 Python Virtual Machine



- **Execution Variations**

- ***Psyco***: a just-in-time compiler for Python bytecode
- ***Shedskin***: Python-to-C++ compiler, if limited dynamic typing (avg 12X Psyco, 40x Python)
- ***Frozen*** binary executables: ***Py2Exe*** (Windows), ***PyInstaller*** (+Linux), ***Freeze*** (Status?)
- Other VMs: ***PyPy*** (Python VM in Python), ***Parrot*** (language neutral)...

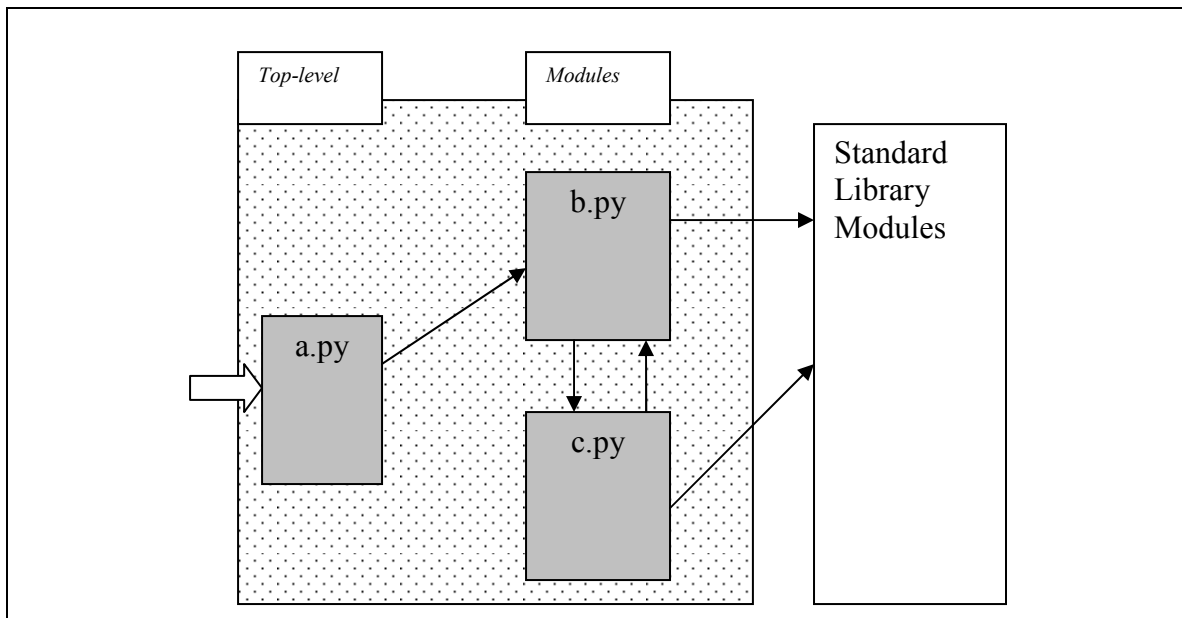
3 Implementations Today

- ***C Python***: the standard
- ***Jython***: for scripting Java apps
- ***IronPython***: for scripting C#/.Net apps
- Plus JPyte, Python.net: other ways to Java/.Net

How you run programs

- **Program architecture**

- Program = multiple .py text files
 - One is the “main” top-level file: launch to run
 - Others are “modules”: libraries of tools
 - Some modules come from the “standard library”
- Modules accessed and linked by imports: “import module”
 - Import = find it, compile it (maybe), run it (once)
- Attributes fetched from objects: “module.attr”
 - Variables inside objects



```
b.py
def f():
    ...
```

```
a.py
import b
b.f()
```

sys.path:

"." | PYTHONPATH | .pth files | Std Libs

• Program launch options

- Interactive coding (">>>")
- Shell/DOS command lines
- Double-click icons (raw_input trick)
- IDLE: a free IDE (Windows Start button, Runscript)
- Module imports, reloads
- Embedding calls

- Unix executable scripts (#!)
- Other IDEs: Komodo, PythonWare,...

Example program launches

◆ UNIX-style scripts

```
file: brian
#!/usr/local/bin/python          #or: /bin/env python
print 'The Bright Side of Life...' # another comment

% brian
```

◆ Command lines, module files

```
% python spam.py -i eggs -o bacon
```

◆ Interactive command line

```
% python
>>> print 'Hello world!'
Hello world!
>>> lumberjack = "okay"      # ctrl-D or ctrl-Z to exit
```

◆ Embedded code/objects (day 3)

```
Py_Initialize();
PyRun_SimpleString("x = brave + sir + robin");
```

◆ Platform-specific startup methods

- ◆ *Command-line interfaces*
- ◆ *GUI start-up interfaces: .pyw files*

Configuration details

- ◆ Module search path, for importing from other directories (only!)
 - PYTHONPATH shell/environment variable
 - “.pth” configuration files: C:\Python24\mypath.pth
 - sys.path builtin list changes , but temporary
 - Module search path = [main file’s home directory + PYTHONPATH + standard libraries + .pth file contents]
 - Only add user-defined directories to search path (e.g. PYTHONPATH), not home directory or standard libs
- ◆ Other settings
 - Interactive startup file: PYTHONSTARTUP
 - GUI variables (not on Windows): TCL_LIBRARY, TK_LIBRARY
 - System search path, to find python: PATH

Installation details

- ◆ Python comes in binary or C source-code forms
- ◆ Windows self-installer: simple double-click, includes Tk
- ◆ Linux, Mac OS X: Python is a standard component
- ◆ C source code configures/builds automatically
- ◆ See Python source distribution “Readme” for install details

A UNIX config file (~/.cshrc or ~/.login)

```
#!/bin/csh
set path = (/usr/local/bin $path)
setenv PYTHONPATH /usr/home/pycode/utilities
setenv PYTHONPATH /usr/lib/pycode/package1:$PYTHONPATH
```

A Windows config file (C:\autoexec.bat)

- Reboot after autoexec.bat changes
- Use the ControlPanel/System/Advanced GUI to set this in recent versions of Windows (and restart Python)
- “cd C:\Python24”, if you don’t want to set your PATH

```
doskey /insert
PATH %PATH%;C:\Python24
set PYTHONPATH=C:\pycode\utilities
set PYTHONPATH=D:\pycode\package1;%PYTHONPATH%
```

A path file (C:\Python24\mypath.pth)

```
c:\pycode\utilities
d:\pycode\package1
```

A GUI test session

- ◆ Type this to test your Python/GUI configuration
- ◆ On Linux, Mac OS X: may need extra packages (CD)

```
% python
>>> from Tkinter import *
>>> w = Button(text="Hello", command='exit')
>>> w.pack()
>>> w.mainloop() # don't type this line in IDLE!
```



Module files: a first look

- ◆ To avoid retyping code interactively
- ◆ ‘.py’ filename suffix needed if imported
- ◆ Import and use names at top-level of module file
- ◆ “dir(module)” lists a module’s attributes
- ◆ “help(module)” gives help
- ◆ “reload(module)” reruns an already-loaded file’s code again; import won’t!

file myfile.py

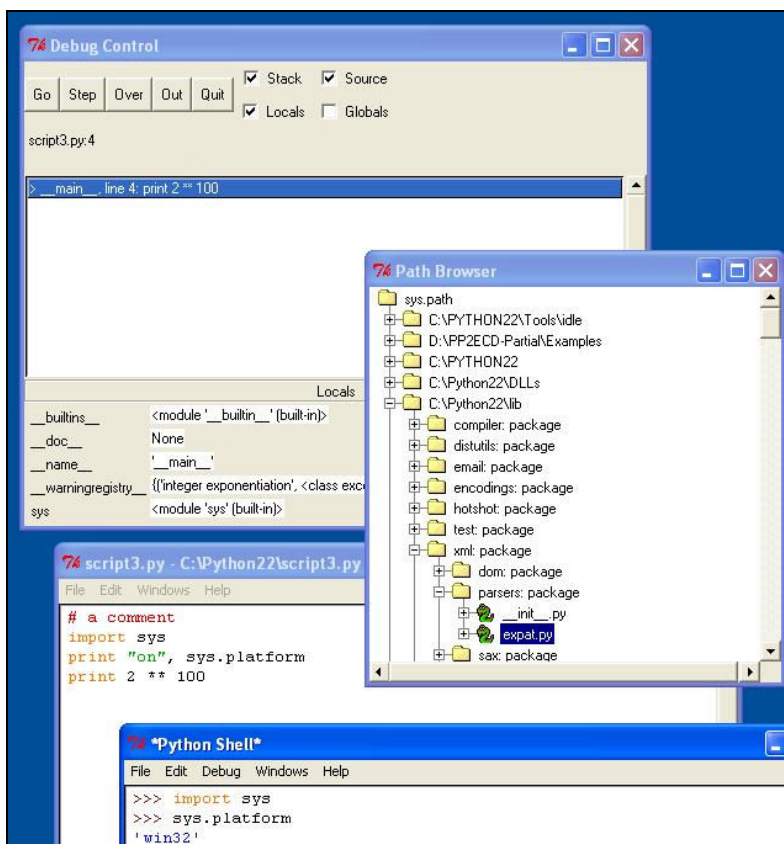
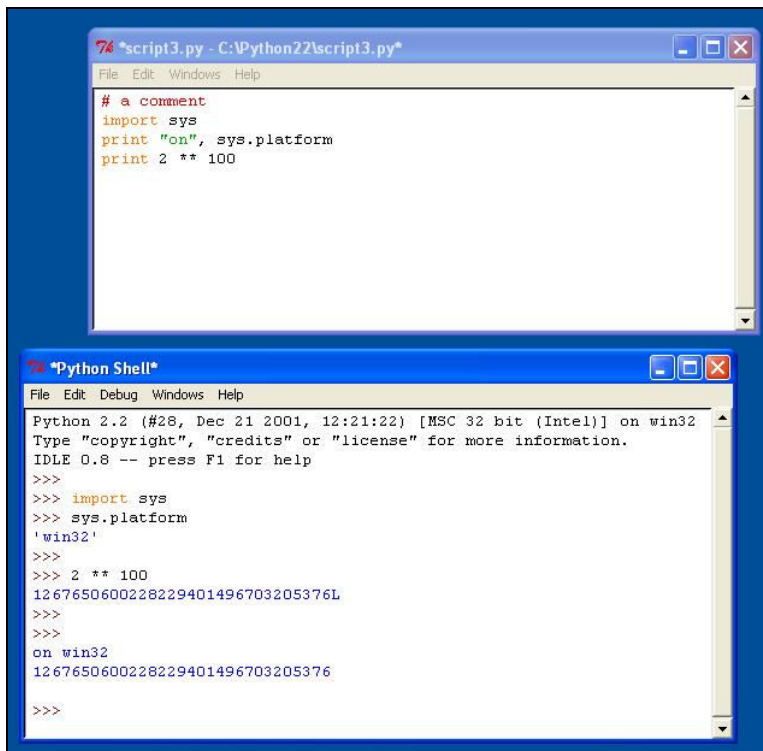
```
name = "The Meaning of Life"
```

% python	↵Start Python
>>> import myfile	↵Load module file
>>> print myfile.name	↵Use its names
The Meaning of Life	

% python	↵Start Python
>>> from myfile import name	↵Load names
>>> print name	↵Use name directly
The Meaning of Life	

The IDLE interface (new in 1.5.2)

- ◆ A simple integrated development environment
- ◆ Shipped and installed as part of the Python package
- ◆ Editor, syntax coloring, debugging, class browser, etc.
- ◆ Alternative to shell command line + editor, or clicking
- ◆ Written in Python, using the Tkinter GUI API
- ◆ See Python “Tools” directory, or Windows “Start” button entry
- ◆ Requires Python 1.5.2 or later, Tk 8.0 or later
- ◆ HINT: Alt-P/Alt-N scroll through command history



Other Python IDEs

- *IDLE*: ships with Python
- Eclipse, with PyDev (or other) Python plug-in
- *ActiveState*: Komodo, VisualPython, PythonWin (<http://www.activestate.com/>)
- *PythonWare*: Pythonworks (still active?) (<http://www.pythonware.com/>)
- *Others*: WingIDE, ... (<http://www.python.org/> editors page)
- *Basic*: text editor (Notepad, vi) + command-line window (DOS, xterm)

GUI Builders (ahead)

- Komodo and PythonWare include interactive GUI builders, that generate Tkinter code
- Other GUI builders for wxPython, PyQt: BoaConstructor, BlackAdder, wxDesigner, QtDesigner
- Application builders: Dabo, PythonCard
- Website builders: Django, TurboGears, Zope, WebWare, mod_python, Plone, ...

Time to start coding

- ◆ Lab exercises are at the [end of your handbook](#)
- ◆ Source code for lecture [examples](#) and lab [solutions](#) on disk
- ◆ Some solution write-ups appear [after the labs section](#)
- ◆ You are encouraged to "cheat": see solutions
- ◆ Ask the instructor for hints, tips, and help

Lab Session 1

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

3. Types and operators

Preview: built-in objects

<i>Object type</i>	<i>Example constants/usage</i>
Numbers	<code>3.14</code> , <code>1234</code> , <code>999L</code> , <code>3+4j</code> , <code>decimal</code>
Strings	<code>'spam'</code> , <code>"guido's"</code>
Lists	<code>[1, [2, 'three'], 4]</code>
Dictionaries	<code>{'food': 'spam', 'taste': 'yum'}</code>
Tuples	<code>(1, 'spam', 4, 'U')</code>
Files	<code>text = open('eggs', 'r').read()</code>

The 'big picture'

Python program structure

- ◆ Programs are composed of modules
- ◆ Modules contain statements
- ◆ Statements contain expressions
- ◆ Expressions create and process **objects**

Why use built-in types?

- ◆ Python provides objects and supports extensions
- ◆ Built-in objects make simple programs easy to write

- ◆ Built-in objects are components of extensions
- ◆ Often more efficient than custom data structures

Numbers

Standard types and operators

- ◆ Integer, floating-point, hex/octal constants
- ◆ ‘long’ integer type with unlimited precision
- ◆ Built-in mathematical functions: ‘pow’, ‘abs’
- ◆ Utility modules: ‘random’, ‘math’
- ◆ Complex numbers, ‘**’ power operator

Numeric Python (NumPy)

- ◆ An optional extension
- ◆ For advanced numeric programming
- ◆ Matrix object, interfaces to numeric libraries, etc.

Numeric constants

<i>Constant</i>	<i>Interpretation</i>
1234, -24	normal integers (C longs)
999999999L	long integers (unlimited size)
1.23, 3.14e-10	floating-point (C doubles)
0177, 0x9ff	octal and hex constants
3+4j, 3.0+4.0j	complex number constants
Decimal('0.11')	fixed-precision decimal (2.4)

Python expressions

- ◆ Usual algebraic operators: '+', '-', '*', '/', ...
- ◆ C's bitwise operators: "<<", "&", ...
- ◆ Mixed types: converted *up* just as in C
- ◆ Parenthesis group sub-expressions

Numbers in action

- ◆ Variables created when assigned
- ◆ Variables replaced with their value when used
- ◆ Variables must be assigned before used
- ◆ Expression results echoed back
- ◆ Mixed integer/float: casts up to float
- ◆ Integer division truncates (until 3.0?)

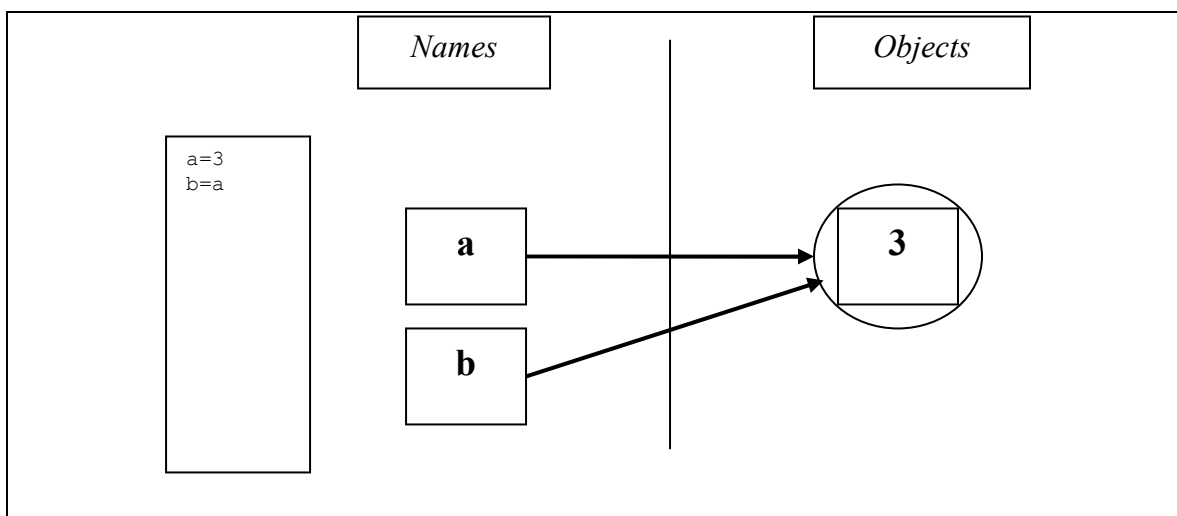
```
% python
>>> a = 3           # name created
>>> b = 4
>>> b / 2 + a       # same as ((4 / 2) + 3)
5
>>> b / (2.0 + a)   # same as (4 / (2.0 + 3))
0.8
```

Hint: use print if you don't want all the precision:

```
>>> 4 / 5.0
0.80000000000000004
>>> print 4 / 5.0
0.8
```

The dynamic typing interlude

- ◆ Names versus objects
- ◆ Names are always “references” to objects
- ◆ Names created when first assigned (or so)
- ◆ Objects have type, names do not
- ◆ Each value is a distinct object (normally)
- ◆ Shared references to mutables: side effects



Back to numbers: bitwise operations

```
>>> x = 1
>>> x << 2      # shift left 2 bits
4
>>> x | 2       # bitwise OR
3
>>> x & 1       # bitwise AND
1
```


	if x is true)
<code>not x</code>	Logical negation
<code><, <=, >, >=, ==, <>, !=, is, is not, in, not in</code>	Comparison operators, sequence membership
<code>x y</code>	Bitwise 'or'
<code>x ^ y</code>	Bitwise 'exclusive or'
<code>x & y</code>	Bitwise 'and'
<code>x << y, x >> y</code>	Shift x left or right by y bits
<code>x + y, x - y</code>	Addition/concatenation, subtraction
<code>x * y, x / y, x % y, x // y</code>	Multiply/repetition, divide, remainder/format, floor divide
<code>x ** y, -x, +x, ~x</code>	Power, unary negation, identity, bitwise compliment
<code>x[i], x[i:j], x.y, x(...)</code>	Indexing, slicing, qualification, function calls
<code>(...), [...], {...}, `...`</code>	Tuple, list, dictionary, conversion to string

Strings

- ◆ Ordered collections of characters
- ◆ No 'char' in Python, just 1-character strings
- ◆ Constants, operators, utility modules ('string', 're')
- ◆ Strings are 'immutable sequences'
- ◆ See 're' module for pattern-based text processing

Common string operations

Operation	Interpretation
-----------	----------------

<code>s1 = ''</code>	empty strings
<code>s2 = "spam's"</code>	double quotes
<code>block = """..."""</code>	triple-quoted blocks
<code>s3 = r"C:\d1\d2\file"</code>	raw strings (\ kept)
<code>s1 + s2, s2 * 3</code>	concatenate, repeat
<code>s2[i], s2[i:j], len(s2)</code>	index, slice, length
<code>"a %s parrot" % 'dead'</code>	string formatting
<code>for x in s2, 'm' in s2</code>	iteration/membership

Newer extensions

- String methods:

`X.split('+')` same as older `string.split(X, '+')`
 string module requires import, methods do not
 methods now faster, preferred to string module

- Unicode strings:

Multi-byte characters, for internationalization (I18N)
`U'xxxx'` constants, Unicode modules, auto conversions
 Can mix with normal strings, or convert: `str(U)`, `unicode(S)`

- Template formatting: string module, see ahead

Strings in action

```
% python
>>> 'abc' + 'def'      # concatenation: a new string
'abcdef'
>>> 'Ni!' * 4          # like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Indexing and slicing

```
>>> S = 'spam'
>>> S[0], S[-2]           # indexing from from or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]  # slicing: extract section
('pa', 'pam', 'spa')
```

Changing and formatting

```
>>> S = S + 'Spam!'      # to change a string, make a new one
>>> S
'spamSpam!'

>>> 'That is %d %s bird!' % (1, 'dead')    # like C sprintf
That is 1 dead bird!
```

Advanced formatting examples

```
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234   ...001234'

>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'

>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> int(x)
1
>>> round(x, 2)
1.23
>>> x = 1.236
>>> round(x, 2)
1.24

>>> "%o %x %X" % (64, 64, 255)
'100 40 FF'
>>> hex(255), int('0xff', 16), eval('0xFF')
('0xff', 255, 255)
>>> ord('s'), chr(115)
(115, 's')
```

Formatting with dictionaries

```
>>> D = {'xx': 1, 'yy': 2}
>>> "%(xx)d => %(yy)s" % D
'1 => 2'
```

```

>>> aa = 3
>>> bb = 4
>>> "%(aa)d => %(bb)s" % vars()
'3 => 4'

>>> reply = """
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40}
>>> print reply % values

Greetings...
Hello Bob!
Your age squared is 40

```

Template formatting (2.4+)

```

>>> ('%(page)i: %(title)s' %
      {'page':2, 'title': 'The Best of Times'})
'2: The Best of Times'

>>> import string
>>> t = string.Template('$page: $title')
>>> t.substitute({'page':2, 'title': 'The Best of Times'})
'2: The Best of Times'

>>> s = string.Template('$who likes $what')
>>> s.substitute(who='bob', what=3.14)
'bob likes 3.14'
>>> s.substitute(dict(who='bob', what='pie'))
'bob likes pie'

```

Common string tools

```

>>> S = "spammify"
>>> S.upper()                # convert to uppercase
'SPAMMIFY'

>>> S.find("mm")             # return index of substring
3

>>> int("42"), str(42)       # convert from/to string
(42, '42')

>>> S.split('mm')            # splitting and joining
['spa', 'ify']
>>> 'XX'.join(S.split("mm"), "XX")
'spaXXify'

```

Example: replacing text

```
# replace method

>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'

>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGG')          # replace all
'xxxxEGGxxxxEGGxxxx'

# finding and slicing

>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')             # search for position
>>> where                             # occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'

exploding to/from list

>>> S = 'spammy'
>>> L = list(S)                        # explode to list
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
>>> L[3] = 'x'                         # multiple in-place changes
>>> L[4] = 'x'                         # cant do this for strings
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
>>> S = ''.join(L)                     # implode back to string
>>> S
'spaxxy'
```

Example: parsing with slices

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]                  # columns at fixed offsets
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

Example: parsing with splits

```
>>> line = 'aaa bbb ccc'              # split around whitespace
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

```
>>> line = 'bob,hacker,40'      # split around commas
>>> line.split(',')
['bob', 'hacker', '40']
```

Generic type concepts

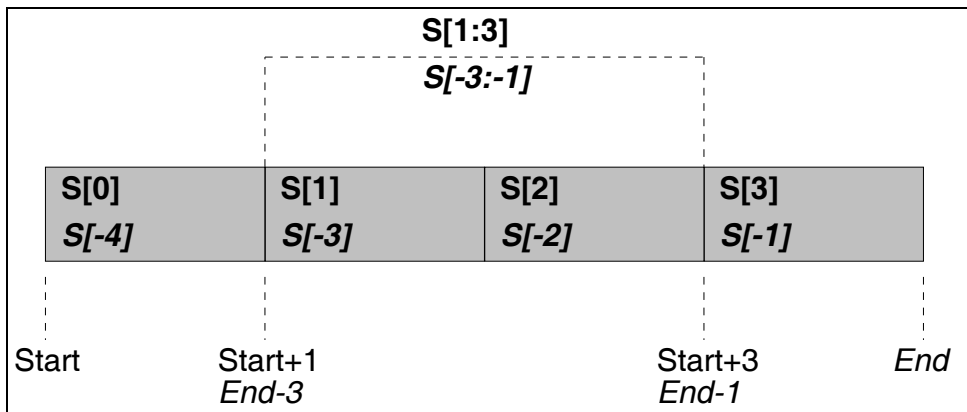
- ◆ Types share operation sets by categories
- ◆ Numbers support addition, multiplication, . . .
- ◆ Sequences support indexing, slicing, concatenation, . . .
- ◆ Mappings support indexing by key, . . .
- ◆ Mutable types can be changed in place
- ◆ Strings are ‘immutable sequences’

Concatenation and repetition

- ◆ *‘X + Y’ makes a new sequence object with the contents of both operands*
- ◆ *‘X * N’ makes a new sequence object with N copies of the sequence operand*

Indexing and slicing

- ◆ **Indexing**
 - ◆ *Fetches components via offsets: zero-based*
 - ◆ *Negative indexes: adds length to offset*
 - ◆ *S[0] is the first item*
 - ◆ *S[-2] is the second from the end (4 - 2)*
 - ◆ *Also works on mappings, but index is a key*
- ◆ **Slicing**
 - ◆ *Extracts contiguous sections of a sequence*
 - ◆ *Slices default to 0 and the sequence length if omitted*
 - ◆ *S[1:3] fetches from offsets 1 upto but not including 3*
 - ◆ *S[1:] fetches from offsets 1 through the end (length)*
 - ◆ *S[:-1] fetches from offsets 0 upto but not including last*
 - ◆ *S[l:j:k] newer, l to j by k, k is a stride/step (S[::2])*



Lists

- ◆ Arrays of object references
- ◆ Access by offset
- ◆ Variable length, heterogeneous, arbitrarily nestable
- ◆ Category: ‘mutable sequence’
- ◆ Ordered collections of arbitrary objects

Common list operations

Operation	Interpretation
<code>L1 = []</code>	an empty list
<code>L2 = [0, 1, 2, 3]</code>	4-items: indexes 0..3
<code>['abc', ['def', 'ghi']]</code>	nested sublists

<code>L2[i], L2[i:j], len(L2)</code>	index, slice, length
<code>L1 + L2, L2 * 3</code>	concatenate, repeat
<code>L1.sort(), L2.append(4)</code>	methods: sort, grow
<code>del L2[k], L2[i:j] = []</code>	shrinking
<code>L2[i:j] = [1,2,3]</code>	slice assignment
<code>range(4), xrange(0, 4)</code>	make integer lists
<code>for x in L2, 3 in L2</code>	iteration/membership

Lists in action

```
% python
>>> [1, 2, 3] + [4, 5, 6]      # concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                # repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Indexing and slicing

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]
'SPAM!'
>>> L[1:]
['Spam', 'SPAM!']
```

Changing lists in-place

```
>>> L[1] = 'eggs'              # index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']   # slice assignment
>>> L                          # replace items 0,1
['eat', 'more', 'SPAM!']
>>> L.append('please')         # append method call
>>> L
['eat', 'more', 'SPAM!', 'please']
```

- ◆ *Only works for ‘mutable’ objects: not strings*
- ◆ *Index assignment replaces an object reference*
- ◆ *Slice assignment deletes a slice and inserts new items*

- ◆ *Append method inserts a new item on the end ('realloc')*

Preview: iteration/membership

```
>>> for x in L: print x,
...
eat more SPAM! please
```

Example: 2-dimensional array

```
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
...
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
```

Dictionaries

- ◆ Tables of object references
- ◆ Access by key, not offset (hash-tables)
- ◆ Variable length, heterogeneous, arbitrarily nestable
- ◆ Category: 'mutable mappings' (not a sequence)
- ◆ Unordered collections of arbitrary objects

Common dictionary operations

<i>Operation</i>	<i>Interpretation</i>
<code>d1 = {}</code>	empty dictionary

<code>d2 = {'spam': 2, 'eggs': 3}</code>	2 items
<code>d3 = {'food': {'ham': 1, 'egg': 2}}</code>	nesting
<code>d2['eggs'], d3['food']['ham']</code>	indexing by key
<code>d2.has_key('eggs'), d2.keys()</code>	methods
<code>d2.get('eggs', default)</code>	default values
<code>len(d1)</code>	length (entries)
<code>d2[key] = new, del d2[key]</code>	adding/changing

Dictionaries in action

```
% python
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2['spam']
2
>>> len(d2)                                # number entries
3
>>> d2.keys()                              # list of keys
['eggs', 'spam', 'ham']
```

Changing dictionaries

```
>>> d2['ham'] = ['grill', 'bake', 'fry']
>>> d2
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> del d2['eggs']
>>> d2
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

Making dictionaries

literals

```
>>> D = {'name': 'Bob', 'age': 42, 'job': 'dev'}
>>> D
{'job': 'dev', 'age': 42, 'name': 'Bob'}
```

keywords

```
>>> D = dict(name='Bob', age=42, job='dev')
>>> D
```

```

{'job': 'dev', 'age': 42, 'name': 'Bob'}

# field by field

>>> D = {}
>>> D['name'] = 'Bob'
>>> D['age'] = 42
>>> D['job'] = 'dev'
>>> D
{'job': 'dev', 'age': 42, 'name': 'Bob'}

# zipped keys/values

>>> pairs = zip(['name', 'age', 'job'], ('Bob', 42, 'dev'))
>>> pairs
[('name', 'Bob'), ('age', 42), ('job', 'dev')]
>>> D = dict(pairs)
>>> D
{'job': 'dev', 'age': 42, 'name': 'Bob'}

# key lists

>>> D = dict.fromkeys(['name', 'age', 'job'], '?')
>>> D
{'job': '?', 'age': '?', 'name': '?'}

```

A language table

```

>>> table = {'Perl': 'Larry Wall',
...          'Tcl': 'John Ousterhout',
...          'Python': 'Guido van Rossum' }
...
>>> language = 'Python'
>>> creator = table[language]
>>> creator
'Guido van Rossum'
>>> for lang in table.keys(): print lang,
...
Tcl Python Perl

```

Dictionary usage notes

- ◆ *Sequence operations don't work!*
- ◆ *Assigning to new indexes adds entries*
- ◆ *Keys need not always be strings*

Example: simulating auto-grown lists

```
>>> L = []                                # L=[0]*100 would help
>>> L[99] = 'spam'
IndexError: list assignment index out of range

>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

Example: dictionary-based “records”

```
>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age'] = 40
>>> rec['job'] = 'trainer/writer'
>>>
>>> print rec['name']
mel

>>> mel = {'name': 'Mark',
...        'jobs': ['trainer', 'writer'],
...        'web': 'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip': 80503}}

>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'
>>> mel['home']['zip']
80503
```

Example: dictionary-based sparse matrix

```
>>> Matrix = {}
>>> Matrix[(2,3,4)] = 88                # tuple key is coordinates
>>> Matrix[(7,8,9)] = 99

>>> X = 2; Y = 3; Z = 4                  # ; separates statements
>>> Matrix[(X,Y,Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}

>>> Matrix.get((0, 1, 2), 'Missing')
'Missing'
```

Tuples

- ◆ Arrays of object references
- ◆ Access by offset
- ◆ Fixed length, heterogeneous, arbitrarily nestable
- ◆ Category: ‘immutable sequences’ (can’t be changed)
- ◆ Ordered collections of arbitrary objects

Common tuple operations

<i>Operation</i>	<i>Interpretation</i>
<code>()</code>	an empty tuple
<code>T1 = (0,)</code>	a one-item tuple
<code>T2 = (0, 1, 2, 3)</code>	a 4-item tuple
<code>T2 = 0, 1, 2, 3</code>	another 4-item tuple
<code>T3 = ('abc', ('def', 'ghi'))</code>	nested tuples
<code>T1[i], t1[i:j], len(t1)</code>	index, slice, length
<code>T1 + t2, t2 * 3</code>	concatenate, repeat
<code>for x in t2, 3 in t2</code>	iteration/membership

Tuples in action

```
>>> T1 = (1, 'spam')
>>> T2 = (2, 'ni')

>>> T1 + T2
(1, 'spam', 2, 'ni')

>>> T1 * 4
(1, 'spam', 1, 'spam', 1, 'spam', 1, 'spam')

>>> T2[1]
```

```
'ni'
>>> T2[1:]
('ni',)
```

Why lists *and* tuples?

- ◆ *Immutability provides integrity*
- ◆ *Some built-in operations require tuples (argument lists)*
- ◆ *Guido is a mathematician: sets versus data structures*

Files

- ◆ A wrapper around C's "stdio" file system
- ◆ The builtin '*open*' function returns a file object
- ◆ File objects export methods for file operations
- ◆ Files are not sequences or mappings (methods only)
- ◆ Files are a built-in C extension type

Common file operations

<i>Operation</i>	<i>Interpretation</i>
<code>O = open('/tmp/spam', 'w')</code>	create output file
<code>I = open('data', 'r')</code>	create input file
<code>I.read()</code> , <code>I.read(1)</code>	read file, byte
<code>I.readline()</code> , <code>I.readlines()</code>	read line, lines list
<code>O.write(S)</code> , <code>O.writelines(L)</code>	write string, lines
<code>O.close()</code>	manual close (or on free)

Files in action

more at the end of the next section

```

>>> newfile = open('test.txt', 'w')
>>> newfile.write(('spam' * 5) + '\n')
>>> newfile.close()

>>> myfile = open('test.txt')
>>> text = myfile.read()
>>> text
'spamspamspamspamspam\n'

```

Related Python tools (day 2 or 3)

- ◆ *Descriptor based files: os module*
- ◆ *DBM keyed files*
- ◆ *Persistent object shelves*
- ◆ *Pipes, fifos*

General object properties

Type categories revisited

- ◆ *Objects share operations according to their category*
- ◆ *Only mutable objects may be changed in-place*

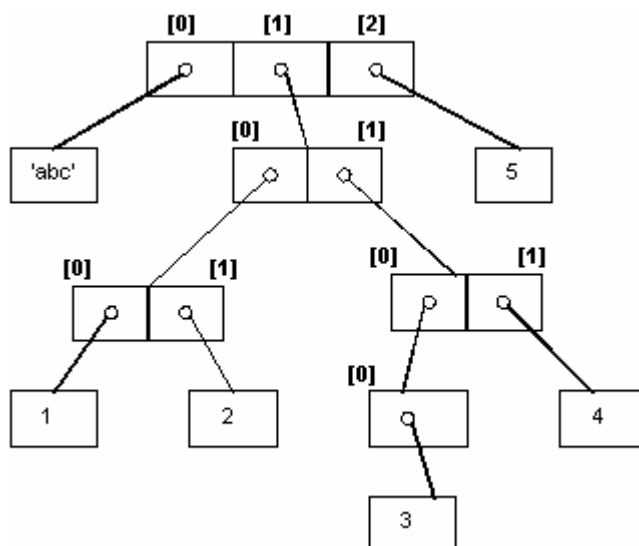
<i>Object type</i>	<i>Category</i>	<i>Mutable?</i>
Numbers	Numeric	No
Strings	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	n/a

Generality

- ◆ Lists, dictionaries, and tuples can hold any kind of object
- ◆ Lists, dictionaries, and tuples can be arbitrarily nested
- ◆ Lists and dictionaries can dynamically grow and shrink

Nesting example

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```



Shared references

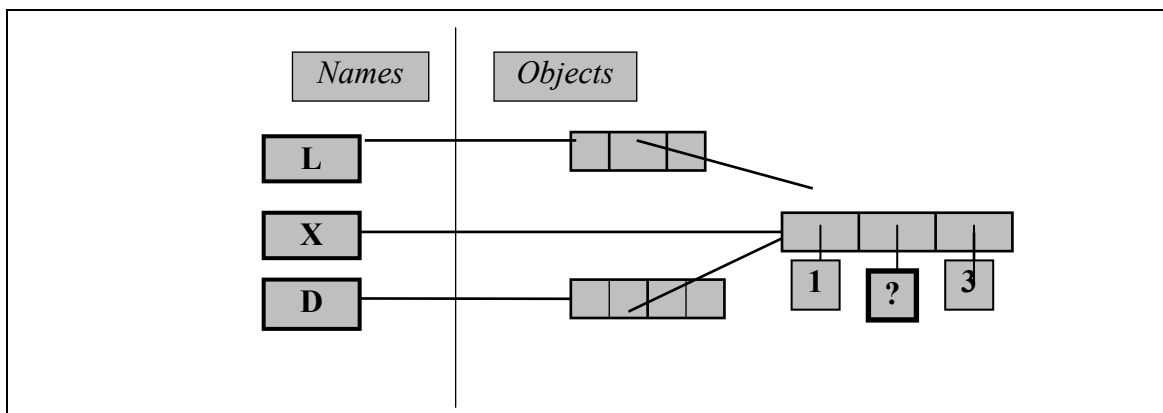
- ◆ Assignments always create references to objects
- ◆ Can generate shared references to the same object
- ◆ Changing a mutable object impacts all references
- ◆ To avoid effect: make copies with `X[:]`, `list(X)`, etc.

◆ Tip: distinguish between names and objects!

◆ ***Names have no "type", but objects do***

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']
>>> D = {'x':X, 'y':2}

>>> X[1] = 'surprise'      # changes all 3 references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```



Equality and truth

- ◆ Applied recursively for nested data structures
- ◆ ‘is’ tests identity (object address)
- ◆ True: non-zero number or non-empty data structure
- ◆ “None” is a special empty/false object

```
>>> L1 = [1, ('a', 3)]          # same value, unique objects
>>> L3 = [1, ('a', 3)]
>>> L1 == L3, L1 is L3         # equivalent?, same object?
(True, False)                  # (True==1, False==0)
```

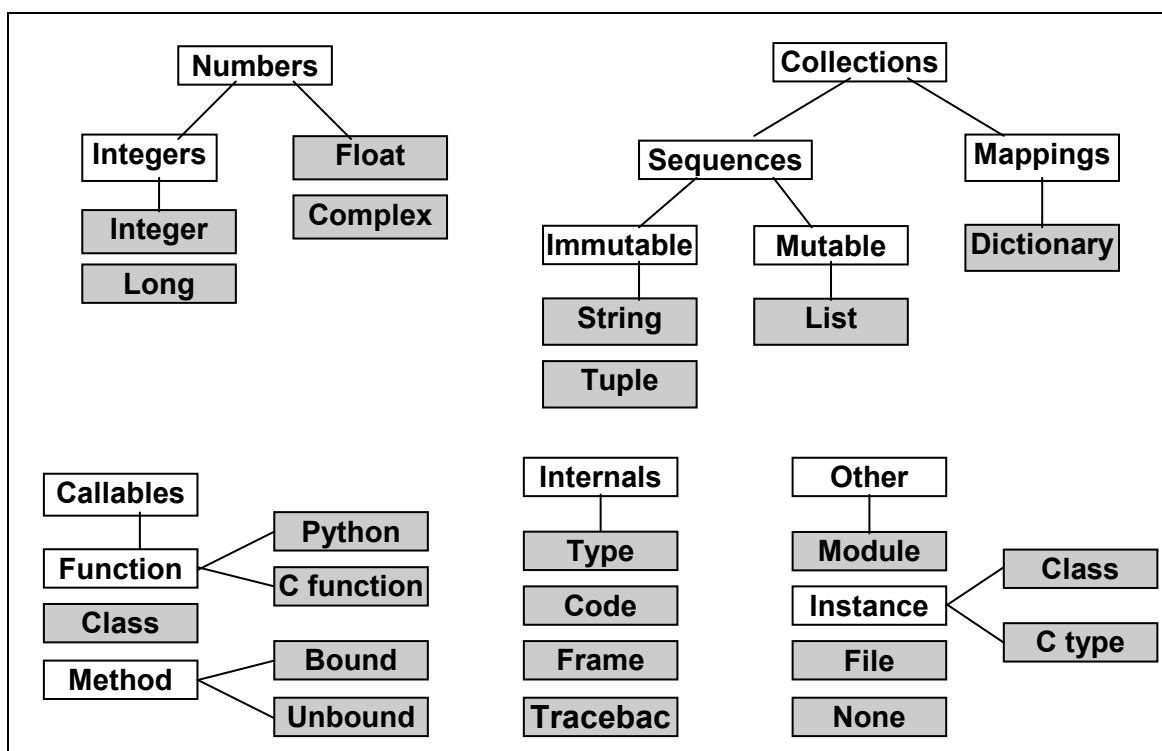
Other comparisons

- ◆ Applied recursively for nested data structures
- ◆ Strings compared lexicographically
- ◆ Lists and tuples compared depth-first, left-to-right
- ◆ Dictionaries compared by sorted (key, value) lists

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2
(False, False, True)
```

Summary: Python's type hierarchies

- ◆ Everything is an 'object' type in Python: "first class"
- ◆ Types are objects too: "type(X)" returns type object of X
- ◆ Preview: C extension modules and types use same mechanisms as Python types



How to break your code's flexibility...

```

>>> L = [1, 2, 3]
>>> if type(L) == type([]):
    print 'yes'

yes
>>> if type(L) == list:
    print 'yes'

```

```
yes
>>> if isinstance(L, list):
    print 'yes'

yes
```

Newer types

Decimal (decimal module): 2.4, see above

Boolean (bool, True, False): 2.3-ish, see next section

Sets: 2.4 (module in 2.3)

```
>>> x = set('abcde')
>>> y = set('bdxyz')
>>> x
set(['a', 'c', 'b', 'e', 'd'])

>>> 'e' in x                                # membership
True

>>> x - y                                    # difference
set(['a', 'c', 'e'])

>>> x | y                                    # union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                                    # intersection
set(['b', 'd'])
```

Built-in type gotchas

◆ Assignment creates references, not copies

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']
>>> M
['X', [1, 2, 3], 'Y']
>>> L[1] = 0
>>> M
['X', [1, 0, 3], 'Y']
```

◆ Repetition adds 1-level deep

```
>>> L = [4, 5, 6]
>>> X = L * 4                                # like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4                              # [L] + [L] + ... = [L, L, ...]
```

```

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
>>> L[1] = 0
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]

```

◆ Cyclic structures can't be printed (till 1.5.1)

```

>>> L = ['hi.']; L.append(L)    # append reference to self
>>> L                          # loop! (ctrl-c breaks)

```

◆ Immutable types can't be changed in-place

```

T = (1, 2, 3)
T[2] = 4          # error!
T = T[:2] + (4,)  # okay: (1, 2, 4)

```

Lab Session 2

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

4. Basic Statements

Python program structure

- ◆ Programs are composed of modules
- ◆ Modules contain statements
- ◆ **Statements** contain expressions: logic
- ◆ Expressions create and process objects

<i>Statement</i>	<i>Examples</i>
Assignment	<code>curly, moe, larry = 'good', 'bad', 'ugly'</code>
Calls	<code>stdout.write("spam, ham, toast\n")</code>
Print	<code>print 1, "spam", 4, 'u',</code>
If/elif/else	<code>if "python" in text: mail(poster, spam)</code>
For/else	<code>for peteSake in spam: print peteSake</code>
While/else	<code>while 1: print 'spam',i; i=i+1</code>
Pass	<code>while 1: pass</code>
Break, Continue	<code>while 1: break</code>
Try/except/finally	<code>try: spam() except: print 'spam error'</code>
Raise	<code>raise overWorked, cause</code>
Import, From	<code>import chips; from refrigerator import beer</code>
Def, Return, Yield	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
Class	<code>class subclass(superclass): staticData = []</code>
Global	<code>def function(): global x, y; x = 'new'</code>
Del	<code>del spam[k]; del spam[i:j]; del spam.attr</code>
Exec	<code>Exec "import " + moduleName in gdict, ldict</code>
Assert	<code>assert name != "", "empty name field"</code>

General concepts

Python syntax

- ◆ No variable/type declarations
- ◆ No braces or semicolons
- ◆ The “what you see is what you get” of languages

Python assignment

- ◆ Assignments create object references
- ◆ Names are created when first assigned
- ◆ Names must be assigned before being referenced

C++/Java:

```
if (x) {
    x = y + z;    // braces, semicolons, parens
}
```

Python:

```
if x:
    x = y + z      # indented blocks, end of line, colon
```

Assignment

- ◆ ‘=’ assigns object references to names or components
- ◆ Implicit assignments: *import, from, def, class, for*, calls

<i>Operation</i>	<i>Interpretation</i>
<code>spam = 'SPAM'</code>	basic form
<code>spam, ham = 'yum', 'YUM'</code>	tuple assignment
<code>[spam, ham] = ['yum', 'YUM']</code>	list assignment
<code>a, b, c, d = 'spam'</code>	sequence assign
<code>spam = ham = 'lunch'</code>	multiple-target
<code>spam += 42; ham *= 12</code>	Augmented (2.0)

Variable name rules

- ◆ ('_' or letter) + (any number of letters, digits, '_'s)
- ◆ Case matters: 'SPAM' is not 'spam'
- ◆ But can't use reserved words:
- ◆ + “yield”, for generators (2.3 and later)
- ◆ + “with” and “as” for context managers (2.6, optional in 2.5)

and	assert	break	class
continue	def	del	elif
else	except	exec	finally
for	from	global	If
import	in	is	lambda
not	or	pass	print
raise	return	try	while

Expressions

- ◆ Useful for calls, and interactive prints
- ◆ Expressions can be used as statements
- ◆ Statements cannot be used as expressions (‘=’)

<i>Operation</i>	<i>Interpretation</i>
<code>spam(eggs, ham)</code>	function calls
<code>spam.ham(eggs)</code>	method calls
<code>spam</code>	interactive print
<code>spam < ham and ham != eggs</code>	compound expr's
<code>spam < ham < eggs</code>	range tests

- ◆ Python 2.0 “list comprehension” expressions (covered in Functions)
 - ◆ Similar to map/lambda combination (result=[0,1,4,9])
- ```
[i**2 for i in range(4)] ...like... map((lambda x: x**2), range(4))
```

## Print

- ◆ ‘print’ statement writes objects to the ‘stdout’ stream
- ◆ File object ‘write’ methods write strings to files
- ◆ Adding a trailing comma suppresses line-feed
- ◆ Reset ‘sys.stdout’ to catch print output

| <i>Operation</i>                     | <i>Interpretation</i>                    |
|--------------------------------------|------------------------------------------|
| <code>print spam, ham</code>         | print objects to <code>sys.stdout</code> |
| <code>print spam, ham,</code>        | don't add linefeed at end                |
| <code>print&gt;&gt;file, spam</code> | Python 2.0: not to <code>stdout</code>   |

## The Python 'Hello world' program

- Expression results don't need to be printed at top-level

```
>>> print 'hello world'
hello world
>>> 'hello world'
'hello world'
```

- The hard way

```
>>> x = 'hello world'
>>> import sys
>>> sys.stdout.write(str(x) + '\n')
```

- `sys.stdout` can be assigned

```
>>> sys.stdout = open('log', 'a') # or a class with .write
>>> print x
```

## If selections

- ◆ Python's main selection construct
- ◆ No 'switch': via `if/elif/else`, dictionaries, or lists

## General format

```
if <test1>:
 <statements1>
elif <test2>: # optional elif's
 <statements2>
else: # optional else
 <statements3>
```

## Examples

```
>>> if 3 > 2:
... print 'yep'
...
yep
```

```
>>> x = 'killer rabbit'
>>> if x == 'bunny':
... print 'hello little bunny'
... elif x == 'bugs':
... print "what's up doc?"
... else:
... print 'Run away!... Run away!...'
...
Run away!... Run away!...
```

```
>>> choice = 'ham'
>>> print {'spam': 1.25, # dictionary switch
... 'ham': 1.99,
... 'eggs': 0.99,
... 'bacon': 1.10}[choice]
1.99
```

```
with actions
{'spam': (lambda: ...),
 'ham': (lambda: ...),
 ...}[choice]()
```

## Python syntax rules

- ◆ Compound statements = header, ‘:’, indented statements
- ◆ Block and statement boundaries detected automatically
- ◆ Comments run from “#” through end of line
- ◆ Documentation strings at top of file, class, function

### Block delimiters

- ◆ Block boundaries detected by line indentation
- ◆ Indentation is any combination of spaces and tabs
- ◆ Tabs = N spaces up to multiple of 8 (but don’t mix)

### Statement delimiters

- ◆ Statement normally end at end-of-line, or ‘;’
- ◆ Statements may span lines if open syntactic pair: ( ), { }, [ ]
- ◆ Statements may span lines if end in backslash (outdated feature)
- ◆ Some string constants span lines too (triple-quotes)

### Special cases

```
L = ["Good",
 "Bad",
 "Ugly"] # open pairs may span lines

x = 1; y = 2; print x # more than 1 simple statement

if 1: print 'hello' # simple statement on header line
```

## Nesting code blocks

```
x = 1 # block0
if x:
 y = 2 # block1
 if y:
 print 'block2'
 print 'block1'
print 'block0'
```

## Documentation Sources Interlude

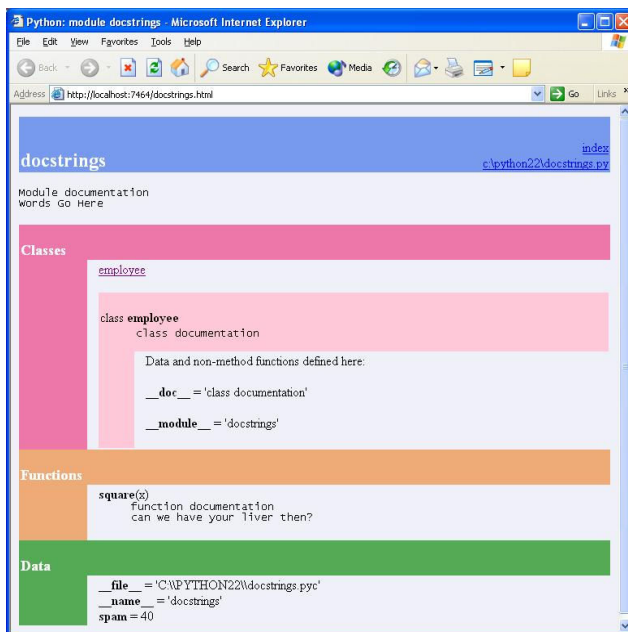
| Form                             | Role                                         |
|----------------------------------|----------------------------------------------|
| # comments                       | In-file documentation                        |
| The dir function                 | Lists of attributes available on objects     |
| Docstrings: <code>__doc__</code> | In-file documentation attached to objects    |
| <b>PyDoc</b> : The help function | Interactive help for objects                 |
| <b>PyDoc</b> : HTML reports      | Module documentation in a browser            |
| Standard manual set              | Official language and library descriptions   |
| Web resources                    | Online tutorial, examples, and so on         |
| Published books                  | Commercially-available texts (see Resources) |

```
>>> import sys
>>> dir(sys) # also works on
types, objects, etc.
['_displayhook_', '__doc__', '__excepthook__', '__name__', ...]
```

```
>>> print sys.__doc__
This module provides access to some objects ...
```

```
>>> help(sys)
Help on built-in module sys: ...
```

```
Start/Python24/ModuleDocs (or pydocgui.pyw):
```



```
File: docstrings.py
"""
Module documentation
Words Go Here
"""

spam = 40

def square(x):
 """
 function documentation
 can we have your liver then?
 """
 return x **2
```

## Truth tests revisited

- ◆ True = non-zero number, or non-empty object
- ◆ Comparisons operators return “True” (1) or “False” (0)
- ◆ Boolean operators short-circuit
- ◆ Boolean operators return an operand *object*

| Object | Value |
|--------|-------|
| "spam" | true  |
| ""     | false |
| []     | false |
| {}     | false |
| 1      | true  |
| 0.0    | false |
| None   | false |

## Examples

```

>>> 2 < 3, 3 < 2 # return True (1) or False (0)
(True, False)

>>> 2 or 3, 3 or 2 # return left operand if true
(2, 3) # else return right operand (T|F)
>>> [] or 3
3
>>> [] or {}
{}

>>> 2 and 3, 3 and 2 # return left operand if false
(3, 2) # else return right operand (T|F)
>>> [] and {}
[]
>>> 3 and []
[]

```

## C's ternary operator in Python

Prior to 2.5: `X = (A and B) or C`

New in 2.5: `X = B if A else C`

## Boolean type (2.3+)

*bool is a subclass of int*

*bool has two instances: True and False*

*True, False are 1, 0 but print differently*

```

>>> 1 > 0
True

>>> True == 1, True is 1
(True, False)

```



```
>>> True + 1
2
```

## While loops

- ◆ Python's most general iteration construct
- ◆ One of two looping statements: *while*, *for*
- ◆ Implicit looping tools: *map*, *reduce*, *filter*, *in*, *list comprehensions*

### General format

```
while <test>:
 <statements>
else:
 <statements2> # optional else
 # run if didn't exit with break
```

### Examples

```
>>> while True:
... print 'Type Ctrl-C to stop me!'
```

```
>>> count = 5
>>> while count:
... print count,
... count -= 1
...
5 4 3 2 1
```

```
>>> x = 'spam'
>>> while x:
... print x,
... x = x[1:] # strip first char off x
...
spam pam am m
```

```
>>> a=0; b=10
>>> while a < b: # one way to code counter loops
... print a,
... a = a+1
```

```
...
0 1 2 3 4 5 6 7 8 9
```

## ***Break, continue, pass, and the loop else***

- ◆ *break*       jumps out of the closest enclosing loop
- ◆ *continue*   jumps to the top of the closest enclosing loop
- ◆ *pass*        does nothing: an empty statement placeholder
- ◆ *loop else*        run if loop exits normally: without a 'break'

### **General loop format**

```
while <test>:
 <statements>
 if <test>: break # exit loop now, skip else
 if <test>: continue # go to top of loop now
else:
 <statements> # if we didn't hit a 'break'
```

### **Examples**

#### ◆ **Pass: an infinite loop**

```
while True: pass # ctrl-C to stop!
```

#### ◆ **Continue: print even numbers**

- ◆ *Avoids statement nesting (but use sparingly!)*

```
x = 10
while x:
 x = x-1
 if x % 2 != 0: continue # odd?--skip
 print x,
```

### ◆ Break: find factors

- ◆ *Avoids search status flags*

```
x = y / 2
while x > 1:
 if y % x == 0: # remainder
 print y, 'has factor', x
 break # skip else
 x = x-1
else: # normal exit
 print y, 'is prime'
```

## For loops

- ◆ A general sequence iterator
- ◆ Works on strings, lists, tuples
- ◆ Replaces most ‘counter’ style loops
- ◆ Repeatedly indexes object until IndexError detected
- ◆ Preview: also works on Python classes and C types

### General format

```
for <target> in <object>: # assign object items to target
 <statements>
 if <test>: break # exit loop now, skip else
 if <test>: continue # go to top of loop now
else:
 <statements> # if we didn't hit a 'break'
```

### Examples

```

>>> for x in ["spam", "eggs", "spam"]:
... print x,
...
spam eggs spam

>>> prod = 1
>>> for i in (1, 2, 3, 4): prod *= i # tuples
...
>>> prod
24

>>> S = 'spam'
>>> for c in S: print c # strings
...
s
p
a
m

```

### Works on any iterable object: files, dicts

```

>>> for line in open('data.txt'):
... print line.upper() # calls .next(), catches exc

>>> for key in D:
... print key, D[key]

```

## Loop coding techniques

- ◆ *for* subsumes most counter loops
- ◆ *range* generates a list of integers to iterate over
- ◆ *xrange* similar, but doesn't create a real list
- ◆ *avoid range, and the temptation to count things!*

### # The easy (and fast) way

```

>>> X = 'spam'
>>> for item in X: print item, # step through items
...
s p a m

```

#### # The hard way: a C-style for loop

```
>>> i = 0
>>> while i < len(X): # manual while indexing
... print X[i],; i += 1
...
s p a m
```

#### # Range and fixed repetitions

```
>>> range(5), range(2, 5)
([0, 1, 2, 3, 4], [2, 3, 4])

>>> for i in range(4): print 'A shrubbery!'
...
A shrubbery!
A shrubbery!
A shrubbery!
A shrubbery!
```

#### # Using range to generate offsets (not items!)

```
>>> X
'spam'
>>> len(X)
4
>>> range(len(X))
[0, 1, 2, 3]

>>> for i in range(len(X)): print X[i], # step through offsets
...
s p a m
```

#### # Using range and slicing for non-exhaustive traversals

```
>>> range(2,10,2)
[2, 4, 6, 8]

>>> S = 'abcdefghijk'
>>> for i in range(0,len(S),2): print S[i],
...
a c e g i k
```

#### in recent releases...

```
>>> for c in S[::-2]: print c, # S[::-1] reverses
...
a c e g i k
```

#### # Using range and enumerate to change a list in-place

```
>>> L = [1, 2, 3, 4]
>>> for x in L: x += 10
```

```
>>> L
[1, 2, 3, 4]

>>> for i in range(len(L)): L[i] += 10

>>> L
[11, 12, 13, 14]
```

### List comprehensions

```
>>> M = [x + 10 for x in L]
>>> M
[21, 22, 23, 24]

>>> lines = [line.rstrip() for line in open('README.txt')]

>>> [row[1] for row in matrix]

... more in functions section
```

### Enumerate in 2.3+

```
>>> for (i, x) in enumerate(L):
 L[i] = x * 2

>>> L
[22, 24, 26, 28]

>>> enumerate(L)
<enumerate object at 0x00B48440>

>>> list(enumerate(L))
[(0, 22), (1, 24), (2, 26), (3, 28)]

>>> E = enumerate(L)
>>> E.next()
(0, 22)
>>> E.next() # see generators in next section
(1, 24)
```

### # Traversing sequences in parallel with zip

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
>>>
>>> zip(L1,L2)
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>>
>>> for (x,y) in zip(L1, L2):
... print x, y, '--', x+y
```

```
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

#### # Traversing dictionaries by sorted keys

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> Ks = D.keys()
>>> Ks.sort()
>>> for k in Ks: print D[k],

1 2 3
```

#### Sorted in 2.4+

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for k in sorted(D): print D[k],

1 2 3
```

## Comprehensive examples

#### # Common ways to read from files

##### # file creation

```
>>> myfile = open('myfile.txt', 'w')
>>> for i in range(3):
>>> myfile.write(('spam' * (i+1)) + '\n')
>>> myfile.close()
```

##### # all at once

```
>>> print open('myfile.txt').read()
spam
spamspam
spamspamspam
```

##### # line by line

```
>>> myfile = open('myfile.txt')
>>> while True:
```

```

 line = myfile.readline()
 if not line: break
 print line,

spam
spamspam
spamspamspam

all lines at once

>>> for line in open('myfile.txt').readlines():
 print line,

spam
spamspam
spamspamspam

file iterators: line by line

>>> for line in open('myfile.txt'):
 print line,

spam
spamspam
spamspamspam

by byte counts

>>> myfile = open('myfile.txt')
>>> while True:
 line = myfile.read(10)
 if not line: break
 print '[' + line + ']',

[spam
spams] [pam
spamsp] [amspam
]

Summing data file columns

>>> print open('data.txt').read()
001.1 002.2 003.3
010.1 020.2 030.3 040.4
100.1 200.2 300.3

>>> sums = {}
>>> for line in open('data.txt'):
 cols = [float(col) for col in line.split()] # next!
 for pos, val in enumerate(cols):
 sums[pos] = sums.get(pos, 0.0) + val

>>> for key in sorted(sums):
 print key, '=', sums[key]
```



```

0 = 111.3
1 = 222.6
2 = 333.9
3 = 40.4

>>> sums
{0: 111.3, 1: 222.59999999999999, 2: 333.90000000000003,
3: 40.399999999999999}

```

## ***Basic coding gotchas***

- ◆ Don't forget to type a ":" at the end of compound statement headers
- ◆ Be sure to start top-level (unnested) code in column 1
- ◆ Blank lines in compound statements are ignored in files, but end the statement at the interactive prompt
- ◆ Avoid mixing tabs and spaces in indentation, unless you're sure what your editor does with tabs
- ◆ C programmers: you don't need "(" around tests in "if" and "while"; you can't use "{" around blocks
- ◆ In-place change operations like `list.append()` and `list.sort()` don't return a value (really, they return "None"); call them without assigning the result.
- ◆ Add parens to call a function: "`file.close()`" is a call, "`file.close`" is a reference only

## ***Preview: program unit statements***

- ◆ Create and process higher-level program components

| <i>Unit</i> | <i>Role</i> |
|-------------|-------------|
|-------------|-------------|

|            |                                          |
|------------|------------------------------------------|
| Functions  | procedural units                         |
| Modules    | code/data packages                       |
| Exceptions | errors and special cases                 |
| Classes    | new objects                              |
| C modules  | optimization, customization, integration |

## ***Lab Session 3***

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 5. Functions

### Why use functions?

- ◆ Code reuse
- ◆ Procedural decomposition
- ◆ Alternative to cut-and-paste: redundancy

### Function topics

- ◆ The basics
- ◆ Scope rules
- ◆ Argument matching modes
- ◆ Odds and ends
- ◆ Design concepts
- ◆ Functions are objects
- ◆ Function gotchas

### ***Function basics***

- ◆ *def* is an executable statement; usually run during import
- ◆ *def* creates a function object and assigns to a name
- ◆ *return* sends a result object back to the caller
- ◆ Arguments are passed by object reference (assignment)
- ◆ Arguments, return types, and variables are not declared
- ◆ *Polymorphism*: code to object interfaces, not datatypes

## General form

```
def <name>(arg1, arg2,... argN):
 <statements>
 return <value>
```

## Definition

```
>>> def times(x, y): # create and assign function
... return x * y # body executed when called
...
```

## Calls

```
>>> times(2, 4) # arguments in parenthesis
8
>>> times('Ni', 4) # functions are 'typeless'
'NiNiNiNi'
```

## ➔ “Polymorphism”

## Example: intersecting sequences

### ◆ Definition

```
def intersect(seq1, seq2):
 res = [] # start empty
 for x in seq1: # scan seq1
 if x in seq2: # common item?
 res.append(x) # add to end
 return res
```

### ◆ Calls

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"
```

```
>>> intersect(s1, s2) # strings
['S', 'A', 'M']

>>> intersect([1, 2, 3], (1, 4)) # mixed types
[1]
```

## Scope rules in functions

- ◆ Enclosing module is a ‘global’ scope
- ◆ Each call to a function is a new ‘local’ scope
- ◆ Assigned names are *local*, unless declared “global”
- ◆ All other names are *global* or *builtin*
- ◆ New in 2.2: enclosing function locals (if any) searched before global

### Name resolution: the “LEGB” rule

- **References search up to 4 scopes:**
  1. **Local** (function)
  2. **Enclosing functions** (if any)
  3. **Global** (module)
  4. **Builtin** (`__builtin__`)
- **Assignments create or change local names by default**
- **“global” declarations map assigned names to module**

### Example

- ◆ Global names: ‘X’, ‘func’
- ◆ Local names: ‘Y’, ‘Z’
- ◆ Interactive prompt: module ‘`__main__`’

```

X = 99 # X and func assigned in module

def func(Y): # Y and Z assigned in function
 Z = X + Y # X not assigned: global
 return Z

func(1) # func in module: result=100

```

## Enclosing Function Scopes (2.2+)

```

def f1():
 x = 88
 def f2():
 print x # 2.2: x found in enclosing function
 f2()
f1() # prints 88

```

```

def f1():
 x = 88
 def f2(x=x): # before 2.2: pass in values with defaults (ahead)
 print x
 f2()
f1()

```

```

More useful with lambda (ahead)
def func():
 x = 42
 action = (lambda n: x ** n) # 2.2

def func():
 x = 42
 action = (lambda n, x=x: x ** n) # before 2.2

```

## More on “global”

- ◆ ‘global’ means assigned at top-level of a module file
- ◆ Global names must be declared only if assigned
- ◆ Global names may be referenced without being declared

```

y, z = 1, 2 # global variables in module

def all_global():
 global x # declare globals assigned
 x = y + z # no need to declare y,z: 3-scope rule

```

## More on “return”

- ◆ Return sends back an object as value of call
- ◆ Can return multiple arguments in a tuple
- ◆ Can return modified argument name values

```

>>> def multiple(x, y):
... x = 2
... y = [3, 4]
... return x, y
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)
>>> X, L
(2, [3, 4])

```

## More on argument passing

- ◆ Passed by assigning shared object to local name
- ◆ Assigning to argument name doesn't effect caller
- ◆ Changing mutable object argument may impact caller

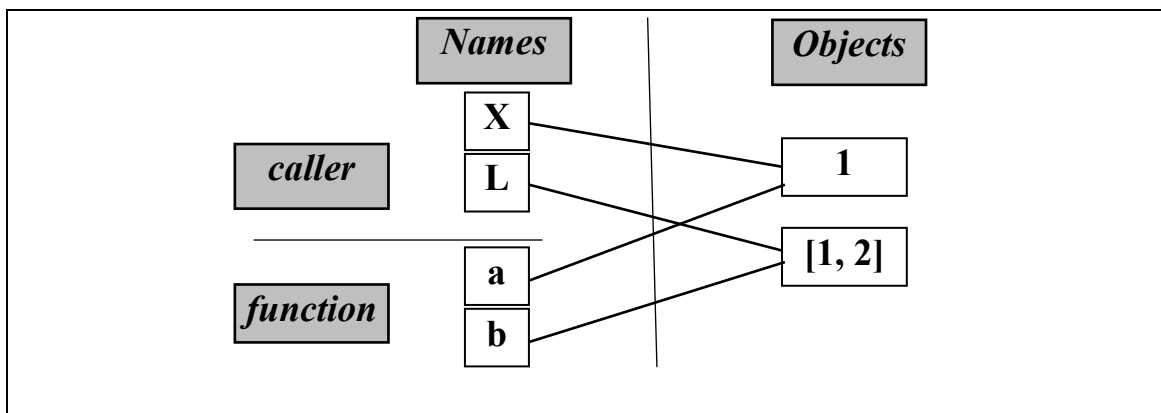
Not pass ‘*by reference*’ (C++), but:

- ◆ ***immutables act like ‘by value’ (C)***
- ◆ ***mutables act like ‘by pointer’ (C)***

```

>>> def changer(a, b):
... a = 2 # changes local name's value only
... b[0] = 'spam' # changes shared object in-place
...
>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
>>> X, L
(1, ['spam', 2])

```



## Special argument matching modes

- ◆ *Positional*      matched left-to-right in header (normal)
- ◆ *Keywords*      matched by name in header
- ◆ *Varargs*      catch unmatched positional or keyword args
- ◆ *Defaults*      header can provide default argument values

| Operation | Location | Interpretation |
|-----------|----------|----------------|
|-----------|----------|----------------|



|                                    |          |                                                               |
|------------------------------------|----------|---------------------------------------------------------------|
| <code>func(value)</code>           | caller   | normal argument: matched by position                          |
| <code>func(name=value)</code>      | caller   | keyword argument: matched by name                             |
| <code>def func(name)</code>        | function | normal argument: matches any by name or position              |
| <code>def func(name=value)</code>  | function | default argument value, if not passed in call                 |
| <code>def func(*name)</code>       | function | matches remaining positional args (tuple)                     |
| <code>def func(**name)</code>      | function | matches remaining keyword args (dictionary)                   |
| <code>func(*args, **kwargs)</code> | caller   | subsumes old <code>apply()</code> : unpack tuple/dict of args |

## Examples

### Positionals and keywords

```
>>> def f(a, b, c): print a, b, c

>>> f(1, 2, 3)
1 2 3

>>> f(c=3, b=2, a=1)
1 2 3

>>> f(1, c=3, b=2)
1 2 3
```

### Defaults

```
>>> def f(a, b=2, c=3): print a, b, c

>>> f(1)
1 2 3
```

```
>>> f(1, 4, 5)
1 4 5

>>> f(1, c=6)
1 2 6
```

## Arbitrary positionals

```
>>> def f(*args): print args

>>> f(1)
(1,)

>>> f(1,2,3,4)
(1, 2, 3, 4)
```

## Arbitrary keywords

```
>>> def f(**args): print args

>>> f()
{}

>>> f(a=1, b=2)
{'a': 1, 'b': 2}

>>> def f(a, *pargs, **kargs): print a, pargs, kargs
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

## Example: min value functions

### Example

- ◆ *Only deals with matching: still passed by assignment*
- ◆ *Defaults retain an object: may change if mutable*

```
def func(spam, eggs, toast=0, ham=0): # first 2 required
 print (spam, eggs, toast, ham)

func(1, 2) # output: (1, 2, 0, 0)
func(1, ham=1, eggs=0) # output: (1, 0, 0, 1)
func(spam=1, eggs=0) # output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3) # output: (3, 2, 1, 0)
```

```
func(1, 2, 3, 4) # output: (1, 2, 3, 4)
```

## Ordering rules

- ◆ *Call*: keyword arguments after non-keyword arguments
- ◆ *Header*: normals, then defaults, then \*name, then \*\*name

## Matching algorithm (see exercise)

1. Assign non-keyword arguments by position
2. Assign keyword arguments by matching names
3. Assign extra non-keyword arguments to \*name tuple
4. Assign extra keyword arguments to \*\*name dictionary
5. Unassigned arguments in header assigned default values

## Odds and ends

- ◆ *lambda* expression creates anonymous functions
- ◆ list comprehensions, *map*, *filter* apply expressions to sequences
- ◆ Generator expressions (2.4+)
- ◆ Generator functions and iterators (new in 2.2, 2.3)
- ◆ *apply* function calls functions with arguments tuple
- ◆ Functions return *None* if they don't use a real *return*

### ◆ Lambda expressions

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
>>> f = lambda x, y, z: x + y + z
```

```
>>> f(2, 3, 4)
9
```

*hint: embedding logic in a lambda body*

```
(A and B) or C
```

```
((A and [B]) or [C])[0] # like if A: B else: C
```

*hint: embedding loops in a lambda body...next topic*

## ◆ List comprehensions (added in 2.0)

```
>>> ord('s')
115
```

```
>>> res = []
>>> for x in 'spam': res.append(ord(x))
...
>>> res
[115, 112, 97, 109]
```

```
>>> map(ord, 'spam') # apply func to sequence
[115, 112, 97, 109]
```

```
>>> [ord(x) for x in 'spam'] # apply expr to sequence
[115, 112, 97, 109]
```

```
adding arbitrary expressions
```

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> map((lambda x: x**2), range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> lines = [line[:-1] for line in open('README.txt')]
>>> lines[:2]
['This is Python version 2.4 alpha 3', '=====']
```

```
adding if tests
```

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

```
>>> filter((lambda x: x % 2 == 0), range(10))
[0, 2, 4, 6, 8]
```

### # advanced usage

```
>>> [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]

>>> [x+y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

>>> res = []
>>> for x in 'abc':
... for y in 'lmn':
... res.append(x+y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

nice for matrixes

>>> M = [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
>>> quad = [M[i][j] for i in (0,1) for j in (0, 1)]
>>> quad
[1, 2, 4, 5]
```

*List comprehensions can become incomprehensible when nested, but map and list comprehensions may be faster than simple for loops*

*(In 2.4, comprehensions are twice as fast as for loops, and for loops are now quicker than map: see middle of [this page](#), and CD's Extras\Misc\timerseqs.py)*

### ◆ Generator expressions (2.4+)

#### # list comprehensions generate entire list in memory

```
>>> squares = [x**2 for x in range(5)]
>>> squares
[0, 1, 4, 9, 16]
```

#### # generator expressions yield 1 result at a time

```
>>> squares = (x**2 for x in range(5))
>>> squares
```

```

<generator object at 0x00B2EC88>
>>> squares.next()
0
>>> squares.next()
1
>>> squares.next()
4
>>> list(squares)
[9, 16]

```

```
iteration contexts automatically call next()
```

```

>>> for x in (x**2 for x in range(5)):
 print x,

```

```
0 1 4 9 16
```

```

>>> sum(x**2 for x in range(5))
30

```

## ◆ Generator functions and iterators

*Generators implement iterator protocol: .next()*

*Retains local scope when suspended*

*Distributes work over time (see also: threads)*

*Related: generator expressions, enumerate function, file iterators*

*In 2.5, caller can pass values to generators: gen.send(X) method, yield is an expression with a value*

```
functions compiled specially when yield
```

```

>>> def gensquares(N):
... for i in range(N): # suspends and resumes itself
... yield i ** 2 # <- return value and resume here later

```

```
generator objects support iteration protocol: .next()
```

```

>>> x = gensquares(10)
>>> x # also retain all local variables between
calls
<generator object at 0x0086C378> # classes define __iter__ to return iter
object
>>> x.next()
0
>>> x.next()
1
>>> x.next()
4
...
>>> x.next()

```

...StopIteration exception raised at end..

# for loops (and others) automatically call .next()

```
>>> for i in gensquares(5): # resume the function each time
... print i, ': ', # print last yielded value
...
0 : 1 : 4 : 9 : 16 :
```

### ◆ Apply built-in and syntax

```
>>> apply(func, (2, 3, 4))
9
>>> apply(f, (2, 3, 4))
9
```

See also Python 2.0+ apply-like call syntax:

```
func(*a, **b) ...like... apply(func, a, b)

>>> def func(a, b, c, d): return a + b + c + d

>>> args1 = (1, 2)
>>> args2 = {'c': 3, 'd': 4}

>>> func(*args1, **args2)
10

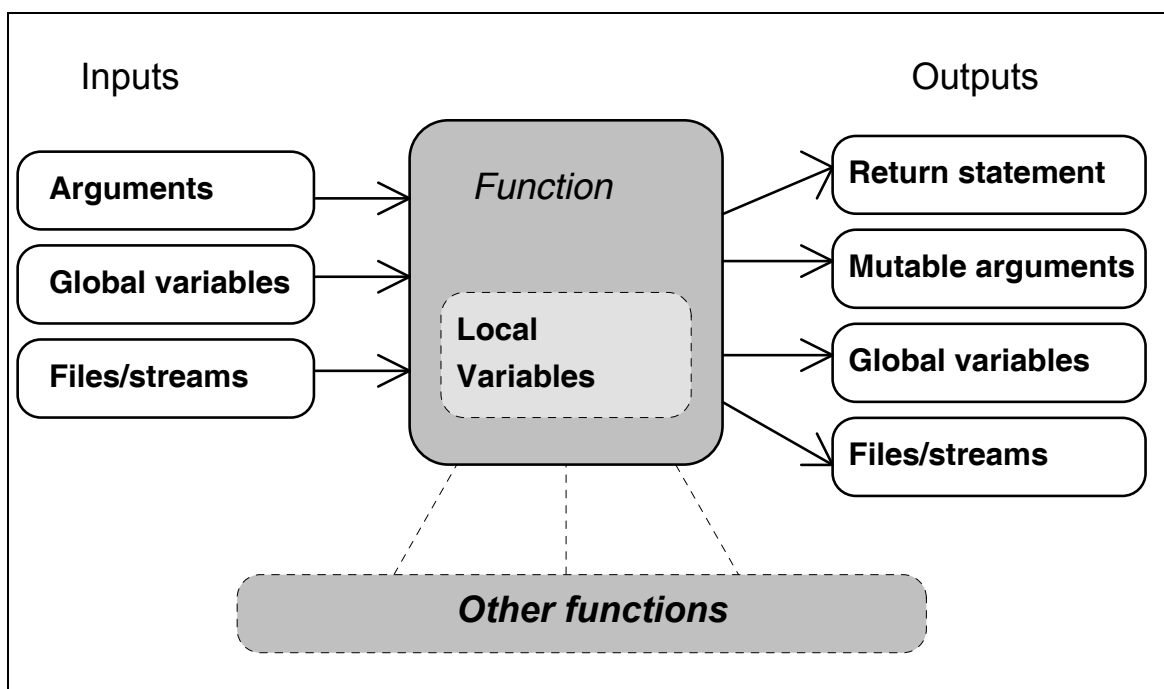
>>> func(1, *(2,), **args2)
10
```

### ◆ Default return values

```
>>> def proc(x):
... print x
...
>>> x = proc('testing 123...')
testing 123...
>>> print x
None
```

## Function design concepts

- ◆ Use global variables only when absolutely necessary
- ◆ Use arguments for input, *'return'* for outputs
- ◆ Don't change mutable arguments unless expected
- ◆ But globals are only state-retention tool without classes
- ◆ But classes depend on mutable arguments (*'self'*)



## ***Functions are objects: indirect calls***

- ◆ Function objects can be assigned, passed, etc.
- ◆ Can call objects generically: function, bound method,...

```
def echo(message): print message
x = echo
x('Hello world!')
```



```
def indirect(func, arg):
 func(arg)
indirect(echo, 'Hello world!')

schedule = [(echo, 'Hello!'), (echo, 'Ni!')]
for (func, arg) in schedule:
 apply(func, (arg,)) # func(arg) works too
```

## File scanners

### file: scanfile.py

```
def scanner(name, function):
 file = open(name, 'r') # create file
 for line in file.readlines():
 function(line) # call function
 file.close()
```

### file: commands.py

```
import string
from scanfile import scanner

def processLine(line):
 print string.upper(line)

scanner("data.txt", processLine) # start scanner
```

## Function gotchas

### Local names are detected statically

*# cant use same name as local+global unless use module*

```
>>> X = 99
>>> def selector(): # X used but not assigned
... print X # X found in global scope
...
>>> selector()
99
```

---

```
>>> def selector():
... print X # does not yet exist!
... X = 88 # X classified as a local name
```

---

```
...
>>> selector()
Traceback (innermost last):
 File "<stdin>", line 1, in ?
 File "<stdin>", line 2, in selector
NameError: X
```

---

```
>>> def selector():
... global X # force X to be global
... print X
... X = 88
...
>>> selector()
99
```

## Mutable defaults created just once

*# to avoid: check for None and set to [] in function body*

```
>>> def grow(A, B=[]):
... B.append(A)
... return B
```

```
>>> grow(1)
[1]
>>> grow(1)
[1, 1]
>>> grow(1)
[1, 1, 1]
```

## Nested functions weren't nested scopes

*# this works as of 2.2 - enclosing function scopes!*

```
>>> def outer(x):
... def inner(i): # assign in outer's local
... print i, # i is in inner's local
... if i: inner(i-1) # not in my local or global!
... inner(x)
...
>>> outer(3)
3
Traceback (innermost last):
 File "<stdin>", line 1, in ?
 File "<stdin>", line 6, in outer
 File "<stdin>", line 5, in inner
NameError: inner
```

---

```
>>> def outer(x):
... global inner
... def inner(i): # assign in enclosing module
... print i,
... if i: inner(i-1) # found in my global scope
... inner(x)
...
>>> outer(3)
3 2 1 0
```

## Use defaults to save references

*# no longer necessary as of 2.2: enclosing function scopes*

```
>>> def outer(x, y):
... def inner():
... return x ** y
... return inner
```

```
>>> x = outer(2, 4)
>>> x()
16
```

---

*# code before 2.2*

```
>>> def outer(x, y):
... def inner(a=x, b=y): # save x,y bindings/objects
... return a**b # from the enclosing scope
... return inner
...
>>> x = outer(2, 4)
>>> x()
16
```

---

```
>>> def outer(x, y):
... return lambda a=x, b=y: a**b
...
>>> y = outer(2, 5)
>>> y()
32
```

## Lab Session 4

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 6. Modules

### Why use modules?

- ◆ Code reuse
- ◆ System name-space partitioning
- ◆ Implementing shared services or data

### Module topics

- ◆ The basics
- ◆ Import variations
- ◆ Reloading modules
- ◆ Design concepts
- ◆ Modules are objects
- ◆ Package imports
- ◆ Odds and ends
- ◆ Module gotchas

### *Module basics*

- |                       |                                                   |
|-----------------------|---------------------------------------------------|
| ◆ Creating modules:   | Python files, C extensions; Java classes (Jython) |
| ◆ Using modules:      | import, from, reload()                            |
| ◆ Module search path: | \$PYTHONPATH                                      |

**file: module1.py**

```
def printer(x): # module attribute
 print x
```

**Module usage**

```
% python
>>> import module1 # get module
>>> module1.printer('Hello world!')
Hello world!

>>> from module1 import printer # get an export
>>> printer('Hello world!')

>>> from module1 import * # get all exports
>>> printer('Hello world!')
```

***Module files are a namespace***

- ◆ A single scope: local==global
- ◆ Module statements run on first import
- ◆ Top-level assignments create module attributes
- ◆ Module namespace: attribute ‘\_\_dict\_\_’, or dir()

**file: module2.py**

```
print 'starting to load...'

import sys
name = 42

def func(): pass

class klass: pass

print 'done loading.'
```

**Usage**

```
>>> import module2
starting to load...
done loading.
```

```

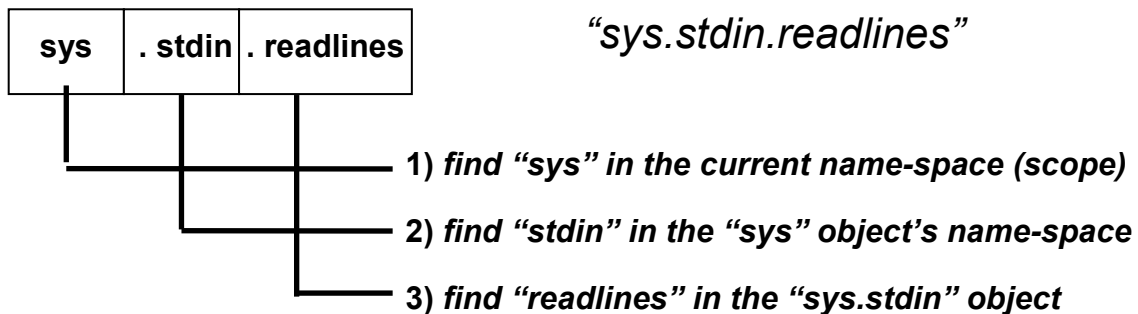
>>> module2.sys
<module 'sys'>
>>> module2.name
42
>>> module2.func, module2.klass
(<function func at 765f20>, <class klass at 76df60>)

>>> module2.__dict__.keys()
['_file_', 'name', '__name__', 'sys', '__doc__', '__builtins__', 'klass',
'func']

```

## Name qualification

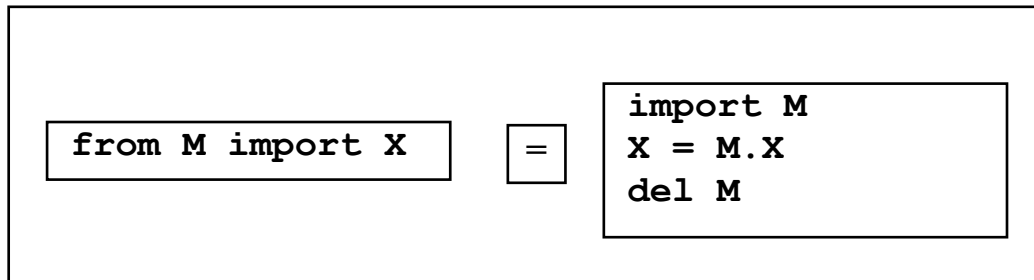
- ◆ Simple variables
- ◆ ***'X' searches for name 'X' in current scopes***
- ◆ Qualification
- ◆ ***'X.Y' searches for attribute 'Y' in object 'X'***
- ◆ Paths
- ◆ ***'X.Y.Z' gives a path of objects to be searched***
- ◆ Generality
- ◆ ***Qualification works on all objects with attributes: modules, classes, C types, etc.***



## Import variants

- ◆ **Module import model**
  - ◆ *module loaded and run on first import or from*
  - ◆ *running a module's code creates its top-level names*
  - ◆ *later import/from fetches already-loaded module*
- ◆ **import and from are assignments**
  - ◆ *import assigns an entire module object to a name*
  - ◆ *from assigns selected module attributes to names*

| <i>Operation</i>                  | <i>Interpretation</i>                   |
|-----------------------------------|-----------------------------------------|
| <code>import mod</code>           | fetch a module as a whole               |
| <code>from mod import name</code> | fetch a specific name from a module     |
| <code>from mod import *</code>    | fetch all top-level names from a module |
| <code>reload(mod)</code>          | force a reload of module's code         |



### Relative import syntax (2.5+)

```

from __future__ import absolute_import # till 2.7?, only from
import string # always finds the standard library's version

from .string import name1, name2 # import names from pkg.string
from . import string # import pkg.string

```



## Reloading modules

- ◆ Imports only load/run module code first time
- ◆ Later imports use already-loaded module
- ◆ *reload* function forces module code reload/rerun
- ◆ Allows programs to be changed without stopping

### General form

```
import module # initial import
[use module.attributes]
... # change module file
...
reload(module) # get updated exports
[use module.attributes]
```

### Usage details

A *function*, not a statement

Requires a module *object*, not a name

- ◆ Changes a module object *in-place*:
- ◆ ***runs module file's new code in current namespace***
- ◆ ***assignments replace top-level names with new values***
- ◆ ***impacts all clients that use 'import' to fetch module***
- ◆ ***impacts future 'from' clients***

```
>>> from M import func
>>> func() # edit func's source
>>> reload(M) → FAILS: no M in this scope

>>> import M
>>> reload(M)
>>> func() → FAILS: old object!

>>> M.func() # okay: refetch from M now

>>> from M import func
>>> func() # okay: refetched name
```

## Reload example

- ◆ Changes and reload file without stopping Python
- ◆ Other common uses: GUI callbacks, embedded code, etc.

```
% cat changer.py
message = "First version"

def printer():
 print message
```

```
% python
>>> import changer
>>> changer.printer()
First version
>>>
```

*[modify changer.py without stopping python]*

```
% vi changer.py
% cat changer.py
message = "After editing"

def printer():
 print 'reloaded:', message
```

*[back to the python interpreter/program]*

```
>>> import changer
>>> changer.printer() # no effect: uses loaded module
First version

>>> reload(changer) # forces new code to load/run
<module 'changer'>
>>> changer.printer()
reloaded: After editing
```

## Package imports

Module package (directory) imports

- ◆ ***module name == "dir.dir.dir" in import statements and reloads***
- ◆ ***"import dir1.dir2.module" => load dir1\dir2\mod.py***

- ◆ *“from dir1.dir2.mod import name”*
- ◆ *dir1 must be contained by a directory on PYTHONPATH*
- ◆ *each dir must have a “\_\_init\_\_.py” file, possibly empty*
- ◆ *\_\_init\_\_.py gives directory’s namespace, \_\_all\_\_ for “from\*”*
- ◆ *simplifies path, disambiguates same-named modules files*

## Example

For:

```
dir0\dir1\dir2\mod.py
```

And:

```
import dir1.dir2.mod
```

- dir0 (container) must be listed on the module search path
- dir1 and dir2 both must contain an \_\_init\_\_.py file
- dir0 does not require an \_\_init\_\_.py

```
dir0\
 dir1\
 __init__.py
 dir2\
 __init__.py
 mod.py
```

## Another example tree

```
root\
 system1\
 __init__.py
 utilities.py
 main.py
 other.py
 system2\
 __init__.py
 utilities.py
 main.py
 other.py
 system3\
 __init__.py
 myfile.py
```

(here or elsewhere)  
(your new code here)

## Odds and ends

- ◆ Python 2.0: “import module as name”

- ◆ *Like “import module” + “name = module”*
- ◆ Loading modules by name string
  - ◆ *exec ‘import ‘ + name*
  - ◆ *\_\_import\_\_(name)*
- ◆ Modules are compiled to byte code on first import
  - ◆ *‘.pyc’ files serve as recompile dependency*
  - ◆ *compilation is automatic and hidden*
- ◆ Data hiding is a convention
  - ◆ *Exports all names defined at the top-level of a module*
  - ◆ *Special case: \_\_all\_\_ list gives names exported by “from \*”*
  - ◆ *Special case: “\_X” names aren’t imported by a “from\*”*
- ◆ `__name__` and `‘__main__’`
  - ◆ *name set to ‘\_\_main\_\_’ when run as a script*
  - ◆ *allows modules to be imported and/or run*

### **file: runme.py**

```
def tester():
 print "It's Christmas in Heaven"

if __name__ == '__main__': # only when run
 tester() # not when imported
```

### **Usage modes**

```
% python
>>> import runme
>>> runme.tester()
It's Christmas in Heaven

% python runme.py
It's Christmas in Heaven
```

## Module design concepts

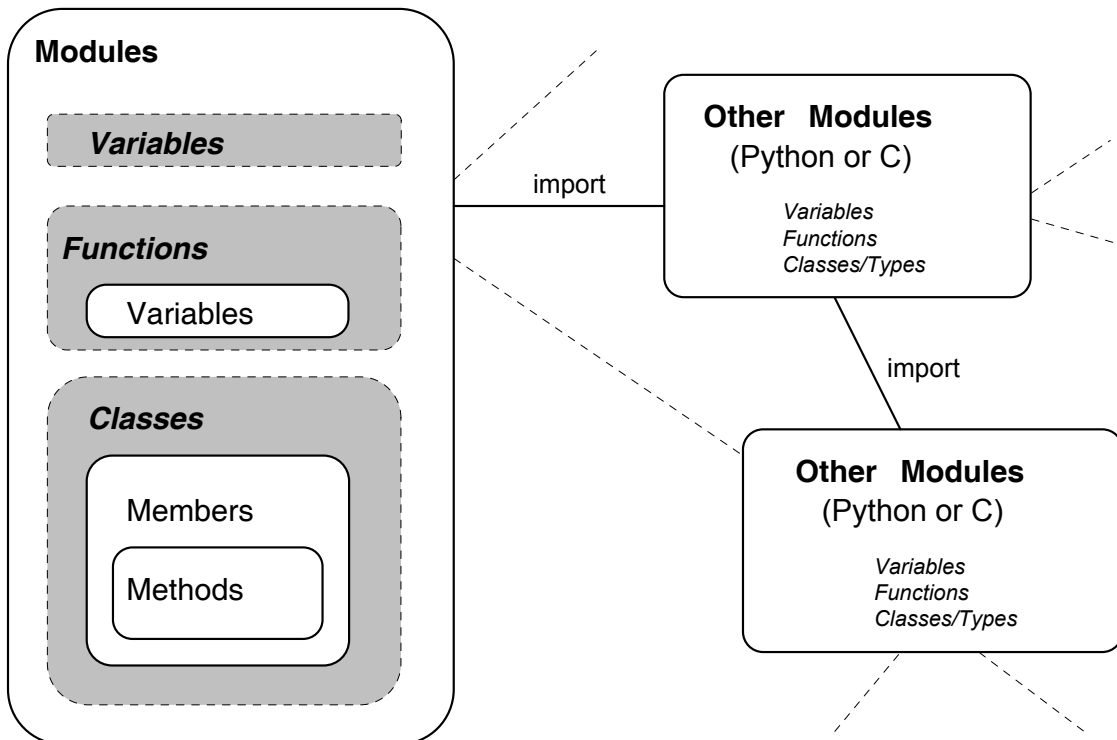
- ◆ Always in a module: `interactive = module __main__`
- ◆ Minimize module *coupling*: global variables
- ◆ Maximize module *cohesion*: unified purpose
- ◆ Modules should rarely change other module's variables

```
mod.py
X = 99 # reader sees only this

import mod
print mod.X ** 2 # always okay to use

import mod
mod.X = 88 # almost always a Bad Idea!

import mod
result = mod.func(88) # better: isolates coupling
```



## ***Modules are objects: metaprograms***

- ◆ A module which lists namespaces of other modules
- ◆ Special attributes: `module.__name__`, `__file__`, `__dict__`
- ◆ `getattr(object, name)` fetches attributes by string name
- ◆ Add to `$PYTHONSTARTUP` to preload automatically

### **File mydir.py**

```

verbose = 1

def listing(module):
 if verbose:
 print "-"*30
 print ("name: %s file: %s" %
 (module.__name__, module.__file__))
 print "-"*30

 count = 0
 for attr in module.__dict__.keys(): # scan names

```

```

 print "%02d) %s" % (count, attr),
 if attr[0:2] == "__":
 print "<built-in name>" # skip specials
 else:
 print getattr(module, attr) #__dict__[attr]
 count = count+1

if verbose:
 print "-"*30
 print module.__name__, "has %d names" % count
 print "-"*30

if __name__ == "__main__":
 import mydir
 listing(mydir) # self-test code: list myself

```

### ◆ Running the module on itself

```

C:\python> python mydir.py

name: mydir file: mydir.py

00) __file__ <built-in name>
01) __name__ <built-in name>
02) listing <function listing at 885450>
03) __doc__ <built-in name>
04) __builtins__ <built-in name>
05) verbose 1

mydir has 6 names

```

## Module gotchas

“from” copies names but doesn’t link

### nested1.py

```

X = 99
def printer(): print X

```

### nested2.py

```

from nested1 import X, printer # copy names out
X = 88 # changes my "X" only!
printer()

```

**nested3.py**

```
import nested1 # get module as a whole
nested1.X = 88 # change nested1's X
nested1.printer()
```

```
% python nested2.py
99
% python nested3.py
88
```

**Statement order matters at top-level****file: order1.py**

```
func1() # error: "func1" not yet assigned

def func1():
 print func2() # okay: "func2" looked up later

func1() # error: "func2" not yet assigned

def func2():
 return "Hello"

func1() # okay: "func1" and "func2" assigned
```

**Recursive “from” import gotchas**

- ◆ Solution: use “import”, or “from” inside functions

**file: recur1.py**

```
X = 1
import recur2 # run recur2 now if doesn't exist
Y = 2
```

**file recur2.py**

```
from recur1 import X # okay: "X" already assigned
from recur1 import Y # error: "Y" not yet assigned
```

```
>>> import recur1
Traceback (innermost last):
 File "<stdin>", line 1, in ?
 File "recur1.py", line 2, in ?
 import recur2
 File "recur2.py", line 2, in ?
 from recur1 import Y # error: "Y" not yet assigned
ImportError: cannot import name Y
```



### “reload” may not impact “from” imports

- ◆ “reload” overwrites existing module object
- ◆ But “from” names have no link back to module
- ◆ Use “import” to make reloads more effective

```
import module ⇒ module.X reflects module reloads
from module import X ⇒ X may not reflect module reloads!
```

### “reload” isn’t applied transitively

- ◆ Use multiple “reloads” to update subcomponents
- ◆ Or use recursion to traverse import dependencies

See CD’s Extras\Misc\reloadall.py

## Lab Session 5

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 7. Classes

### Why use classes?

- ♦ Implementing new objects
- ♦ *Multiple instances*
- ♦ *Customization via inheritance*
- ♦ *Operator overloading*

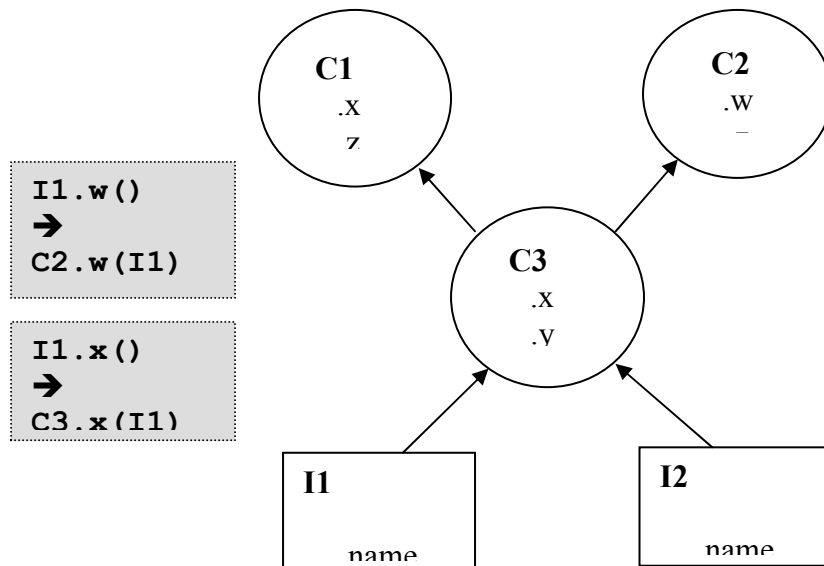
### Class topics

- ♦ Class basics
- ♦ The class statement
- ♦ Class methods
- ♦ Attribute inheritance
- ♦ Operator overloading
- ♦ Name spaces and scopes
- ♦ OOP, inheritance, and composition
- ♦ Classes and methods are objects
- ♦ Odds and ends
- ♦ Class gotchas

## OOP: The Big Picture

### How it works

- All about: “object.attr”
- Kicks off a search for first “attr” => “inheritance”
- Searches trees of linked namespace objects
- Class objects: supers and subs
- Instance objects: generated from a class
- Classes can also define expression behavior



```

class C1: ... # make class objects (ovals)
class C2: ...
class C3(C1, C2): ... # links to superclasses, search order

I1 = C3() # make instance objects (rectangles)
I2 = C3() # linked to their class
I1.x # finds customized version in C3

```

## Why OOP?

- Inheritance: basis of specializing, customizing software -- lower in tree means customization
- Program by customizing in new levels of hierarchy, not changing
- OOP great at code reuse, encapsulation, structure
- Example: an employee database app

## A first look: class basics

### 1. Classes generate multiple instance objects

- ◆ *Classes implement new objects: state + behavior*
- ◆ *Calling a class like a function makes a new instance*
- ◆ *Each instance inherits class attributes, and gets its own*
- ◆ *Assignments in class statements make class attributes*
- ◆ *Assignments to "self.attr" make per-instance attributes*

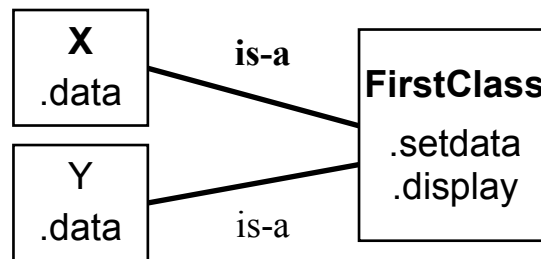
```
>>> class FirstClass: # define a class object
... def setdata(self, value): # define class methods
... self.data = value # self is the instance
... def display(self):
... print self.data # self.data: per instance

>>> x = FirstClass() # make two instances
>>> y = FirstClass() # each is a new namespace

>>> x.setdata("King Arthur") # call methods: self=x/y
>>> y.setdata(3.14159)

>>> x.display() # self.data differs in each
King Arthur
>>> y.display()
3.14159

>>> x.data = "New value" # can get/set attributes
>>> x.display() # outside the class too
New value
```



**3 objects, and  
3 namespaces**

## 2. Classes are specialized by inheritance

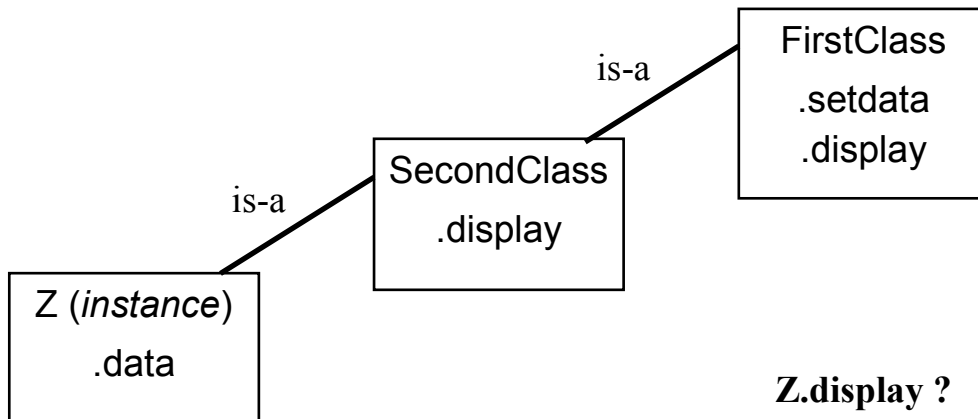
- ◆ *Superclasses listed in parenthesis in class's header*
- ◆ *Classes inherit attributes from their superclasses*
- ◆ *Instances inherit attributes from all accessible classes*
- ◆ *Logic changes made in subclasses, not in-place*

```
>>> class SecondClass(FirstClass): # inherits setdata
... def display(self): # changes display
... print 'Current value = "%s"' % self.data

>>> z = SecondClass()
>>> z.setdata(42) # setdata found in FirstClass

>>> z.display() # finds/calls overridden method
Current value = "42"

>>> x.display() # x is a FirstClass instance
New value
```



### 3. Classes can intercept Python operators

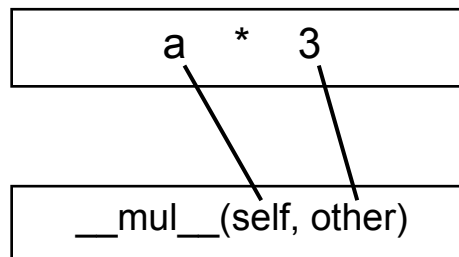
- ◆ *Methods with names like "\_\_X\_\_" are special hooks*
- ◆ *Called automatically when Python evaluates operators*
- ◆ *Classes may override most built-in type operations*
- ◆ *Allows classes to integrate with Python's object model*

```
>>> class ThirdClass(SecondClass): # isa SecondClass
... def __init__(self, value): # "ThirdClass(x)"
... self.data = value
... def __add__(self, other): # "self + other"
... return ThirdClass(self.data + other)
... def __mul__(self, other):
... self.data = self.data * other # "self * other"
```

```
>>> a = ThirdClass("abc") # new __init__ called
>>> a.display() # inherited method
Current value = "abc"
```

```
>>> b = a + 'xyz' # new __add__ called
>>> b.display()
Current value = "abcxyz"
```

```
>>> a * 3 # new __mul__ called
>>> a.display()
Current value = "abcabcabc"
```



**A More Realistic Example:**

using classes as database records  
see CD's Extras\people

## ***A closer look: class terminology***

➔ Dynamic typing and polymorphism are keys to Python

➔ “self” and “\_\_init\_\_” are key concepts in Python OOP

◆ **Class**

◆ *An object (and statement) which defines inherited members and methods*

◆ **Instance**

◆ *Objects created from a class, which inherit its attributes; each instance is a new namespace*

◆ **Member**

◆ *An attribute of a class or instance object, that's bound to an object*

◆ **Method**

◆ *An attribute of a class object, that's bound to a function object (a callable member)*

◆ **Self**

◆ *By convention, the name given to the implied instance object in methods*

◆ **Inheritance**

◆ *When an instance or class accesses a class's attributes*

◆ **Superclass**

◆ *Class or classes another class inherits attributes from*

◆ **Subclass**

◆ *Class which inherits attribute names from another class*



## Using the class statement

- ◆ Python's main OOP tool (like C++)
- ◆ Superclasses are listed in parenthesis
- ◆ Special protocols, operator overloading: `__X__`
- ◆ Multiple inheritance: `"class X(A, B, C)"` (df, l-to-r)

## General form

```
class <name>(superclass,...): # assign to name
 data = value # shared class data
 def method(self,...): # methods
 self.member = value # per-instance data
```

## Example

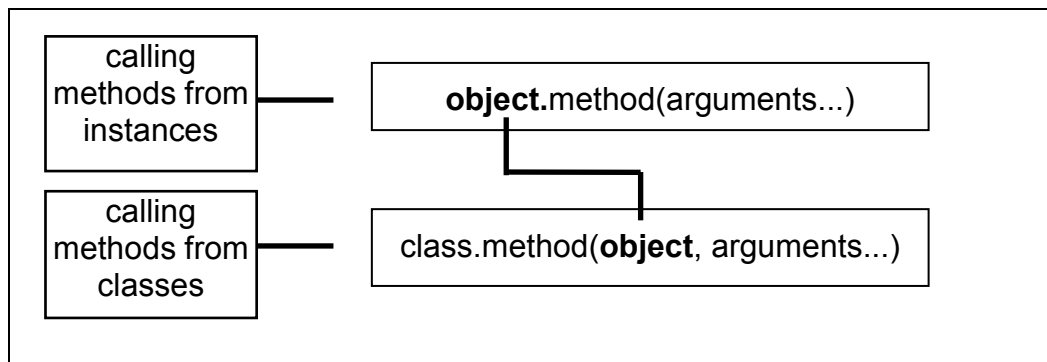
- ◆ *"class" introduces a new local scope*
- ◆ *Assignments in "class" create class object attributes*
- ◆ *"self.name = X" creates/changes instance attribute*

```
class Subclass(Superclass): # define subclass
 data = 'spam' # assign class attr
 def __init__(self, value): # assign class attr
 self.data = value # assign instance attr
 def display(self):
 print self.data, Subclass.data # instance, class

>>> x, y = Subclass(1), Subclass(2)
>>> x.display(); y.display()
1 spam
2 spam
```

## Using class methods

- ◆ “*class*” statement creates and assigns a *class* object
- ◆ Calling a class object generates an *instance* object
- ◆ Class *methods* provide behavior for instance objects
- ◆ *Methods* are nested “*def*” functions, with a ‘*self*’
- ◆ ‘*self*’ is passed the implied *instance* object
- ◆ Methods are all ‘*public*’ and ‘*virtual*’ in C++ terms



## Example

```

class NextClass: # define class
 def printer(self, text): # define method
 print text

>>> x = NextClass() # make instance
>>> x.printer('Hello world!') # call its method
Hello world!

>>> NextClass.printer(x, 'Hello world!') # class method
Hello world!

```

## Commonly used for calling superclass constructors

```
class Super:
 def __init__(self, x):
 ...default code...

class Sub(Super):
 def __init__(self, x, y):
 Super.__init__(self, x) # run superclass init
 ...custom code... # do my init actions

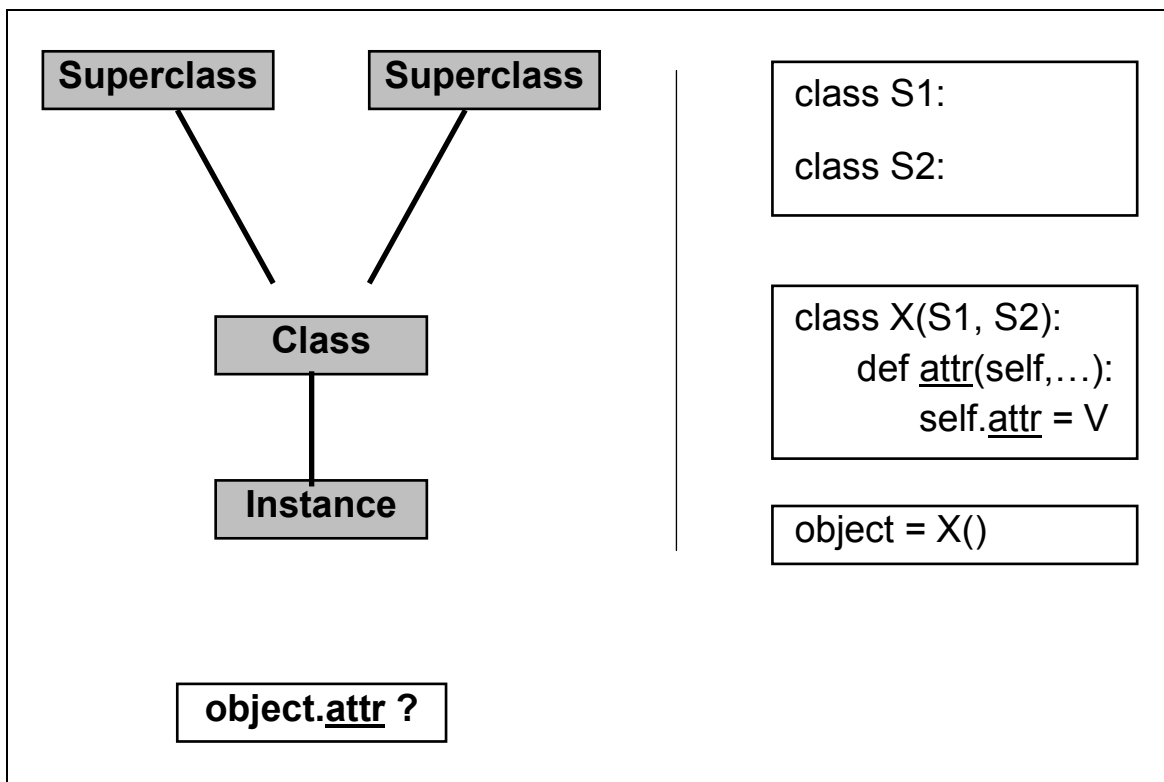
I = Sub(1, 2)
```

## Customization via inheritance

- ◆ Inheritance uses attribute definition *tree* (namespaces)
- ◆ “object.attr” searches up namespace tree for first “attr”
- ◆ Lower definitions in the tree override higher ones

### Attribute tree construction:

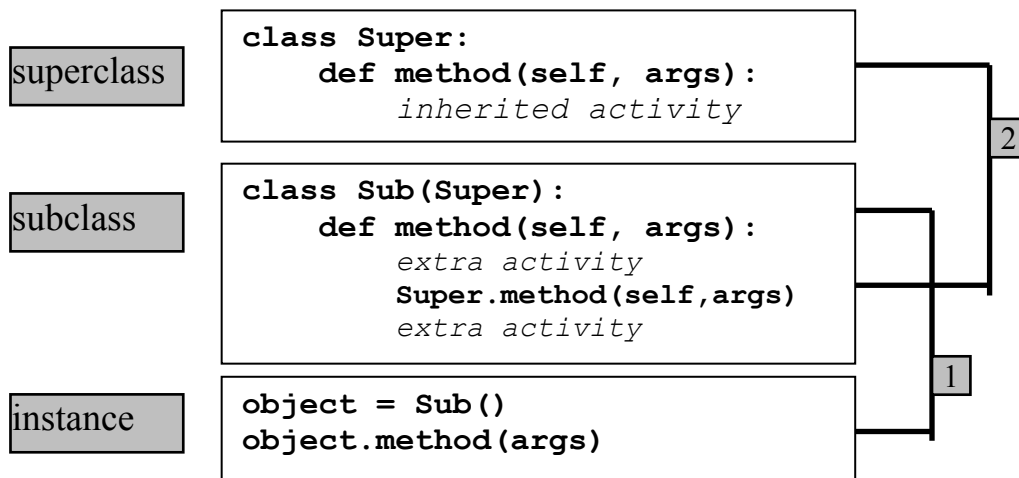
1. Instance ⇒ assignments to ‘self’ attributes
2. Class ⇒ statements (assignments) in class statements
3. Superclasses ⇒ classes listed in parenthesis in header



## Specializing inherited methods

- ◆ Inheritance finds names in subclass before superclass
- ◆ Subclasses may *inherit*, *replace*, *extend*, or *provide*
- ◆ Direct superclass method calls: `Class.method(self,...)`

```
>>> class Super:
... def method(self):
... print 'in Super.method'
...
>>> class Sub(Super):
... def method(self):
... print 'starting Sub.method'
... Super.method(self)
... print 'ending Sub.method'
...
>>> x = Super()
>>> x.method()
in Super.method
>>> x = Sub()
>>> x.method()
starting Sub.method
in Super.method
ending Sub.method
```



**file: specialize.py**

```

class Super:
 def method(self):
 print 'in Super.method' # default
 def delegate(self):
 self.action() # expected

class Inheritor(Super):
 pass

class Replacer(Super):
 def method(self):
 print 'in Replacer.method'

class Extender(Super):
 def method(self):
 print 'starting Extender.method'
 Super.method(self)
 print 'ending Extender.method'

class Provider(Super):
 def action(self):
 print 'in Provider.action'

if __name__ == '__main__':
 for klass in (Inheritor, Replacer, Extender):
 print '\n' + klass.__name__ + '...'
 klass().method()
 print '\nProvider...'
 Provider().delegate()

```

**% python specialize.py**

```

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action

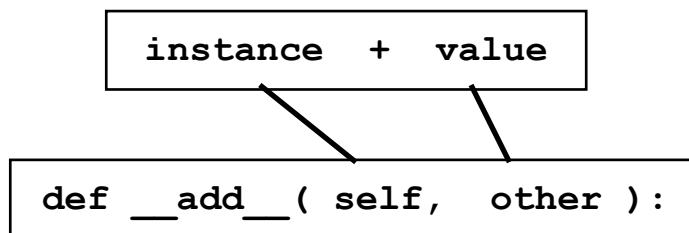
```

## Operator overloading in classes

- ◆ Lets classes intercept normal Python operations
- ◆ Can overload all Python expression operators
- ◆ Can overload object operations: print, call, qualify,...
- ◆ Makes class instances more like built-in types
- ◆ Via providing specially-names class methods

```
class Number:
 def __init__(self, start): # on Number()
 self.data = start
 def __add__(self, other): # on x + other
 return Number(self.data + other)
```

```
>>> X = Number(4)
>>> Y = X + 2
>>> Y.data
6
```



### Common operator overloading methods

- ◆ Special method names have 2 “\_” before and after
- ◆ See Python manuals or reference books for the full set

| <i>Method</i>            | <i>Overloads</i> | <i>Called for</i>                                |
|--------------------------|------------------|--------------------------------------------------|
| <code>__init__</code>    | constructor      | object creation: <code>X()</code>                |
| <code>__del__</code>     | destructor       | object reclamation                               |
| <code>__add__</code>     | operator '+'     | <code>X + Y</code>                               |
| <code>__or__</code>      | operator ' '     | <code>X   Y</code>                               |
| <code>__repr__</code>    | printing         | <code>print X</code> , <code>'X'</code>          |
| <code>__call__</code>    | function calls   | <code>X()</code>                                 |
| <code>__getattr__</code> | qualification    | <code>X.undefined</code>                         |
| <code>__getitem__</code> | indexing         | <code>X[key]</code> , iteration, <code>in</code> |
| <code>__setitem__</code> | qualification    | <code>X[key] = value</code>                      |
| <code>__len__</code>     | length           | <code>len(X)</code> , truth tests                |
| <code>__cmp__</code>     | comparison       | <code>X == Y</code> , <code>X &lt; Y</code>      |
| <code>__radd__</code>    | operator '+'     | <code>non-instance + X</code>                    |



## Examples

### ♦ getitem intercepts all index references

```
>>> class indexer:
... def __getitem__(self, index):
... return index ** 2
...
>>> X = indexer()
>>> for i in range(5):
... print X[i], # __getitem__
...
0 1 4 9 16
```

### ♦ getattr catches undefined attribute references

```
>>> class empty:
... def __getattr__(self, attrname):
... return attrname + ' not supported!'
...
>>> X = empty()
>>> X.age # __getattr__
'age not supported!'
```

- ♦ init     called on instance creation
- ♦ add     intercepts '+' expressions
- ♦ repr    returns a string when called by 'print'

```
>>> class adder:
... def __init__(self, value=0):
... self.data = value # init data
... def __add__(self, other):
... self.data = self.data + other # add other
... def __repr__(self):
... return `self.data` # to string
...
>>> X = adder(1) # __init__
>>> X + 2; X + 2 # __add__
>>> X # __repr__
5
```

## Namespace rules: the whole story

- ◆ Unqualified names (“X”) deal with lexical scopes
- ◆ Qualified names (“O.X”) use object namespaces
- ◆ Scopes initialize object namespaces: modules, classes

### The “Zen” of Python Namespaces

```

mod.py
all 5 Xs are different variables

X = 1 # global

def f():
 X = 2 # local

class C:
 X = 3 # class
 def m(self):
 X = 4 # local
 self.X = 5 # instance

```

## Unqualified names: global unless assigned

- ◆ Assignment: “X = value”
  - ◆ *Makes names local: creates or changes name in the current local scope, unless declared ‘global’*
- ◆ Reference: “X”
  - ◆ *Looks for names in the current local scope, then the current global scope, then the outer built-in scope*

## Qualified names: object name-spaces

- ◆ Assignment: “X.name = value”
  - ◆ *Creates or alters the attribute name in the namespace of the object being qualified*
- ◆ Reference: “X.name”
  - ◆ *Searches for the attribute name in the object, and then all accessible classes above it (none for modules)*

## Namespace dictionaries

- ◆ Object name-spaces: built-in “\_\_dict\_\_” attributes
- ◆ Qualification == indexing a name-space dictionary
- ◆ *To get a ‘name’ from a module “M”:*
  - M.name
  - M.\_\_dict\_\_['name']
  - sys.modules['M'].name
  - sys.modules['M'].\_\_dict\_\_['name']
  - sys.\_\_dict\_\_['modules']['M'].\_\_dict\_\_['name']

## Attribute inheritance == searching dictionaries

```
>>> class super:
... def hello(self):
... self.data = 'spam' # in self.__dict__
...
>>> class sub(super):
... def howdy(self): pass
...
>>> X = sub()
>>> X.__dict__ # a new name-space/dict
{}
>>> X.hola = 42 # add member to X object
>>> X.__dict__
{'hola': 42}

>>> sub.__dict__
{'__doc__': None, 'howdy': <function howdy at 762100>}

>>> super.__dict__
{'hello': <function hello at 769fd0>, '__doc__': None}
```

```
>>> x.hello()
>>> x.__dict__
{'data': 'spam', 'hola': 42}
```

## Optional reading: OOP and inheritance

- ◆ Inheritance based on attribute qualification
- ◆ In OOP terminology: ‘is-a’ relationship
- ◆ On “X.name”, looks for “name” in:

*Instance*       $\Leftarrow$  X’s own name-space

*Class*           $\Leftarrow$  class that X was made from

*Superclasses*  $\Leftarrow$  depth-first, left-to-right

### Example: a zoo hierarchy in Python

#### file: zoo.py

```
class Animal:
 def reply(self): self.speak()
 def speak(self): print 'spam'

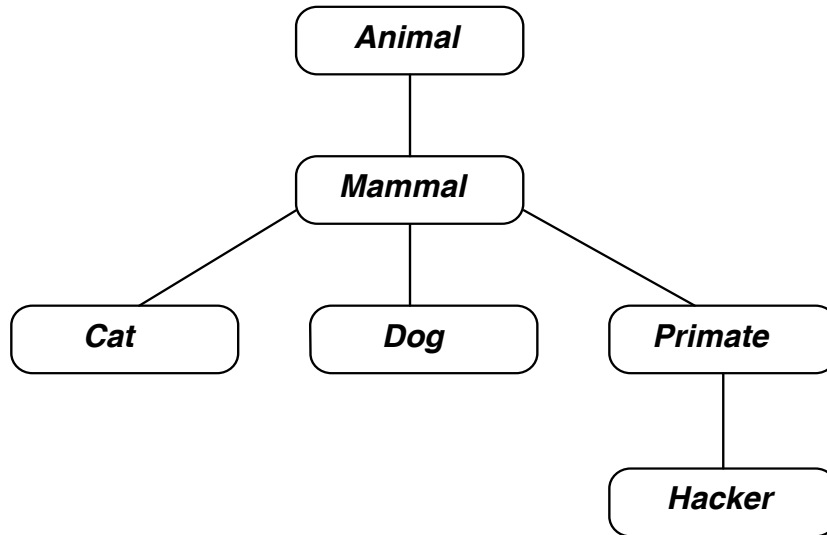
class Mammal(Animal):
 def speak(self): print 'huh?'

class Cat(Mammal):
 def speak(self): print 'meow'

class Dog(Mammal):
 def speak(self): print 'bark'

class Primate(Mammal):
 def speak(self): print 'Hello world!'

class Hacker(Primate): pass
```



```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply() # Animal.reply, Cat.speak
meow
>>> data = Hacker() # Animal.reply, Primate.speak
>>> data.reply()
Hello world!
```

## ***Optional reading: OOP and composition***

- ◆ Class instances simulate objects in a domain
- ◆ Nouns⇒classes, verbs⇒methods
- ◆ Class objects embed and activate other objects
- ◆ In OOP terminology: ‘has-a’ relationship

### **Example: the dead-parrot skit in Python**

#### **file: parrot.py**

```
class Actor:
 def line(self): print self.name + ': ', `self.says()`

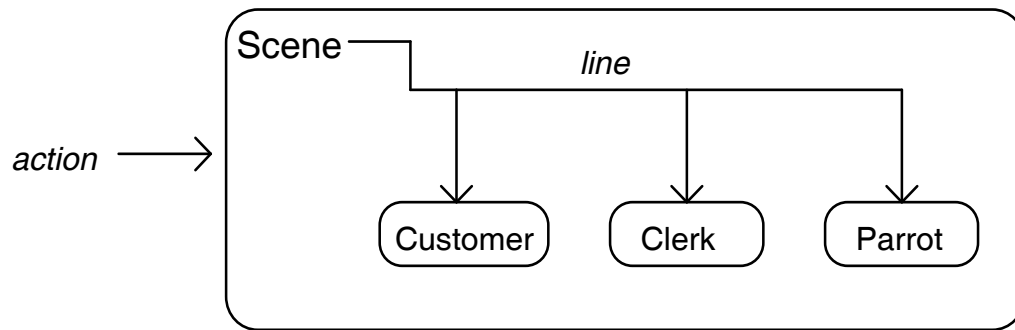
class Customer(Actor):
 name = 'customer'
 def says(self): return "that's one ex-bird!"

class Clerk(Actor):
 name = 'clerk'
 def says(self): return "no it isn't..."

class Parrot(Actor):
 name = 'parrot'
 def says(self): return None

class Scene:
 def __init__(self):
 self.clerk = Clerk() # embed some instances
 self.customer = Customer() # Scene is a composite
 self.subject = Parrot()

 def action(self):
 self.customer.line() # delegate to embedded
 self.clerk.line()
 self.subject.line()
```



```
% python
>>> import parrot
>>> parrot.Scene().action() # activate nested objects
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```

## ***Classes are objects: factories***

- ◆ Everything is a first-class ‘object’
- ◆ Only objects derived from classes are OOP ‘objects’
- ◆ Classes can be passed around as data objects

```
def factory(aClass, *args): # varargs tuple
 return apply(aClass, args) # call aClass

class Spam:
 def doit(self, message):
 print message

class Person:
 def __init__(self, name, job):
 self.name = name
 self.job = job

object1 = factory(Spam) # make a Spam
object2 = factory(Person, "Guido", "guru") # make a Person
```

## ***Methods are objects: bound or unbound***

- ◆ *Unbound* class methods: call with a ‘self’
- ◆ *Bound* instance methods: instance + method pairs

```
object1 = Spam()
x = object1.doit # bound method object
x('hello world') # instance is implied

t = Spam.doit # unbound method object
t(object1, 'howdy') # pass in instance
```



## Odds and ends

### Pseudo-private attributes

- ◆ Data hiding is a convention (until Python1.5 or later)
- ◆ “We’re all consenting adults” –Guido van Rossum
- ◆ name mangling: “self.\_\_X”  $\Rightarrow$  “self.\_Class\_\_X”
  - Class name prefix makes names unique in “self” instance
- ◆ Only works in class, and only if at most 1 trailing “\_”
- ◆ Mostly for larger, multi-programmer, OO projects

```
class C1:
 def meth1(self): self.__X = 88 # now X is mine
 def meth2(self): print self.__X # becomes _C1__X in I

class C2:
 def metha(self): self.__X = 99 # me too
 def methb(self): print self.__X # becomes _C2__X in I

class C3(C1, C2): pass
I = C3() # two X names in I

I.meth1(); I.metha()
print I.__dict__
I.meth2(); I.methb()

% python private.py
{'_C2__X': 99, '_C1__X': 88}
88
99
```

### Documentation strings

Still not universally used

Woks for classes, modules, functions, methods

- ◆ **String constant before any statements**
- ◆ **Stored in object's `__doc__` attribute**

**file: docstr.py**

```
"I am: docstr.__doc__"

class spam:
 "I am: spam.__doc__ or docstr.spam.__doc__"

 def method(self, arg):
 "I am: spam.method.__doc__ or self.method.__doc__"
 code...

def func(args):
 "I am: docstr.func.__doc__"
 code...
```

**Classes versus modules**

- ◆ Modules...
  - ◆ *Are data/logic packages*
  - ◆ *Creation: files or extensions*
  - ◆ *Usage: imported*
- ◆ Classes...
  - ◆ *Implement new objects*
  - ◆ *Always live in a module*
  - ◆ *Creation: statements*
  - ◆ *Usage: called*

**OOP and Python**

- ◆ Inheritance
  - ◆ *Based on attribute lookup: "X.name"*

## ◆ Polymorphism

- ◆ *In “X.method()”, the meaning of ‘method’ depends on the type (class) of ‘X’*

## ◆ Encapsulation

- ◆ *Methods and operators implement behavior; data hiding is a convention (for now)*

```
class C:
 def meth(self, x): # like x=1; x=2
 ...
 def meth(self, x, y, z): # the last one wins!
 ...

class C:
 def meth(self, *args):
 if len(args) == 1:
 ...
 elif type(arg[0]) == int:
 ...

class C:
 def meth(self, x): # the python way:
 x.operation() # assume x does the right thing
```

## Python's dynamic nature

Members may be added/changed outside class methods

```
>>> class C: pass
...
>>> x = C()
>>> x.name = 'bob'
>>> x.job = 'psychologist'
```

Scopes may be expanded dynamically: run-time binding

### file: delayed.py

```
def printer():
 print message # name resolved when referenced

% python
>>> import delayed
>>> delayed.message = "Hello" # set message now
>>> delayed.printer()
Hello
```

## Subclassing builtin types in 2.2 (Advanced)

- ◆ All types now behave like classes: list, str, tuple, dict,...
- ◆ Subclass to customize builtin object behavior
- ◆ Alternative to writing “wrapper” code

```
subclass builtin list type/class
map 1..N to 0..N-1, call back to built-in version

class MyList(list):
 def __getitem__(self, offset):
 print '(indexing %s at %s)' % (self, offset)
 return list.__getitem__(self, offset - 1)

if __name__ == '__main__':
 print list('abc')
 x = MyList('abc')
 print x
 print x[1]
 print x[3]
 x.append('spam'); print x
 x.reverse(); print x

__init__ inherited from list
__repr__ inherited from list
MyList.__getitem__
customizes list superclass method
attributes from list superclass

% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indexing ['a', 'b', 'c'] at 1)
a
(indexing ['a', 'b', 'c'] at 3)
c
['a', 'b', 'c', 'spam']
['spam', 'c', 'b', 'a']
```

## “New Style” Classes in 2.2 (Advanced)

- ◆ Adds new feature, changes one inheritance case
- ◆ Only if “object” or a builtin type as a superclass
- ◆ Py 3.0: all classes automatically new style, per BDFL

```
class newstyle(object):
 ...normal code...
```

Adds: slots, limits legal attributes set

```
>>> class limiter(object):
... __slots__ = ['age', 'name', 'job']
...
>>> x.ape = 1000
AttributeError: 'limiter' object has no attribute
```

Adds: properties, computed attributes alternative

```
>>> class classic:
... def __getattr__(self, name):
... if name == 'age':
... return 40
... else:
... raise AttributeError
...
>>> x = classic()
>>> x.age # runs __getattr__
40

>>> class newprops(object):
... def getage(self):
... return 40
... age = property(getage, None, None, None) # get,set,del
...
>>> x = newprops()
>>> x.age # runs getage
40
```

Adds: static and class methods, new calling patterns

```
class Spam:
 numInstances = 0 # static: no self
 # class: class, not instance
 def __init__(self):
 Spam.numInstances += 1
 def printNumInstances():
 print "Number of instances:", Spam.numInstances
 printNumInstances = staticmethod(printNumInstances)

>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Number of instances: 3
```

Function decorators (2.4+)

```
class C:
 @staticmethod
 def meth():
 ...

is equivalent to...

class C:
 def meth():
 ...
 meth = staticmethod(meth) # rebind name

@A @B @C
```

```
def f(): ...

is equivalent to...

def f(): ...
f = A(B(C(f)))
```

Changes: behavior of “diamond” multiple inheritance

```
>>> class A: attr = 1 # classic
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): pass # tries A before C
>>> x = D() # more depth-first
>>> x.attr
1

>>> class A(object): attr = 1 # new style
>>> class B(A): pass
>>> class C(A): attr = 2
>>> class D(B,C): pass # tries C before A
>>> x = D() # more breadth-first
>>> x.attr
2
```

## Class gotchas

- ◆ **Multiple inheritance: order matters**
  - ◆ *Solution: use sparingly and/or carefully*

file: mi.py

```
class Super1:
 def method2(self): # a 'mixin' superclass
 print 'in Super1.method2'

class Super2:
 def method1(self):
 self.method2() # calls my method2??
 def method2(self):
 print 'in Super2.method2'

class Sub1(Super1, Super2):
 pass # gets Super1's method2

class Sub2(Super2, Super1):
```

```

 pass # gets Super2's method2

class Sub3(Super1, Super2):
 method2 = Super2.method2 # pick method manually

Sub1().method1()
Sub2().method1()
Sub3().method1()

% python mi.py
in Super1.method2
in Super2.method2
in Super2.method2

```

## Summary: OOP in Python

- ◆ **Class objects provide default behavior**
  - ◆ *Classes support multiple copies, attribute inheritance, and operator overloading*
  - ◆ *The class statement creates a class object and assigns it to a name*
  - ◆ *Assignments inside class statements create class attributes, which export object state and behavior*
  - ◆ *Class methods are nested defs, with special first arguments to receive the instance*
- ◆ **Instance objects are generated from classes**
  - ◆ *Calling a class object like a function makes a new instance object*
  - ◆ *Each instance object inherits class attributes, and gets its own attribute namespace*
  - ◆ *Assignments to the first argument ("self") in methods create per-instance attributes*
- ◆ **Inheritance supports specialization**
  - ◆ *Inheritance happens at attribute qualification time: on "object.attribute", if object is a class or instance*
  - ◆ *Classes inherit attributes from all classes listed in their class statement header line (superclasses)*
  - ◆ *Instances inherit attributes from the class they are generated from, plus all its superclasses*

- ♦ *Inheritance searches the instance, then its class, then all accessible superclasses (depth-first, left-to-right)*

## Lab Session 6

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)



## 8. Exceptions

### Why use exceptions?

- ◆ Error handling
- ◆ Event notification
- ◆ Special-case handling
- ◆ Unusual control-flows

### Exception topics

- ◆ The basics
- ◆ Exception idioms
- ◆ Exception catching modes
- ◆ Class exceptions
- ◆ Exception gotchas

### ***Exception basics***

- ◆ A high-level control flow device
- ◆ *try* statements catch exceptions
- ◆ *raise* statements trigger exceptions
- ◆ Exceptions are raised by Python or programs

## Basic forms

- *Python 2.5+: except/finally can now be mixed*
- *Python 2.5+: with/as context managers*

### ◆ try/except/else

```
try:
 <statements> # run/call actions
except <name>:
 <statements> # if name raised during try block
except <name>, <data>:
 <statements> # if 'name' raised; get extra data
else:
 <statements> # if no exception was raised
```

### ◆ try/finally

```
try:
 <statements>
finally:
 <statements> # always run 'on the way out'
```

### ◆ raise

```
raise <name> # manually trigger an exception
raise <name>, <data> # pass extra data to catcher too
```

### ◆ assert

```
assert <test>, <message>

if not test: raise AssertionError, message
```

### ◆ with/as context managers (2.5+)

```
alternative to common try/finally idioms

from __future__ import with_statement # till 2.6

with open('/etc/passwd', 'r') as f: # auto-closed after with
 for line in f: # even if exception in block
 print line
 ... more processing code ...

lock = threading.Lock()
with lock: # auto acquired, released
 # critical section of code # classes may define managers
```

## ***First examples***

### **Builtin exceptions**

- ◆ Python triggers builtin exceptions on errors
- ◆ Displays message at top-level if not caught

```
def kaboom(list, n):
 print list[n] # trigger IndexError

try:
 kaboom([0, 1, 2], 3)
except IndexError: # catch exception here
 print 'Hello world!'
```

### **User-defined exceptions**

- ◆ Python (and C) programs raise exceptions too
- ◆ User-defined exceptions are objects

```
MyError = "my error"

def stuff(file):
 raise MyError

file = open('data', 'r') # open a file
try:
 stuff(file) # raises exception
finally:
 file.close() # always close file
```

## ***Exception idioms***

- ◆ EOFError sometimes signals end-of-file

```

while 1:
 try:
 line = raw_input() # read from stdin
 except EOFError:
 break
 else:
 <process next 'line' here>

```

- ◆ Searches sometimes signal success by 'raise'

```

Found = "Item found"

def searcher():
 <raise Found or return>

try:
 searcher()
except Found:
 <success>
else:
 <failure>

```

- ◆ Outer 'try' statements can be used to debug code

```

try:
 <run program>
except:
 # all uncaught exceptions come here
 import sys
 print 'uncaught!', sys.exc_info()[2] # type, value

```

## ***Exception catching modes***

- ◆ Try statements nest (are stacked) at runtime
- ◆ Python selects first clause that matches exception
- ◆ Try blocks can contain a variety of clauses
- ◆ Multiple excepts: catch 1-of-N exceptions
- ◆ Try can contain 'except' or 'finally', but not both

## Try block clauses

| <i>Operation</i>                    | <i>Interpretation</i>              |
|-------------------------------------|------------------------------------|
| <code>except:</code>                | catch all exception types          |
| <code>except name:</code>           | catch a specific exception only    |
| <code>except name, value:</code>    | catch exception and its extra data |
| <code>except (name1, name2):</code> | catch any of the listed exceptions |
| <code>else:</code>                  | run block if no exceptions raised  |
| <code>finally:</code>               | always perform block               |

- ◆ Exceptions nest at run-time
- ◆ *Runs most recent matching except clause*

### file: nestexc.py

```
def action2():
 print 1 + [] # generate TypeError

def action1():
 try:
 action2()
 except TypeError: # most recent matching try
 print 'inner try'

 try:
 action1()
 except TypeError: # here iff action1 re-raises
 print 'outer try'

% python nestexc.py
inner try
```

- ◆ Catching 1-of-N exceptions
- ◆ *Runs first match: top-to-bottom, left-to-right*
- ◆ *See manuals or reference text for a complete list*

```

try:
 action()
except NameError:
 ...
except IndexError:
 ...
except KeyError:
 ...
except (AttributeError, TypeError, SyntaxError):
 ...
else:
 ...

```

- ◆ **'finally' clause executed on the way out**
- ◆ *useful for 'cleanup' actions: closing files,...*
- ◆ *block executed whether exception occurs or not*
- ◆ *Python propagates exception after block finishes*
- ◆ *but exception lost if finally runs a raise, return, or break*

**file: finally.py**

```

def divide(x, y):
 return x / y # divide-by-zero error?

```

```

def tester(y):
 try:
 print divide(8, y)
 finally:
 print 'on the way out...'

```

```

print '\nTest 1: '; tester(2)
print '\nTest 2: '; tester(0) # trigger error

```

```

% python finally.py

```

```

Test 1:
4
on the way out...

```

```

Test 2:
on the way out...
Traceback (innermost last):
 File "finally.py", line 11, in ?
 print 'Test 2: '; tester(0)
 File "finally.py", line 6, in tester
 print divide(8, y)
 File "finally.py", line 2, in divide
 return x / y # divide-by-zero error?

```

`ZeroDivisionError: integer division or modulo`

- ◆ **Optional data**
- ◆ *Provides extra exception details*
- ◆ *Python passes None if no explicit data*

**file: helloexc.py**

```
myException = 'Error' # string object

def raiser1():
 raise myException, "hello" # raise, pass data

def raiser2():
 raise myException # raise, None implied

def tryer(func):
 try:
 func()
 except myException, extraInfo:
 print 'got this:', extraInfo
```

```
% python
>>> from helloexc import *
>>> tryer(raiser1)
got this: hello
>>> tryer(raiser2)
got this: None
```

## **Class exceptions**

- ◆ *Should use classes today: only option in 3.0, per BDFL*
- ◆ *Useful for catching categories of exceptions*
- ◆ *String exception match: same object ('is' identity)*
- ◆ *Class exception match: named class or subclass of it*
- ◆ *Class exceptions support exception hierarchies*

## General raise forms

```
raise string # matches same string object
raise string, data # optional extra data (default=None)
raise class, instance # matches class or its superclass
raise instance # = instance.__class__, instance
```

## Example

### file: classexc.py

```
class Super: pass
class Sub(Super): pass

def raiser1():
 X = Super() # raise listed class instance
 raise X

def raiser2():
 X = Sub() # raise instance of subclass
 raise X

for func in (raiser1, raiser2):
 try:
 func()
 except Super: # match Super or a subclass
 import sys
 print 'caught:', sys.exc_info()[0]

% python classexc.py
caught: <class Super at 770580>
caught: <class Sub at 7707f0>
```

## Example: numeric library

```
class NumErr: pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
raise DivZero()

import mathlib
try:
 mathlib.func(...)
except mathlib.NumErr:
 ...report and recover...
```



## Example: using raised instance

```
class MyBad:
 def __init__(self, file, line):
 self.file = file
 self.line = line
 def display(self):
 print self.file * 2

def parser():
 raise MyBad('spam.txt', 5)

try:
 parser()
except MyBad, X:
 print X.file, X.line
 X.display()

built-in file error numbers

def parser():
 open('nonesuch')

try:
 parser()
except IOError, X:
 print X.errno, '=>', X.strerror
```

## Exception gotchas

### What to wrap in a try statement?

- ◆ *Things that commonly fail: files, sockets, etc.*
- ◆ *Calls to large functions, not code inside the function*
- ◆ *Anything that shouldn't kill your script*
- ◆ *Simple top-level scripts often should die on errors*
- ◆ *See also `atexit` module for shutdown time actions*

### Catching too much?

- ♦ *Empty except clauses catch everything*
- ♦ *But may intercept error expected elsewhere*

```
try:
 [...]
except:
 [...] # everything comes here: even sys.exit()!
```

### Catching too little?

- ♦ *Specific except clauses only catch listed exceptions*
- ♦ *But need to be updated if add new exceptions later*
- ♦ *Class exceptions would help here: category name*

```
try:
 [...]
except (myerror1, myerror2): # what if I add a myerror3?
 [...] # non-errors
else:
 [...] # assumed to be an error
```

### Solution: exception protocol design

## Lab Session 7

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 9. Built-in Tools Overview

### ♦ Python's tool box

- ♦ Types and operations: *lists, dictionaries, files, slices,...*
- ♦ Functions: *len, range, apply, getattr...*
- ♦ Modules (Python and C): *string, os, Tkinter, pickle,...*
- ♦ Exceptions: *IndexError, KeyError,...*
- ♦ Attributes: *\_\_dict\_\_, \_\_name\_\_,...*
- ♦ Peripheral tools: *NumPy, SWIG, JPython, PythonWin,...*

### Topics

- ♦ Debugging options
- ♦ Inspecting name-spaces
- ♦ Dynamic coding tools
- ♦ Timing and profiling Python programs
- ♦ Packaging Python programs
- ♦ Development tools for larger projects
- ♦ Testing tools
- ♦ Summary: Python tool set layers

But first, the secret handshake...

## Debugging options

- ◆ Imported modules, written in Python
- ◆ May also be run as scripts, in latest release
- ◆ ‘pdb’ debugger: dbx-like command line interface
- ◆ New: ‘IDLE’ Tkinter-based debugger GUI
- ◆ See library manuals for pdb commands and usage

### Other error-handling tricks

- ◆ *Top-level stack tracebacks*
- ◆ *Inserting print statements*
- ◆ *Outer exception handlers*

```
def safe(entry, *args):
 try:
 apply(entry, args) # catch everything else
 except:
 import sys
 print sys.exc_info()[0], sys.exc_info()[1] # type, value
```

## Debugging example

### file: boom.py

```
def func(x, y):
 return x / y
```

### Session

```
>>> import boom
>>> boom.func(1, 0)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
 File "boom.py", line 3, in func
 return x / y
ZeroDivisionError: integer division or modulo
>>>
>>> import pdb
>>> pdb.run('boom.func(1, 0)') ← run/debug code
> <string>(0)?()
(Pdb) b boom.func ← set breakpoint
(Pdb) c ← continue program
> boom.py(2)func()
-> def func(x, y):
(Pdb) s ← step 1 line
> boom.py(3)func()
-> return x / y
(Pdb) s
ZeroDivisionError: 'integer division or modulo'
> boom.py(3)func()
-> return x / y
(Pdb) where ← stack trace
 <string>(1)?()
> boom.py(3)func()
-> return x / y
(Pdb) p y ← print variables
0
```

## Inspecting name-spaces

- ♦ Python lookups use 3-scope rule: local, global, built-in
- ♦ `locals()`, `globals()`: return name-spaces as dictionaries

```
% python
>>> def func(x):
... a = 1
... print locals() # on function call
... print globals().keys()
...
>>> class klass:
... def __init__(self):
... print locals() # on instance creation
... print globals().keys()
... print locals() # on class creation
... print globals().keys()
...
{'__init__': <function __init__ at 76ed00>}
['__builtins__', '__name__', 'func', '__doc__']

>>> func(1)
{'a': 2, 'x': 1}
['__builtins__', '__name__', 'func', 'klass', '__doc__']

>>> x = klass()
{'self': <klass instance at 76f8d0>, 'arg': None}
['__builtins__', '__name__', 'func', 'klass', '__doc__']

>>> def nester(L, M, N):
... class nested: # assigns class to name
... def __init__(self):
... pass
... print locals() # local=class global=mod
... print globals().keys() # no access to L/M/N!
... return nested
...
>>> nester(1, 2, 3)
{'__init__': <function __init__ at 761e30>}
['__doc__', 'nester', '__name__', 'x', 'func', 'klass', ...]
<class nested at 762960>
```

## Dynamic coding tools

- ◆ *apply* runs functions with argument tuples
- ◆ *eval* evaluates a Python expression code-string
- ◆ *exec* runs a Python statement code-string
- ◆ *getattr* fetches an object's attribute by name string
- ◆ Supports run-time program construction
- ◆ Supports embedding Python in Python

### Basic usage

```
>>> x = "2 ** 5"
>>> a = eval(x)
>>> a
32

>>> exec "print a / 2"
16

>>> def echo(a, b, c): print a, b, c
...
>>> apply(echo, (1, 2, 3))
1 2 3

>>> import string
>>> getattr(string, "uppercase")
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

>>> D = {}
>>> exec "import math\nx = math.pi" in D, D
>>> D['x']
3.14159265359
```

## Example

- ◆ Import module by name, run function by name/args
- ◆ Runs: [ message.printer('sir', 'robin') ]
- ◆ Also see: new `__import__` function
- ◆ Preview: will revisit model to embed Python in C

### file: message.py

```
def printer(str1, str2):
 return 'brave', str1, str2
```

### file: dynamic.py

```
def runFunction(moduleName, functionName, argsTuple):
 exec 'import ' + moduleName
 module = eval(moduleName)
 function = getattr(module, functionName)
 return apply(function, argsTuple)

if __name__ == '__main__':
 from sys import argv
 print runFunction(argv[1], argv[2], tuple(argv[3:]))
```

### Command line

```
% python dynamic.py message printer sir robin
('brave', 'sir', 'robin')
```



## Timing and profiling Python programs

- ◆ *time* module contains C library type tools
- ◆ Useful for performance tweaking (along with profiler)
- ◆ Warning: be careful to compare apples to apples!

### Example: inline stack alternatives

- ◆ List based stacks

```
['spam0', 'spam1', 'spam2'] ← top
```

- ◆ *Use list in-place changes: append/del*
- ◆ *Lists resized on demand: grown in increments*

- ◆ Tuple-pair based stacks

```
top ⇒ ('spam2', ('spam1', ('spam0', None)))
```

- ◆ *Use tuple packing/unpacking assignments*
- ◆ *Build a tree of 2-item tuples: (item, tree)*
- ◆ *Like Lisp 'cons' cells/linked lists: avoids copies*

**file: testinline.py**

```
#!/opt/local/bin/python
import time
from sys import argv, exit
numtests = 20

try:
 pushes, pops = eval(argv[1]), eval(argv[2])
except:
 print 'usage: testinline.py <pushes> <pops>'; exit(1)

def test(reps, func):
 start_cpu = time.clock()
 for i in xrange(reps):
 x = func()
 return time.clock() - start_cpu

def inline1():
 # builtin lists
 x = []
 for i in range(pushes): x.append('spam' + `i`)
 for i in range(pops): del x[-1]

def inline2():
 # builtin tuples
 x = None
 for i in range(pushes): x = ('spam' + `i`, x)
 for i in range(pops): (top, x) = x

print 'lists: ', test(numtests, inline1) # run 20 times
print 'tuples:', test(numtests, inline2)
```

**Results**

```
% testinline.py 500 500 --20*(500 pushes + 500 pops)
lists: 0.77 --lists: append/del
tuples: 0.43 --tuples: pack/unpack
% testinline.py 1000 1000 --20K pushes + 20K pops
lists: 1.54
tuples: 0.87
% testinline.py 200 200
lists: 0.31
tuples: 0.17
% testinline.py 5000 5000
lists: 7.72
tuples: 4.5
```

**Related Modules****datetime**

```
>>> from datetime import datetime, timedelta
>>> x = datetime(2004, 11, 21)
>>> y = datetime(2005, 3, 19)
```

```
>>>
>>> y - x
datetime.timedelta(118)
>>>
>>> x + timedelta(30)
datetime.datetime(2004, 12, 21, 0, 0)
```

## profile

- **cProfile: more efficient version in 2.5+**
- **Run as script or interactive**
- **pstats to report on results later**
- **Optimize: profile, time, shedskin/psyco, move to C**

```
>>> import profile
>>> profile.run('import test1')
35 function calls in 0.027 CPU seconds
```

Ordered by: standard name

| ncalls | totttime | percall | cumtime | percall | filename:lineno(function) |
|--------|----------|---------|---------|---------|---------------------------|
| 16     | 0.001    | 0.000   | 0.001   | 0.000   | :0(range)                 |
| 1      | 0.005    | 0.005   | 0.005   | 0.005   | :0(setprofile)            |
| 1      | 0.002    | 0.002   | 0.022   | 0.022   | <string>:1(?)             |
| 1      | 0.000    | 0.000   | 0.027   | 0.027   | profile:0(import test1)   |
| 0      | 0.000    |         | 0.000   |         | profile:0(profiler)       |
| 1      | 0.000    | 0.000   | 0.020   | 0.020   | test1.py:2(?)             |
| 15     | 0.019    | 0.001   | 0.020   | 0.001   | test1.py:2(myfun)         |

### Example: timing iteration alternatives

## Packaging programs for delivery

### Byte code

*Modules compiled to portable byte-code on import: .pyc*

*compileall module forces imports, to make .pyc's*

*.pyo files created and run with -O command-line flag*

### Frozen Binaries

*Package byte-code + Python in an executable*

*Don't require Python to be installed*

*Protect your program code*

## Other options

*Import hooks support zip files, decryption, etc.*

*Pickler converts objects to/from text stream (serializer)*

| <i>Format</i>                    | <i>Medium</i>                 |
|----------------------------------|-------------------------------|
| Source files                     | .py files, scripts            |
| Source files, no console         | .pyw files (Windows, GUI)     |
| Compiled byte-code files         | .pyc files, .pyo files(1.5+)  |
| Encrypted byte-code files        | Import hooks, PyCrypto        |
| Frozen binaries, self-installers | Py2Exe, PyInstaller, Freeze   |
| distutils                        | Setup.py installation scripts |
| Zip files of modules             | Auto in 2.4+ (zipimport 2.3)  |
| Pickled objects                  | raw objects                   |
| Embedding mediums                | databases, etc.               |
| Jython: Java bytecode            | network downloads, etc.       |

## ***Development tools for larger projects***

|           |                                              |
|-----------|----------------------------------------------|
| PyDoc     | Displaying docstrings, program structure     |
| PyChecker | Pre-run error checking (a “lint” for Python) |
| PyUnit    | Unit testing framework (a.k.a. unittest)     |
| Doctest   | docstring-based regression test system       |
| IDEs      | IDLE, Komodo, PythonWin, PythonWorks         |

|                |                                                                                                         |
|----------------|---------------------------------------------------------------------------------------------------------|
| Profilers      | profile, hotshot                                                                                        |
| Debuggers      | pdb, IDLE point-and-click, print                                                                        |
| Optimization   | Psyco, .pyo bytecode, C extensions, SWIG                                                                |
| Packaging      | Py2Exe, Installer, Freeze (above)                                                                       |
| Distutils      | packaging, install, build script system                                                                 |
| Language tools | module packages, private attributes, class exceptions, <code>__name__ == '__main__'</code> , docstrings |

## ***Testing tools in the standard library***

### **Simplest convention**

```
if __name__ == '__main__':
 unit test code here...
```

### **doctest module**

- automates interactive session
- regression test system

```
file spams.py

"""
This module works as follows:

>>> spams(3)
'spamspamspam'
>>> shrubbery()
'spamspamspamspam!!!'
"""

def spams(N):
 return 'spam' * N

def shrubbery():
 return spams(4) + '!!!!'

if __name__ == '__main__':
 import doctest
 doctest.testmod()
```

```

C:\Python25>python spams.py -v
Trying:
 spams(3)
Expecting:
 'spamspamspam'
ok
Trying:
 shrubbery()
Expecting:
 'spamspamspamspam!!!'
ok
2 items had no tests:
 __main__.shrubbery
 __main__.spams
1 items passed all tests:
 2 tests in __main__
2 tests in 3 items.
2 passed and 0 failed.
Test passed.

```

## unittest module (PyUnit)

- class structure for unit test code
- See library manual: test suites, ...

```

import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

 def setUp(self):
 self.seq = range(10)

 def testshuffle(self):
 # make sure the shuffled sequence does not lose any elements
 random.shuffle(self.seq)
 self.seq.sort()
 self.assertEqual(self.seq, range(10))

 def testchoice(self):
 element = random.choice(self.seq)
 self.assertIn(element, self.seq)

 def testsample(self):
 self.assertRaises(ValueError, random.sample, self.seq, 20)
 for element in random.sample(self.seq, 5):
 self.assertIn(element, self.seq)

if __name__ == '__main__':
 unittest.main()

```

## Summary: Python tool-set layers

### ♦ Built-ins

- ♦ *Lists, dictionaries, strings, library modules, etc.*
- ♦ *High-level tools for simple, fast programming*

### ♦ Python extensions

- ♦ *Functions, classes, modules*
- ♦ *For adding extra features, and new object types*

### ♦ C extensions

- ♦ *C modules, C types*
- ♦ *For integrating external systems, optimizing components, customization*

## Lab Session 7

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 10. System interfaces

### System modules overview

#### Python tools: sys

- ♦ *Python-related exports*
- ♦ *path: initialized from PYTHONPATH, changeable*
- ♦ *platform: 'sunos', 'win32', 'linux2', etc.*
- ♦ *sys.exit(N), sys.exc\_info(), ...*

```
>>> import sys
>>> sys.path
['.', '/usr/local/lib/python', ...]
>>> sys.platform
'sunos4'
```

#### System tools: os

- ♦ *POSIX bindings: operating system exports*
- ♦ *~200 attributes on some platforms*

#### *Content survey*

- ♦ *Shell environment variables*  
     `os.environ`
- ♦ *Running shell commands, programs:*  
     `os.system, os.popen, os.popen2/3/4, os.startfile`
- ♦ *Spawning processes:*  
     `os.fork, os.pipe, os.exec, os.waitpid, os.kill`
- ♦ *Descriptor files, with locking:*



```
os.open, os.read, os.write
```

♦ **File processing:**

```
os.remove, os.rename, os.mkfifo, os.mkdir, os.rmdir
```

♦ **Administrative tools:**

```
os.getcwd, os.chdir, os.chmod, os.getpid, os.listdir
```

♦ **Portability tools:**

```
os.sep, os.pathsep, os.curdir, os.path.split, os.path.join
```

♦ **os.path nested submodule: pathname tools**

```
os.path.exists('filepathname')
os.path.isdir('filepathname')
os.path.getsize('filepathname')
```

*Running shell commands*

```
>>> import os
>>> listing = os.popen("ls *.py").readlines()
>>> for name in listing: print name,
...
cheader1.py
finder1.py
summer.py

>>> for name in listing: os.system("vi " + name)
...
```

Variations:

```
os.popen('cmd', 'w').write('data')
(i, o) = os.popen2('cmd')
(i, o, e) = os.popen3('cmd')
(i, o_e) = os.popen4('cmd')
os.startfile('file')
```

- See **pty**, **pexpect** to automate interactive programs without deadlocks

**Others: glob, socket, select, thread, fcntl,...**

- ♦ See *Python library manual for the full story*

**Example: testing command-line scripts**

## Arguments, streams, shell variables

### Shell environment variables: `os`

- ◆ *`os.environ`: read/write access to shell variables*
- ◆ *normal dictionary interface*

```
>>> import os
>>> os.environ['USER']
'mlutz'
>>> os.environ['USER'] = 'Bob' # changes for process and its children
```

- See `getopt`, `optparse` modules for parsing complex commands lines

### Arguments and streams: `sys`

- ◆ *`sys.argv`: command-line arguments*
- ◆ *`sys.stdin/stdout/stderr`: standard stream files*

```
% cat play.py
#!/usr/local/bin/python
import sys
print sys.argv
sys.stdout.write("ta da!\n") # same as: print 'ta da!'

% play.py -x -i spammyfy
['play.py', '-x', '-i', 'spammyfy']
ta da!
```

## ***File tools***

- **Built-in file objects**

*For most file applications*

- **Processing binary data on Windows**

*Use “rb” and “wb” to suppress line-end translations*

- **Module os descriptor-based file tools**

*For special/advanced file processing modes*

- **Module os filename tools**

*Deletions, renamings, etc.*

- **Module os.path tools**

*File existence, directory tests, size, etc.*

- **See also**

*Sockets, pipes, fifos, shelves, DBM files*

## ***Directory tools***

### **Single directories**

1) Running directory listing commands: non-portable

```
C:\temp>python
>>> import os
>>> os.popen('dir /B').readlines()
['about-pp.html\012', 'python1.5.tar.gz\012', 'about-pp2e.html\012',
'about-ppr2e.html\012', 'newdir\012']

>>> os.popen('ls C:\PP2ndEd').readlines()
['README.txt\012', 'cdrom\012', 'chapters\012', 'etc\012', 'examples\012',
'examples.tar.gz\012', 'figures\012', 'shots\012']
```

## 2) The glob module: patterns

```
>>> import glob
>>> glob.glob('C:\PP2ndEd*')
['C:\PP2ndEd\examples.tar.gz', 'C:\PP2ndEd\README.txt',
'C:\PP2ndEd\shots', 'C:\PP2ndEd\figures', 'C:\PP2ndEd\examples',
'C:\PP2ndEd\etc', 'C:\PP2ndEd\chapters', 'C:\PP2ndEd\cdrom']
```

## 3) The os.listdir call: quick, portable

```
>>> os.listdir('C:\PP2ndEd')
['examples.tar.gz', 'README.txt', 'shots', 'figures', 'examples', 'etc',
'chapters', 'cdrom']

>>> os.listdir(".")
['summer.out', 'summer.py', 'table1.txt', ...]
```

## Directory trees

### 1) os.path.walk

```
>>> import os
>>> def lister(dummy, dirname, filesindir):
... print '[' + dirname + ']'
... for fname in filesindir:
... print os.path.join(dirname, fname) # handle one file
...
>>> os.path.walk('.', lister, None)
[.]
.\about-pp.html
.\python1.5.tar.gz
.\about-pp2e.html
.\about-ppr2e.html
.\newdir
```

```
[.\newdir]
.\newdir\temp1
.\newdir\temp2
.\newdir\temp3
.\newdir\more
[.\newdir\more]
.\newdir\more\xxx.txt
.\newdir\more\yyy.txt
```

## 2) os.walk generator (2.3+)

```
>>> import os
>>> for (thisDir, dirsHere, filesHere) in os.walk('.'):
 print thisDir, '=>'
 for filename in filesHere:
 print '\t', filename

. =>
 w9xpopen.exe
 py.ico
 pyc.ico
 README.txt
 NEWS.txt
 ...
```

## 3) The find module: deprecated in 2.0, gone today (see below)

```
C:\temp>python
>>> import find
>>> find.find('*')
['.\\about-pp.html', '.\\about-pp2e.html', '.\\about-ppr2e.html',
'.\\newdir', '.\\newdir\\more', '.\\newdir\\more\\xxx.txt',
'.\\newdir\\more\\yyy.txt', '.\\newdir\\temp1', '.\\newdir\\temp2',
'.\\newdir\\temp3', '.\\python1.5.tar.gz']
```

## 4) Recursive traversals

```
list files in dir tree by recursion
import sys, os

def mylister(currdir):
 print '[' + currdir + ']'
 for file in os.listdir(currdir):
 path = os.path.join(currdir, file)
 if not os.path.isdir(path):
 print path
 else:
 mylister(path)

if __name__ == '__main__':
 # list files here
 # add dir path back
 # recur into subdirs
```

```
mylister(sys.argv[1])
```

```
dir name in cmdline
```

### Example: finding large files

## **Renaming a set of files**

```
>>> import glob, string, os
>>> glob.glob("*.py")
['cheader1.py', 'finder1.py', 'summer.py']

>>> for name in glob.glob("*.py"):
... os.rename(name, string.upper(name))
...

>>> glob.glob("*.PY")
['FINDER1.PY', 'SUMMER.PY', 'HEADER1.PY']
```

## **Rolling your own find module (Extras dir)**

```
#!/usr/bin/python
#####
custom version of the now deprecated find module
in the standard library--import as "PyTools.find";
equivalent to the original, but uses os.path.walk,
has no support for pruning subdirs in the tree, and
is instrumented to be runnable as a top-level script;
results list sort differs slightly for some trees;
exploits tuple unpacking in function argument lists;
#####

import fnmatch, os

def find(pattern, startdir=os.curdir):
 matches = []
 os.path.walk(startdir, findvisitor, (matches, pattern))
 matches.sort()
 return matches

def findvisitor((matches, pattern), thisdir, nameshere):
 for name in nameshere:
 if fnmatch.fnmatch(name, pattern):
 fullpath = os.path.join(thisdir, name)
 matches.append(fullpath)

if __name__ == '__main__':
```

```
import sys
namepattern, startdir = sys.argv[1], sys.argv[2]
for name in find(namepattern, startdir): print name
```

## Forking processes

- ◆ Spawns (copies) a program
- ◆ Parent and child run independently
- ◆ *fork* spawns processes; *system/popen* spawn commands
- ◆ Not on Windows, today (use threads or spawnv)

```
starts programs until you type 'q'
import os

parm = 0
while 1:
 parm = parm+1
 pid = os.fork()
 if pid == 0:
 os.execlp('python', 'python', 'child.py', str(parm)) # copy process
 assert 0, 'error starting program' # overlay program
 # shouldn't return
 else:
 print 'Child is', pid
 if raw_input() == 'q': break
```

### Example: cross-linking streams

- ◆ Input ⇒ connect *stdin* to program's *stdout*: `raw_input`
- ◆ Output ⇒ connect *stdout* to program's *stdin*: `print`
- ◆ Also see: `os.popen2` call

**file: ipc.py**  
import os

```

def spawn(prog, args):
 pipe1 = os.pipe() # (parent input, child output)
 pipe2 = os.pipe() # (child input, parent output)
 pid = os.fork() # make a copy of this process
 if pid:
 # in parent process
 os.close(pipe1[1]) # close child ends here
 os.close(pipe2[0])
 os.dup2(pipe1[0], 0) # sys.stdin = pipe1[0]
 os.dup2(pipe2[1], 1) # sys.stdout = pipe2[1]
 else:
 # in child process
 os.close(pipe1[0]) # close parent ends here
 os.close(pipe2[1])
 os.dup2(pipe2[0], 0) # sys.stdin = pipe2[0]
 os.dup2(pipe1[1], 1) # sys.stdout = pipe1[1]
 cmd = (prog,) + args
 os.execv(prog, cmd) # overlay new program

```

## Python thread modules

- ◆ Runs function calls in parallel, share global (module) memory
- ◆ Portable: runs on Windows, Solaris, any with pthreads
- ◆ Global interpreter lock: one thread running code at a time
- ◆ Thread switches on bytecode counter and long-running calls
- ◆ Must still synchronize concurrent updates with thread locks
- ◆ C extensions release and acquire global lock too

```

import thread

def counter(myId, count):
 # synchronize stdout access to avoid multi prints on 1 line
 for i in range(count):
 mutex.acquire()
 print "[%s] => %s" % (myId, i)
 mutex.release()

mutex = thread.allocate_lock()
for i in range(10):
 thread.start_new(counter, (i, 100))

import time
time.sleep(10)
print 'Main thread exiting.'

```



## Output

```
.
.
.
[3] => 98
[4] => 98
[5] => 98
[7] => 98
[8] => 98
[0] => 99
[9] => 98
[6] => 99
[1] => 99
[2] => 99
[3] => 99
[4] => 99
[5] => 99
[7] => 99
[8] => 99
[9] => 99
Main thread exiting.
```

## Locking concurrent updaters

### Fails:

```
fails on windows due to concurrent updates;
works if check-interval set higher or lock
acquire/release calls made around the adds

import thread, time
count = 0

def adder():
 global count
 count = count + 1 # update shared global
 count = count + 1 # thread swapped out before returns

for i in range(100):
 thread.start_new(adder, ()) # start 100 update threads
time.sleep(5)
print count
```

### Works:

```

import thread, time, sys
mutex = thread.allocate_lock()
count = 0

def adder():
 global count
 mutex.acquire()
 count = count + 1 # update shared global
 count = count + 1 # thread swapped out before returns
 mutex.release()

for i in range(100):
 thread.start_new(adder, ()) # start 100 update threads
time.sleep(5)
print count

```

### See also:

*“threading” module’s class-based interface*

*“Queue” module’s thread-safe queue get/put*

```

import threading

class mythread(threading.Thread): # subclass Thread object
 def __init__(self, myId, count):
 self.myId = myId
 self.count = count
 threading.Thread.__init__(self)
 def run(self): # run provides thread logic
 for i in range(self.count): # still synch stdout access
 stdoutmutex.acquire()
 print '[%s] => %s' % (self.myId, i)
 stdoutmutex.release()

stdoutmutex = threading.Lock() # same as
thread.allocate_lock()
thread = mythread()
thread.start()
thread.join() # wait for exit

```

**Examples: Queue module, on CD Extras/PP3E**

## Fork versus spawnv

- ◆ For starting programs on Windows
- ◆ Spawnv like fork+exec for Unix
- ◆ See also: `os.system("start file.py")`

```
#####
do something similar by forking process instead of threads
this doesn't currently work on Windows, because it has no
os.fork call; use os.spawnv to start programs on Windows
instead; spawnv is roughly like a fork+exec combination;
#####

import os, sys

for i in range(10):
 if sys.platform[:3] == 'win':
 path = r'C:\program files\python\python.exe'
 os.spawnv(os.P_DETACH, path,
 ('python', 'thread-basics6.py'))
 else:
 pid = os.fork()
 if pid != 0:
 print 'Process %d spawned' % pid
 else:
 os.execvp('python', 'python', 'thread-basics6.py')
print 'Main process exiting.'
```

## Lab Session 8

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 11. GUI programming

### What makes Python good at GUIs?

- ◆ Rapid turnaround... easy to experiment
- ◆ Very high-level... fast to code GUIs
- ◆ Object-oriented... code reusability
- ◆ Dynamic reloading... changes without stopping

### Python GUI Options

- ◆ **Tkinter**
  - ◆ *Python's OO interface to the portable Tk API*
  - ◆ *Part of Python, lightweight, well documented, robust*
  - ◆ *Meshes well with a scripting language: from Tcl*
  - ◆ *Runs with native look-and-feel on X, MS-Windows, Macs*
  - ◆ *Structure: Tkinter + Tk lib + (X, Windows, Mac libs)*
  - ◆ *A de-facto standard: Tk used by Python, Perl, and TCL*
  - ◆ *An open source system, supported by [Scriptics](#)*
  - ◆ *Base API originally developed by John Ousterhout*



**◆ wxPython**

- ◆ *Second most popular GUI API for Python (?)*
- ◆ *Portable GUI class library written in C++ (X, Windows, Mac)*
- ◆ *Wraps a C++ API: wxWindows + wxPython*
- ◆ *Rich widget set: trees, html viewers, notebooks*
- ◆ *Tends to be more complex: C++ API (wx) verses scripting API (Tk)*
- ◆ *Was less documented, but wxPython book in 2006*
- ◆ *GUI builders: BoaConstructor, wxDesigner*
- ◆ *Tkinter + PMW package roughly as rich as wxPython*

**◆ PyQt, PyGTK**

- ◆ *From KDE & Gnome Linux libraries, but now portable*
- ◆ *PyQt third most popular GUI API for Python (?)*
- ◆ *PyQt: works on Sharp Zaurus PDAs, open source book*
- ◆ *PyQt: not completely open source (still true?)*
- ◆ *Qt GUI builders: BlackAdder, QT Designer*

**◆ PyWin32 (former known as win32all)**

- ◆ *MFC integration for Python on Windows (only)*

**◆ Jython (a.k.a. Jpython)**

- ◆ *Access to Java GUI APIs: AWT, Swing, etc.*

**◆ IronPython .NET/Mono port**

- ◆ *.NET may provide GUI tools*

**◆ PythonCard**

- ◆ *API + drag-and-drop builder on top of wxPython*

**◆ Dabo**

- ◆ *3-tier application builder with GUI, Dbase, logic*
- ◆ **Anygui**
  - ◆ *Portable API implemented on top of Tk, wx, Qt*

See: [Extras\Gui\wxPython\wxPython.doc](#) on CD for Tkinter, wxPython comparisons

## ***The Tkinter 'hello world' program***

- ◆ Widgets: class instances
- ◆ Options: keyword arguments
- ◆ Tkinter exports classes and constants
- ◆ Widgets must be packed (or gridded, placed)
- ◆ 'mainloop' shows widgets, catches events

### **file: gui1.py**

```
from Tkinter import * # get widget classes
Label(text="Hello GUI world!").pack() # make a label
mainloop() # show, catch events
```

```
% python gui1.py
```



**Example: a live GUI demo*****Adding buttons, frames, and callbacks***

- ◆ Frames are widget containers
- ◆ Widgets attach to sides of a parent
- ◆ Event handlers are any callable object
- ◆ Button handlers are registered as ‘command’ options

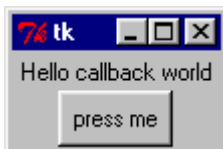
**file: gui2.py**

```
from Tkinter import * # get widgets

def callback(): # define handler
 print 'hello stdout world...'

top = Frame() # make a container
top.pack()
Label(top, text="Hello callback world").pack(side=TOP)
Button(top, text="press me",
 command=callback).pack(side=BOTTOM)
top.mainloop()
```

```
% python gui2.py
hello stdout world...
hello stdout world...
```



## Getting input from a user

- ◆ Entry is a one-line input field
- ◆ lambda defers call to add an input argument
- ◆ showinfo is one of a set of common dialog calls
- ◆ Tk main window & Toplevel popups have icon, title
- ◆ + multi-line text, menus, radio/check buttons, dialogs,...

### file: gui3.py

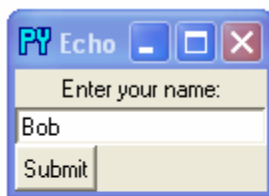
```
from Tkinter import *
from tkMessageBox import showinfo

def reply(name):
 showinfo(title='Reply', message='Hello %s!' % name)

top = Tk()
top.title('Echo')
top.iconbitmap('py-blue-trans-out.ico')

Label(top, text="Enter your name:").pack(side=TOP)
ent = Entry(top)
ent.pack(side=TOP)
btn = Button(top, text="Submit",
 command=(lambda: reply(ent.get())))
btn.pack(side=LEFT)

top.mainloop()
```





## More details

### Other event interfaces

- `widget.bind("<B1-Motion>", func)`    [\[CD bind.py\]](#)
- `widget.after(msecs, func)`
- *slider/widget linkage, data stream conditions*

### GUI construction

- ♦ *"What you build is what you get"*
- ♦ *Layout = build order + pack options*
- ♦ *Each Toplevel instance is a window*
- ♦ *Default Toplevel window if no parent*
- ♦ *Widgets nest in Frames, Frames nest in other Frames*
- ♦ *Object trees: Tkinter cross-links parents to children*

#### "Decreasing Cavity Model"

- *pack gives widget entire side*
- *widgets packed later get side of what's left*
- *grid, place: alternative geometry managers*

## OOP: Building GUIs by subclassing frames

- ♦ Organization framework: namespace, inheritance
- ♦ Methods attach widgets to *'self'*
- ♦ Callbacks are bound methods of *'self'*

#### file: hello.py

```
#!/usr/local/bin/python
from Tkinter import * # get widgets

class Hello(Frame): # container subclass
 def __init__(self, parent=None):
 Frame.__init__(self, parent) # superclass init
 self.pack()
 self.make_widgets() # attach to self
```

```

def make_widgets(self):
 widget = Button(self, text='Hello',
 command=self.onPress)
 widget.pack(side=LEFT)

def onPress(self):
 print 'Hi.' # write to stdout

if __name__ == '__main__': Hello().mainloop()

```

```

% hello.py
Hi.
Hi.

```



## ***OOP: Reusing GUIs by subclassing, 'is-a'***

- ◆ GUI classes can be extended by inheritance
- ◆ Subclasses may extend, or replace methods
- ◆ In OOP terms: a 'is-a' relationship

### **file: hellosub.py**

```

#!/usr/local/bin/python
from hello import Hello # get superclass
from Tkinter import * # get Tkinter widgets

class HelloExtender(Hello): # is-a hello.Hello
 def make_widgets(self): # extend method
 Hello.make_widgets(self)
 mine = Button(self, text='Extend',
 command=self.quit)
 mine.pack(side=RIGHT)

 def onPress(self):
 print 'Greetings!' # replace method

if __name__ == '__main__': HelloExtender().mainloop()

```

```
% python hellosub.py
Greetings!
Greetings!
```



## ***OOP: Reusing GUIs by attaching, 'has-a'***

- ◆ Frames can be attached to other Frames
- ◆ Tkinter records object tree internally
- ◆ In OOP terms: a 'has-a' relationship

### **file: hellouse.py**

```
#!/usr/local/bin/python
from hello import Hello # get class to attach
from Tkinter import * # get Tkinter widgets

class HelloContainer(Frame): # has-a hello.Hello
 def __init__(self, parent=None):
 Frame.__init__(self, parent)
 self.pack()
 self.make_widgets()

 def make_widgets(self):
 mine = Button(self, text='Attach',
 command=self.quit)
 mine.pack(side=LEFT)
 Hello(self) # attach a Hello to me

if __name__ == '__main__': HelloContainer().mainloop()
```

```
% hellouse.py
Hi.
Hi.
```



## Images

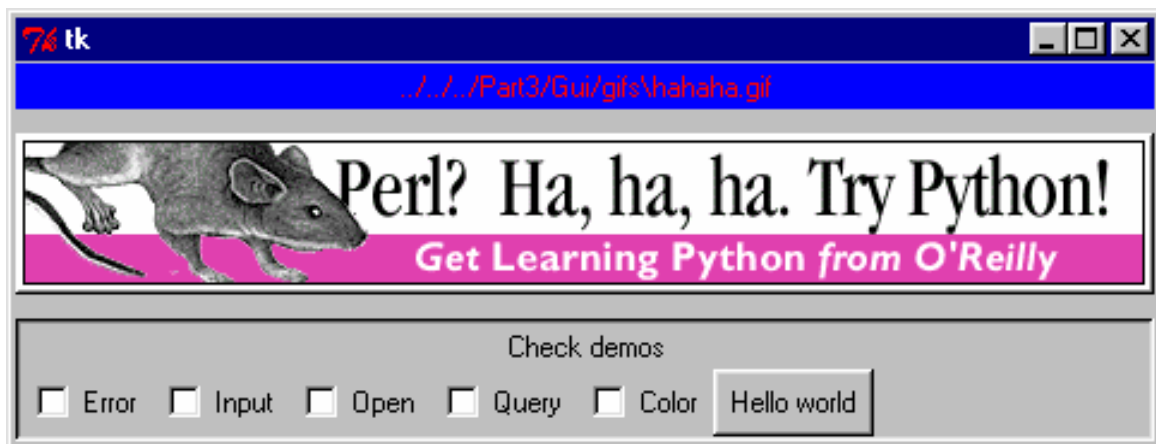
See “[Extras\Gui](#)” directory on class CD to run [this example](#)

```
from Tkinter import * # get base widget set
from glob import glob # file name expansion
import hellouseCheck # attach the last example to "me"
import random # pick a picture at random
gifdir = '../Part3/Gui/gifs/' # where to look for gif files

def draw():
 name, photo = random.choice(images)
 lbl.config(text=name)
 pix.config(image=photo)

root=Tk()
lbl = Label(root, text="none", bg='blue', fg='red')
pix = Button(root, text="Press me", command=draw, bg='white')
lbl.pack(fill=BOTH)
pix.pack(pady=10)
hellouseCheck.HelloContainer(root, relief=SUNKEN, bd=2).pack(fill=BOTH)

files = glob(gifdir + "*.gif")
images = map(lambda x: (x, PhotoImage(file=x)), files)
print files
root.mainloop()
```



## Grid layouts

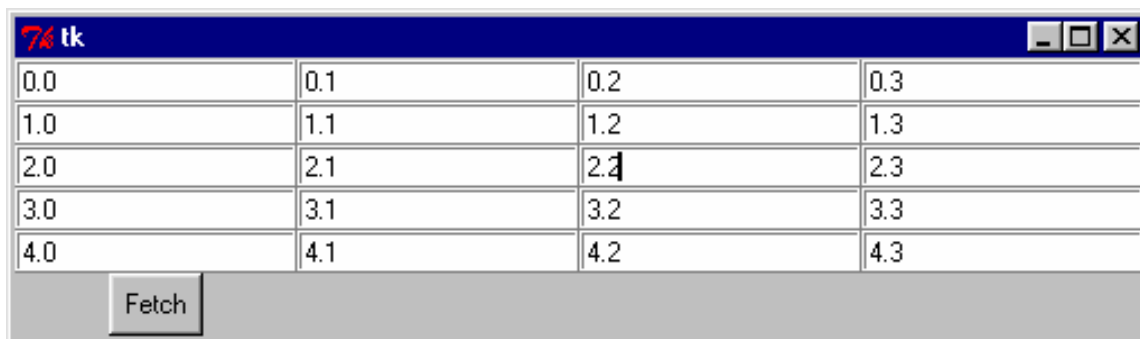
```
2d table of input fields

from Tkinter import *

rows = []
for i in range(5):
 cols = []
 for j in range(4):
 e = Entry(relief=RIDGE)
 e.grid(row=i, column=j, sticky=NSEW)
 e.insert(END, '%d.%d' % (i, j))
 cols.append(e)
 rows.append(cols)

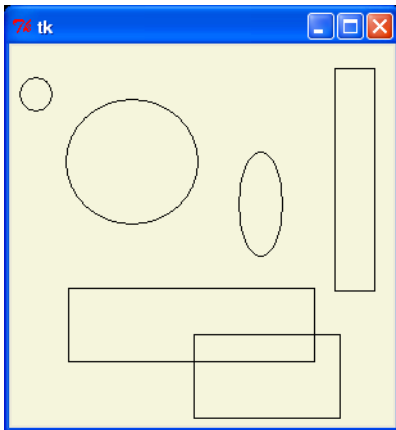
def onPress():
 for row in rows:
 for col in row:
 print col.get(),
 print

Button(text='Fetch', command=onPress).grid()
mainloop()
```



## Canvas, Text, dialogs, and Toplevel

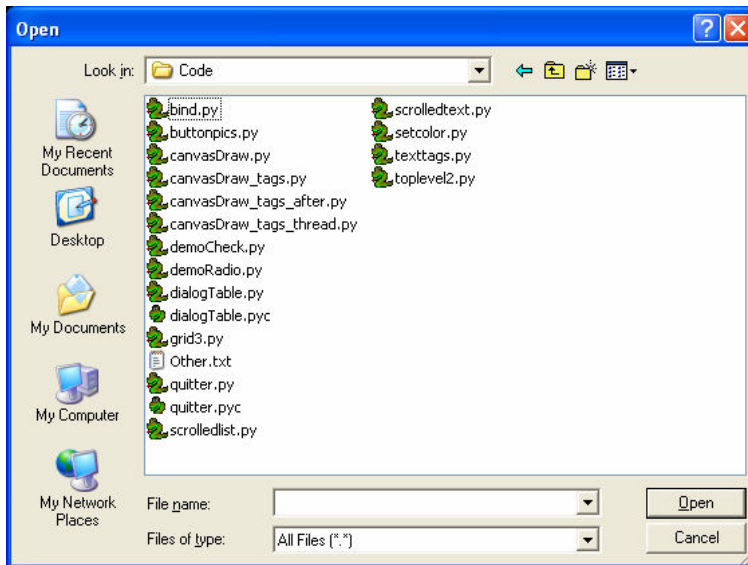
See [Extras\Gui\Code](#) to run [this example](#)



See [Extras\Gui\Code](#) to run [this example](#)



See [Extras\Gui\Code](#) to run [this example](#)



See [Extras\Gui\Code](#) to run [this example](#)

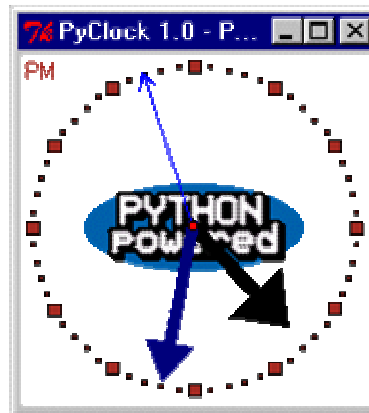


## Larger examples: Programming Python, Ed. 2+

See CD's top-level *PP3E-Examples* folder

Also available at <http://examples.oreilly.com/python3/>

A clock drawn on a canvas and updated with a timer, embedded picture

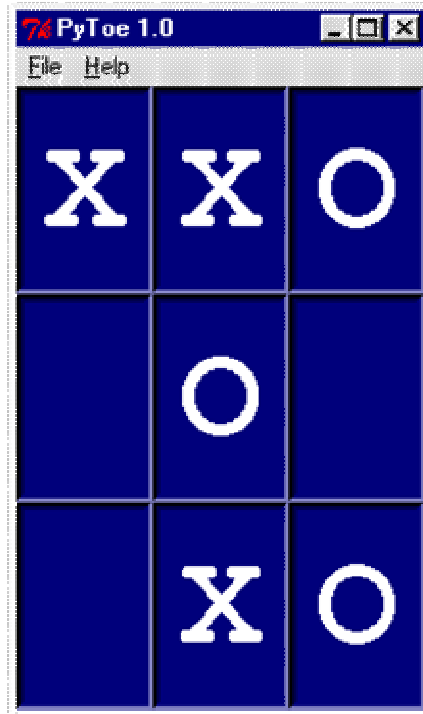


A calculator: frames of buttons, popup history, command lines, key bindings

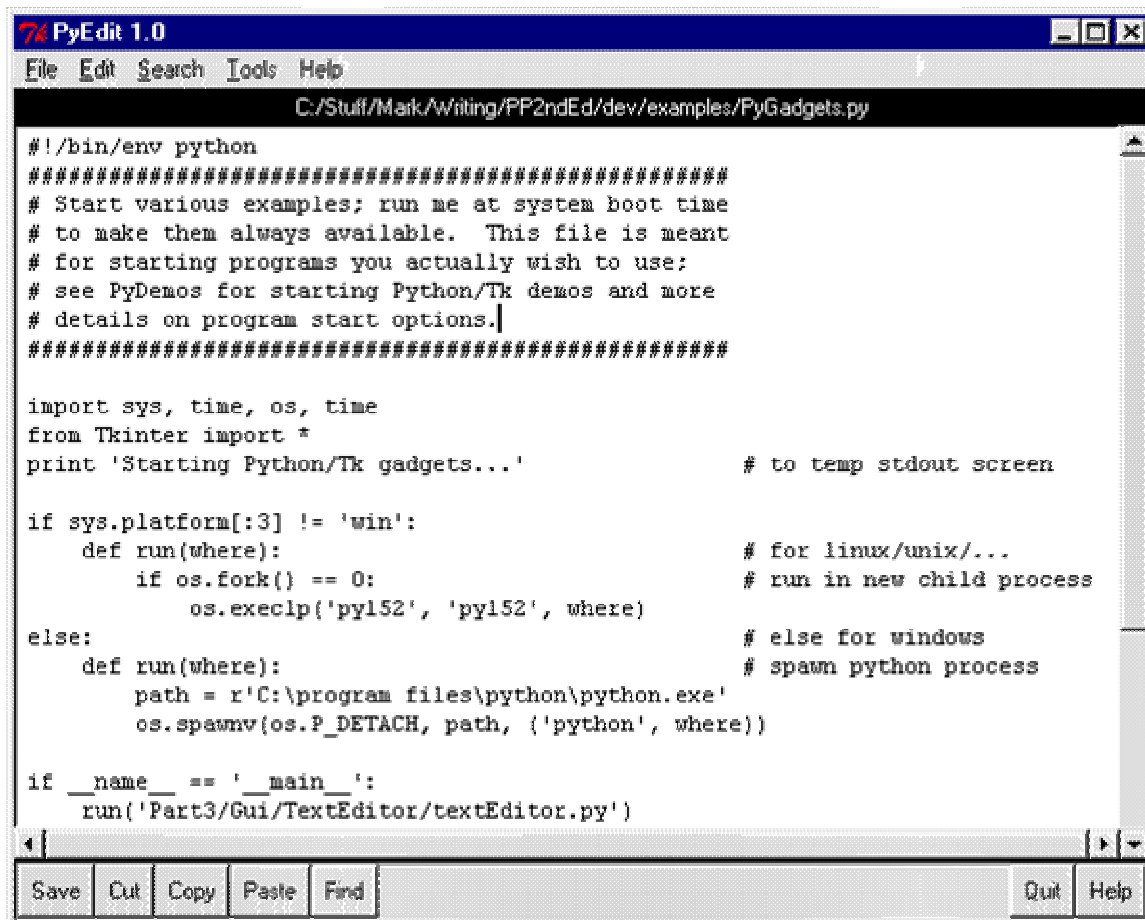


An artificially intelligent tic-tac-toe game: labels and mouse event bindings





A text editor object/program: menus, text, scrollbars, toolbar frame



```

PyEdit 1.0
File Edit Search Tools Help
C:/Stuff/Mark/Writing/PP2ndEd/dev/examples/PyGadgets.py

#!/bin/env python
#####
Start various examples; run me at system boot time
to make them always available. This file is meant
for starting programs you actually wish to use;
see PyDemos for starting Python/Tk demos and more
details on program start options.
#####

import sys, time, os, time
from Tkinter import *
print 'Starting Python/Tk gadgets...' # to temp stdout screen

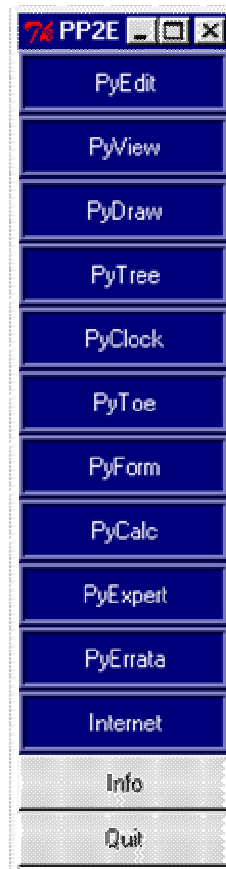
if sys.platform[:3] != 'win':
 def run(where): # for linux/unix/...
 if os.fork() == 0: # run in new child process
 os.execvp('py152', 'py152', where)
else:
 def run(where): # else for windows
 # spawn python process
 path = r'C:\program files\python\python.exe'
 os.spawnv(os.P_DETACH, path, ('python', where))

if __name__ == '__main__':
 run('Part3/Gui/TextEditor/textEditor.py')

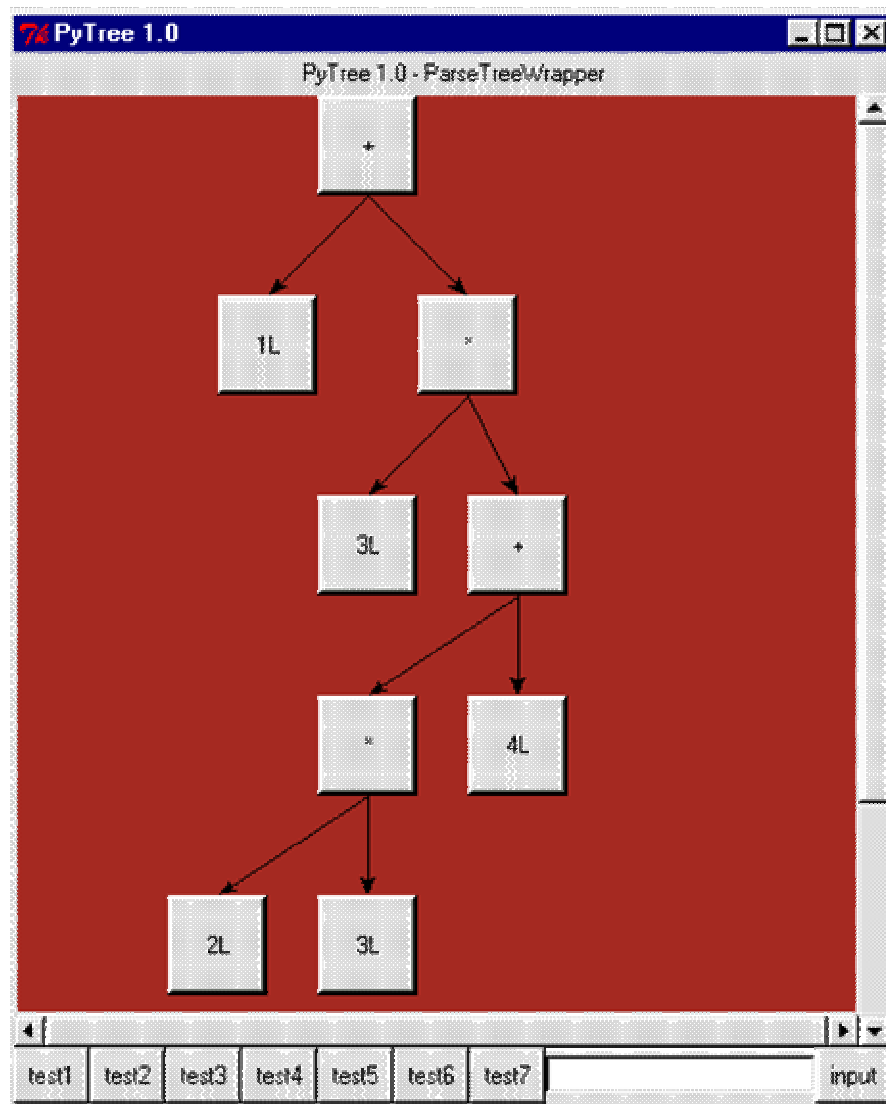
```

Save Cut Copy Paste Find Quit Help

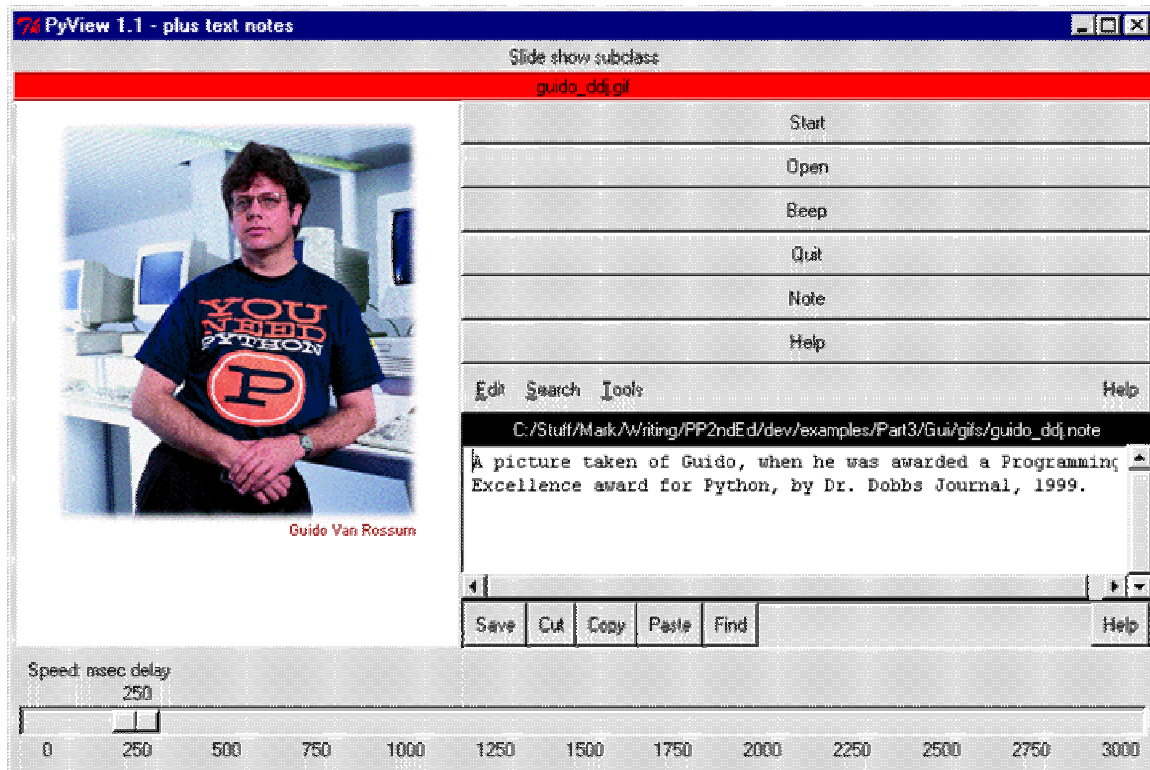
A demo launcher button bar: `os.fork()` on Linux, `os.spawnv()` on Windows



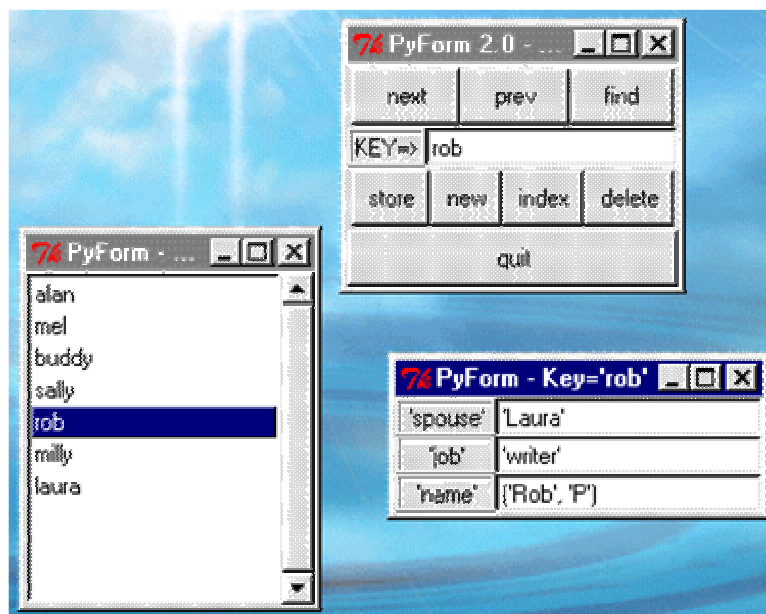
A generic tree data structure drawer: canvas, lines, event bindings



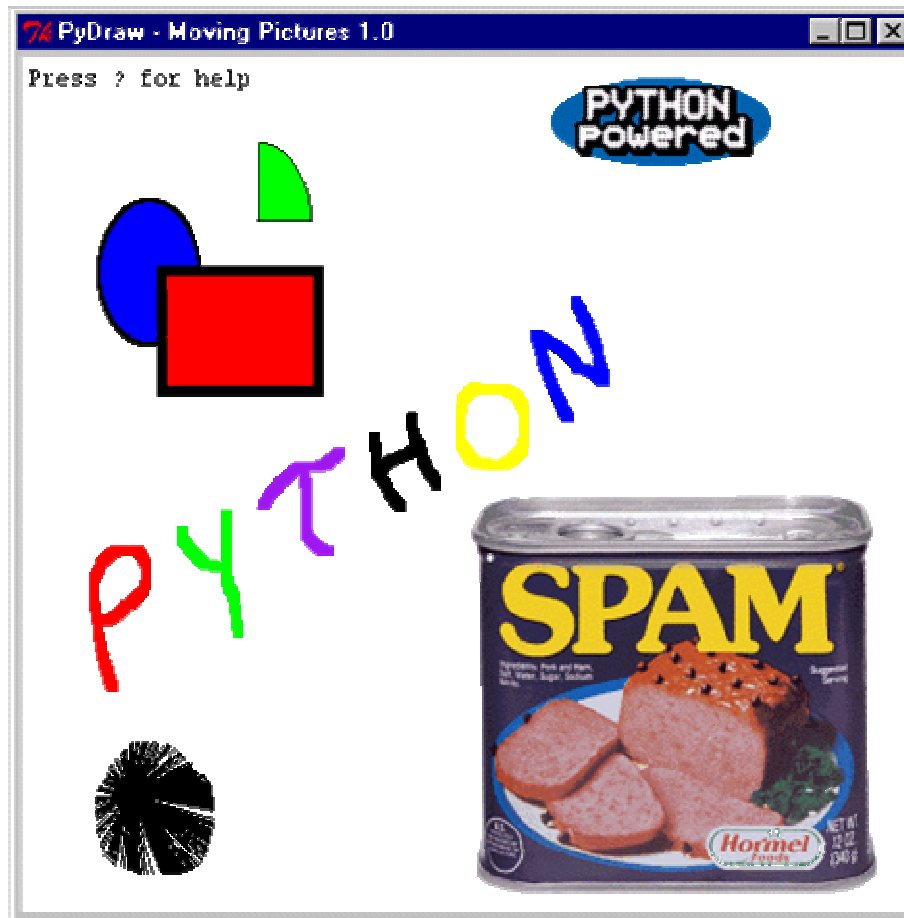
An image file slideshow: embeds text editor, picture, scale, timer, etc.



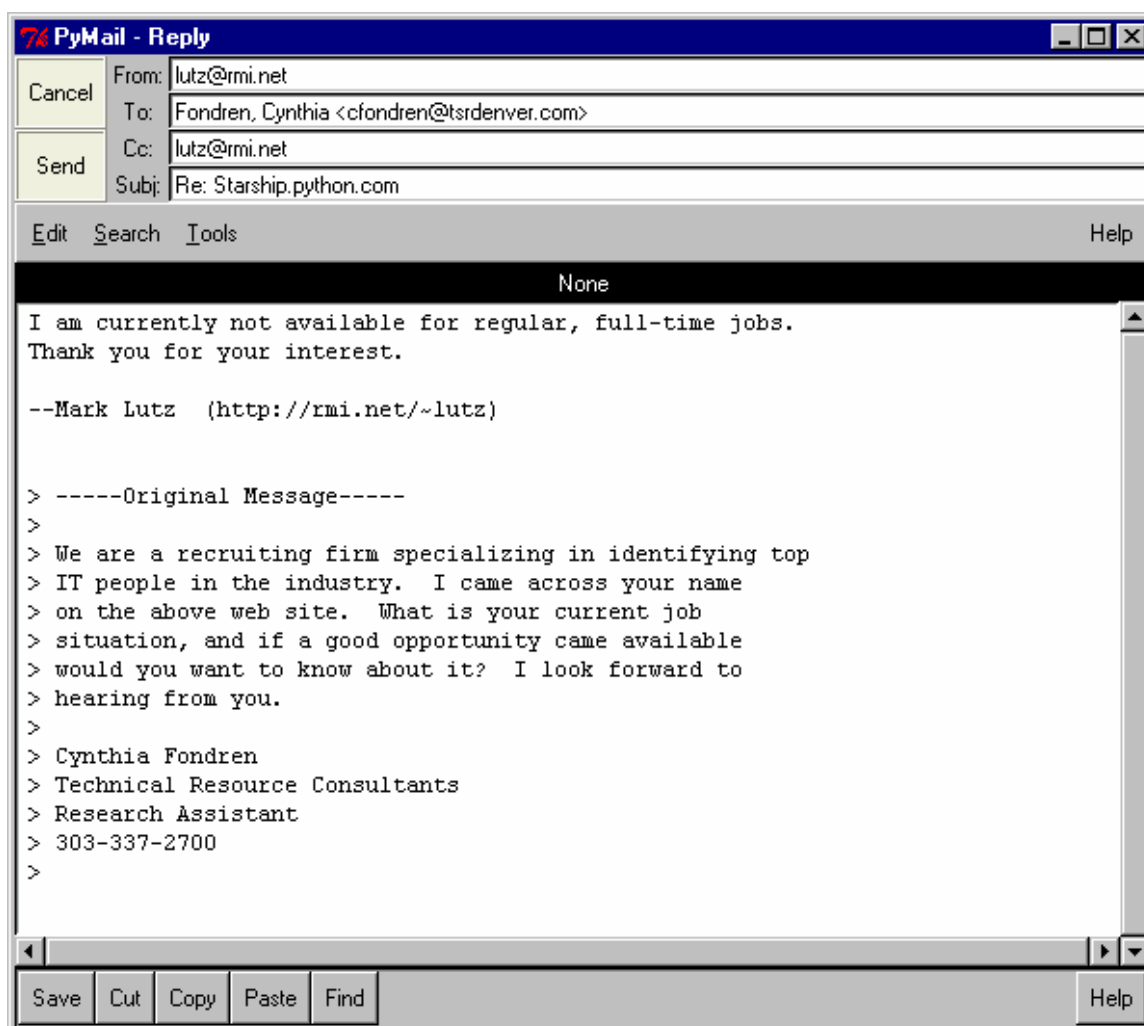
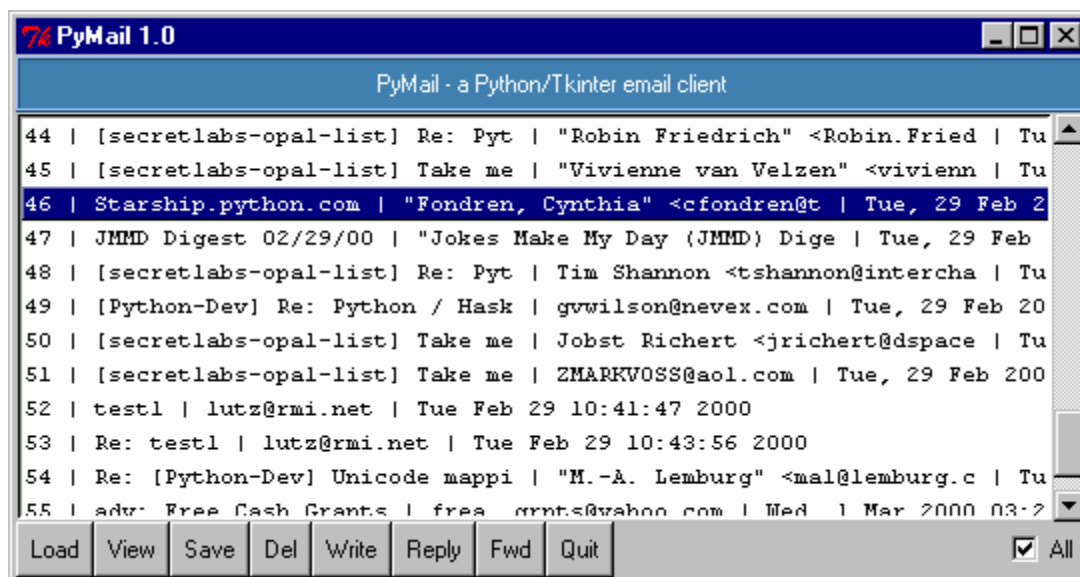
A table browser GUI: listboxes, forms, persistence, etc.



A drawing program: draw and move graphic objects



A POP/SMTP email client in Python/Tk, embeds text editor  
(a beta version is included in the class Internet Examples directory)



## Tkinter odds and ends

### ◆ Portability

- ◆ *Tkinter programs run with native look-and-feel on X, MS-Windows, Macintosh*

### ◆ Other GUI widgets

- ◆ *25 widgets + dialogs, etc.: Text, Canvas, Listbox, Scrollbar, Image, Menu, Entry, Radiobutton, Checkbutton, Scale,...*

### ◆ Other Tkinter tools

- ◆ *File handlers, scheduled events, “grid” manager*
- ◆ *Pmw (Python Mega Widgets), PIL (imaging), etc.; see Vaults site*
- ◆ *Interactive GUI builders: [PythonWorks?](#), [ActiveState Komodo](#)*
- ◆ *BoaConstructor, BlackAdder Gui buildes for wxPython, PyQt*

### ◆ Implementation structure

- ◆ *Tk + C extension module + Python wrapper classes module*
- ◆ *Uses extending (Tk lib) plus embedding (route events to handlers)*

### ◆ Documentation

- ◆ *“Programming Python 2<sup>nd</sup> Edition”, Part II, 250 pages*
- ◆ *“Programming Python 3rd Edition”, Part II, 300 pages*
- ◆ *Other books: Manning book, upcoming O'Reilly book?*
- ◆ *Tutorial, reference: <http://www.pythonware.com/library>*
- ◆ *Tcl/Tk books and manuals provide basic widget docs*

### ◆ Other links

- [Extras\Gui\wxPython\wxPython.doc](#) (Tkinter/wxPython comparison)
- [Extras\Gui\Code](#) (additional Tkinter examples)



<http://examples.oreilly.com/python3/> (*book examples*)

**See Also:**

CD\PP3E-Examples\Distribution\PP3E-Examples-1.1\Examples\PP3E\Launch\_PyDemos.pyw  
Tkinter book examples on CD

wxPython.org  
wxPython demo program

## **Lab Session 8**

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 12. Databases and persistence

### Topics

- ◆ Persistent object shelves
- ◆ Storing class instances
- ◆ Pickling without shelves
- ◆ DBM-style files
- ◆ Shelf gotchas
- ◆ Python SQL database API
- ◆ ZODB object-oriented database
- ◆ Odds and ends

### ***Object persistence: shelves***

- ◆ Shelf = dbm file + object pickling (serialization)
- ◆ Stores arbitrary Python objects by string key
- ◆ Shelves are processed with normal Python code
- ◆ Pickler handles nested and circular objects

### **Basic usage**

- ◆ Shelves are *dictionaries* that must be *opened*

```
import shelve
dbase = shelve.open("mydbase")
```
- ◆ Assigning to a shelf key *stores* an object

```
dbase['key'] = object
```

- ◆ Indexing a shelf *fetches* a stored object

```
value = dbase['key']
```

- ◆ Most dictionary operations supported

```
len(dbase) # number items stored
dbase.keys() # stored item key index
```

- ◆ **Creating a new shelf**

- ◆ *Creates dbm file(s) to store objects*

- ◆ *Manual 'close' only needed for bsddb*

```
% python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> object1 = ['The', 'bright', ('side', 'of'), ['life']]
>>> object2 = {'name': 'Brian', 'age': 33, 'motto': object1}
>>> dbase['brian'] = object2
>>> dbase['knight'] = {'name': 'Knight', 'motto': 'Ni!'}
>>> dbase.close()
```

- ◆ **Using an existing shelf**

```
% python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> len(dbase) # entries
2

>>> dbase.keys() # index
['knight', 'brian']

>>> dbase['knight'] # fetch
{'motto': 'Ni!', 'name': 'Knight'}

>>> for row in dbase.keys():
... print row, '=>'
... for field in dbase[row].keys():
... print ' ', field, '=', dbase[row][field]
...
knight =>
 motto = Ni!
 name = Knight
brian =>
 motto = ['The', 'bright', ('side', 'of'), ['life']]
 age = 33
 name = Brian
```

## Storing class instances

- ◆ Classes defined at top-level of module
- ◆ No non-default constructor arguments (until 1.5.1!)
- ◆ Python stores instance `__dict__`, not class
- ◆ Changing class changes stored object behavior

### file: person.py

# a person object: fields + behavior

```
class Person:
 def __init__(self, name='', job='', pay=0):
 self.name = name
 self.job = job
 self.pay = pay # real instance data
 def tax(self):
 return self.pay * 0.25 # computed on call
 def info(self):
 return self.name, self.job, self.pay, self.tax()
```

% python

```
>>> from person import Person
>>> bob = Person('bob', 'psychologist', 70000)
>>> emily = Person('emily', 'teacher', 40000)
>>>
>>> import shelve
>>> dbase = shelve.open('cast') # make new shelve
>>> for obj in (bob, emily): # store objects
>>> dbase[obj.name] = obj
>>> dbase.close() # need for bsddb
```

% python

```
>>> import shelve
>>> dbase = shelve.open('cast') # reopen shelve
>>> print dbase['bob'].tax() # call: bob's tax
17500.0
```

## Changing classes changes behavior

- ◆ Classes = records + processing programs
- ◆ Example: changing the Person.tax method

### file: person.py

# a person object: fields + behavior  
 # change: the tax method is now a virtual member

```
class Person:
 def __init__(self, name='', job='', pay=0):
 self.name = name
 self.job = job
 self.pay = pay # real instance data
 def __getattr__(self, attr): # on person.attr
 if attr == 'tax':
 return self.pay * 0.30 # computed on access
 else:
 raise AttributeError # other unknown names
 def info(self):
 return self.name, self.job, self.pay, self.tax
```

### % python

```
>>> import shelve
>>> dbase = shelve.open('cast') # reopen shelve
>>> print dbase['bob'].tax # no need to call tax()
21000.0
```

## Pickling objects without shelves

- ♦ *Shelve = dbm file + object pickler*
- ♦ *Pickler serializes Python objects into text streams*
- ♦ *Streams may be sent to flat file, dbm file, socket, etc.*
- ♦ *Also see: 'marshal' module (object limitations)*

### file: testpickle.py

```
import pickle
```

```
def saveDbase(filename, table):
 file = open(filename, 'w')
 pickle.dump(table, file) # pickle to file
 file.close()
```

```
def loadDbase(filename):
 file = open(filename, 'r')
 table = pickle.load(file) # unpickle from file
 file.close()
 return table
```

```
% python
```

```
>>> from testpickle import *
>>> L = [0]
>>> D = {'x':0, 'y':L}
>>> table = {'A':L, 'B':D} # L appears twice (D)
>>> saveDbase('myfile', table) # serialize to file
```

```
% python
```

```
>>> from testpickle import *
>>> table = loadDbase('myfile') # reload/unpickle
>>> print table
{'B': {'x': 0, 'y': [0]}, 'A': [0]}
>>> table['A'][0] = 1 # change shared list
>>> saveDbase('myfile', table)
```

```
% python
```

```
>>> from testpickle import *
>>> print loadDbase('myfile') # both L's updated!
{'B': {'x': 0, 'y': [1]}, 'A': [1]}
```

## Using simple dbm files

- ◆ *Shelve = dbm file + object pickler*
- ◆ *Dbm files can only store strings (not any object)*
- ◆ *'anydbm' selects from dbhash, gdbm, dbm or dumbdbm*
- ◆ *Dictionary-like interface like shelve (shelve uses dbm)*

```
% python
>>> import anydbm
>>> file = anydbm.open('languages', 'c') # create, in+out
>>> file['perl'] = "Text processing" # store
>>> file['tcl'] = "Simple glue"
>>> file['python'] = "OO scripting"
>>> file.close()
```

```
% python
>>> import anydbm
>>> file = anydbm.open('languages', 'c') # existing file
>>> file['python'] # fetch
'OO scripting'
>>> for lang in file.keys(): # index
... print lang + ': ', file[lang]
...
perl: Text processing
tcl: Simple glue
python: OO scripting
>>> del file['tcl'] # delete
```

```
% python
>>> import anydbm
>>> anydbm.open('languages', 'c').keys() # sorry, tcl!
['python', 'perl']
```

## ***Shelve gotchas***

- ◆ Keys must still be strings

```
dbase[42] = value # fails
```

- ◆ Shared objects are only noticed within a given slot

```
dbase[key] = object # store parts just once
dbase[key] = object # two copies of object!
```

- ◆ Updates must treat as fetch-modify-store mappings

```
dbase[key].attr = value # shelve unchanged!

object = dbase[key] # fetch it
object.attr = value # modify it
dbase[key] = object # store back
```

- ◆ Doesn't support simultaneous updates (e.g. CGI scripts)

### **Plus class pickling rules**

- ◆ At the top-level of a module (importable)
- ◆ No non-default constructor arguments (until 1.5.1!)
- ◆ Class changes must be backward-compatible
- ◆ Classes can use special methods to break constructor rule

## ***ZODB object-oriented database***

- *Full-blown OODB, 3<sup>rd</sup> party open source add-on*
- ◆ *ZODB – Zope's advanced OODB, available by itself*
- ◆ *Adds object identifiers to Python pickling: write-thru on change*



- ◆ ***Supports concurrent update, transaction commit/rollback***
- ◆ ***Like shelve, but extra boilerplate code***
- ◆ ***Concurrent updates for multi-threaded***
- ◆ ***ZEO distributed objects server for multi-process***
- ◆ ***Can map ZODB db to flat files, relation dbase, other***
- ◆ ***Legacy from the web: faster at reads than writes***

## Creating a ZODB database

```

...\PP3E\Database\ZODBscripts\>python
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage(r'C:\Mark\temp\mydb.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()

>>> object1 = (1, 'spam', 4, 'YOU')
>>> object2 = [[1, 2, 3], [4, 5, 6]]
>>> object2.append([7, 8, 9])
>>> object2
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
>>> object3 = {'name': ['Bob', 'Doe'],
 'age': 42,
 'job': ('dev', 'mgr')}

>>> root['mystr'] = 'spam' * 3
>>> root['mytuple'] = object1
>>> root['mylist'] = object2
>>> root['mydict'] = object3
>>> root['mylist']
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> import transaction
>>> transaction.commit()
>>> storage.close()

```

## Using a ZODB database

```

...\PP3E\Database\ZODBscripts\>python
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage(r'C:\Mark\temp\mydb.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root() # connect

>>> print len(root), root.keys() # size, ix
4 ['mylist', 'mystr', 'mytuple', 'mydict']
>>>
>>> print root['mylist'] # fetch obs

```

```

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> print root['mydict']
{'job': ('dev', 'mgr'), 'age': 42, 'name': ['Bob', 'Doe']}

>>> root['mydict']['name'][-1] # last name
'Doe'

>>> for key in root.keys():
 print key.ljust(10), '=>', root[key]

mylist => [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
mystr => spamspamspace
mytuple => (1, 'spam', 4, 'YOU')
mydict => {'job': ('dev', 'mgr'), 'age': 42, 'name': ['Bob', 'Doe']}

```

## Classes under ZODB

### # zodb\_class\_make

```

import time
mydbfile = 'data/class.fs' # dbase file
from persistent import Persistent

class BookMark(Persistent): # use zodb
 def __init__(self, title, url):
 self.hits = 0
 self.updateBookmark(self, url)
 def updateBookmark(self, title, url):
 self.title = title # updates db
 self.url = url
 self.modtime = time.asctime()

def connectdb(dbfile):
 from ZODB import FileStorage, DB
 storage = FileStorage.FileStorage(dbfile)
 db = DB(storage)
 connection = db.open()
 root = connection.root()
 return root, storage

def addobjects():
 root, storage = connectdb(mydbfile)
 root['ora'] = BookMark('Oreilly', 'http://www.oreilly.com')
 root['pp3e'] = BookMark('PP3E',
 'http://www.rmi.net/~lutz/about-pp.html')

 import transaction
 transaction.commit()
 storage.close()

```

### # zodb\_class\_read

```

mydbfile = 'data/class.fs'
from zodb_class_make import connectdb
root, storage = connectdb(mydbfile)

```

```
this updates db: attrs changed in method
print 'pp3e url:', root['pp3e'].url
print 'pp3e mod:', root['pp3e'].modtime
root['pp3e'].updateBookmark('PP3E',
 'www.rmi.net/~lutz/about-pp3e.html')

this updates too: attr changed here
ora = root['ora']
print 'ora hits:', ora.hits
ora.hits += 1

commit changes made
import transaction
transaction.commit()
storage.close()
```

## Python SQL database API

- *Support for Oracle, Sybase, Informix, ODBC, Postgres, mySql,...*
- *Portable API Implemented by vendor-specific extension modules*

### Connection objects

*represent a connection to a database*  
*are the interface to rollback and commit operations*  
*generate cursor objects*

### Cursor objects

*represent a single SQL statement submitted as a string*  
*can be used to step through SQL statement results*  
*can execute DDL (create), DML (insert), and DQL (select) statements*

### Query results

*SQL select results returned to scripts as Python data*  
*Tables of rows are Python lists of tuples*  
*Field values within rows are normal Python objects*  
*An example query result: [('bob',38), ('emily',37)]*

```
#####
example of portable database API usage
#####

from cxoracle import Connect
connobj = Connect("user/password@system")
cursobj = connobj.cursor()

value1,value2 ='developer', 40
query = 'SELECT name, shoesize FROM empl WHERE job = ? AND age = ?'

cursobj.execute(query, (value1, value2))
results = cursobj.fetchall()
for (name, size) in results:
 print name, size
```

## Other DB API concepts

- *Select results: fetchall(), fetchone(), fetchmany()*
- *Transactions: commit, rollback on connection object*
- *update, insert, delete statements: rowcount*
- *Stored procedure methods*
- *Blobs*
- *Thread Safety check*
- *Description field: column names, types*

## Other SQL DB concepts

- *PostgreSQL interface also available online*
- *SQLObject 3<sup>rd</sup> party extension object relational mapper: table/row/column  
→ class/instance/attr*

**Full API at [python.org](http://python.org): search Google for “Python Database API” -- PEP, 2.0**

## Sqlite – standard library in Python 2.5

- **In-process C lib based SQL engine, no server**
- **Useful for internal program data, prototyping**

```
import sqlite3
conn = sqlite3.connect('/tmp/example') # or ":memory:"
c = conn.cursor()
```

```

Create table
c.execute('''create table stocks
 (date text, trans text, symbol text,
 qty real, price real)''')

Insert a row of data
c.execute("""insert into stocks
 values ('2006-01-05','BUY','RHAT',100,35.14)""")

run a query
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
 ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
 ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
):
 c.execute('insert into stocks values (?,?,?,?,?)', t)

```

## MySQL

- Server-based database engine
- Install MySQL, mysql-python

```

from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='python')
curs = conn.cursor()
try:
 curs.execute('drop database testpeopledb')
except:
 pass # did not exist

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)',
 ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)',
 ('Sue', 'dev', 60000))

curs.execute('select * from people')
for row in curs.fetchall():
 print row

curs.execute('select * from people')
colnames = [desc[0] for desc in curs.description]
while True:
 print '-' * 30
 row = curs.fetchone()
 if not row: break
 for (name, value) in zip(colnames, row):
 print '%s => %s' % (name, value)

conn.commit() # save inserted records

```

## ***Persistence odds and ends***

- ◆ **Flat files**
  - ◆ *Objects and descriptors*
  
- ◆ **'marshal' module**
  - ◆ *Serializes objects like pickler, but only simple types*
  
- ◆ **Performance**
  - ◆ *'cPickle' module is 1 to 3 orders of magnitude faster than 'pickle' (auto selected by shelve)*
  
- ◆ **Database interfaces available**
  - ◆ *At python.org: Oracle, Sybase, PostgreSQL Informix, ODBC, MySQL, etc.*
  - ◆ *(See above) Portable database API: works on many systems*
  - ◆ *All-Python "gadfly" SQL database system (in memory)*

## ***Lab Session 9***

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 13. Text processing

### Topics

- ◆ String objects
- ◆ Splitting strings
- ◆ Regular expressions
- ◆ Parsing languages
- ◆ XML parsing (see Internet section)

### *String objects: review*

- ◆ Handle basic text processing tasks
- ◆ Operations: slicing, concatenation, indexing, etc.
- ◆ String methods: searching, replacement, splitting, etc.
- ◆ Built-in string functions: ord()
- ◆ Running code strings: 'eval', 'exec', 'execfile'
- ◆ Unicode (16-bit char) wide strings supported in Python 2.0

```
>>> text = "Hello world"
>>> text = 'M' + text[1:6] + 'World'
>>> text
'Mello World'
>>> exec 'print "J" + text[1:]'
Jello World
```

## Splitting and joining strings

- ◆ `str.split` returns a list of columns: around whitespace
- ◆ `str.split` allows arbitrary delimiters to be used
- ◆ `str.join` puts string lists back together
- ◆ `eval` converts column strings to Python objects

### Example: column sum alternatives

#### Example: summing columns in a file

*# see also: newer column summer code at end of Basic Statements chapter*

##### file: summer.py

```
import sys

def summer(numCols, fileName):
 sums = [0] * numCols
 for line in open(fileName, 'r'):
 cols = line.split()
 for i in range(numCols):
 sums[i] += eval(cols[i]) # any expression will work!
 return sums

if __name__ == '__main__':
 print summer(eval(sys.argv[1]), sys.argv[2])
```

#### Example: replacing substrings

##### file: replace.py

```
manual global substitution

def replace(str, old, new):
 list = str.split(old) # XoldY -> [X, Y]
 return new.join(list) # [X, Y] -> XnewY
```



## Example: analyzing data files

- ◆ *Collect all entries for keys on right*
- ◆ *Data file contains “histogram” data*

```
% cat histo1.txt
```

```
1 one
2 one
3 two
7 three
8 two
10 one
14 three
19 three
20 three
30 three
```

```
% cat histo.py
```

```
#!/usr/bin/env python
import sys
```

```
entries = {}
for line in open(sys.argv[1]):
 [left, right] = line.split()
 try:
 entries[right].append(left) # or use has_key, or get
 except KeyError:
 entries[right] = [left] # e[r] = e.get(r, []) + [l]
```

```
for (right, lefts) in entries.items():
 print "%04d '%s'\titems => %s" % (len(lefts), right, lefts)
```

```
% histo.py histo1.txt
```

```
0003 'one' items => ['1', '2', '10']
0005 'three' items => ['7', '14', '19', '20', '30']
0002 'two' items => ['3', '8']
```

## Regular expressions

- ◆ For matching patterns in strings
- ◆ Matched substrings may be extracted after a match as “groups”
- ◆ Compiled regular expressions are first-class objects: optimization
- ◆ Now supported by the ‘re’ standard module: Perl5-style patterns
- ◆ Supports non-greedy operators, character classes, etc.
- ◆ Older options: the “regex”, ‘regex’ modules: emacs/awk/grep patterns

### Basic interface

```
>>> import re

>>> mobj = re.match('Hello(.*?)world', 'Hello---spam---world')
>>> mobj.group(1)
'---spam---'

>>> pobj = re.compile('Hello[\t]*(.*?)')
>>> mobj = pobj.match('Hello SPAM!')
>>> mobj.group(1)
'SPAM!'
```

### Example: searching C files

Finds #include and #define lines in a C file

### Operators

X+ repeat X one or more times

X\* repeat X zero or more times

[abc] any of a or b or c

(X) keep substring that matches X (“group”)

^X match X at start of line

**Methods**

re.compile                    precompiles expression into pattern object

patternobj.match            returns match object, or None if match fails

matchobj.group              returns matched substring[i] (pattern part in parens)

matchobj.span               returns start/stop indexes of match substring[i]

also has methods for replacement, nongreedy match operators,...

**file: cheader.py**

```
#!/usr/local/bin/python
import sys, re

pattDefine = re.compile(
 '^#[\t]*define[\t]+([a-zA-Z0-9_]+)[\t]*(.*)' # compile to pattobj
 # "# define xxx yyy..."

pattInclude = re.compile(
 '^#[\t]*include[\t]+<\"([a-zA-Z0-9_/\.]*)\"' # "# include <xxx>..."

def scan(file):
 count = 0
 while 1:
 line = file.readline()
 if not line: break
 count = count + 1
 matchobj = pattDefine.match(line) # None if match fails
 if matchobj:
 name = matchobj.group(1) # substrings for (...) parts
 body = matchobj.group(2)
 print count, 'defined', name, '=', body.strip()
 continue
 matchobj = pattInclude.match(line)
 if matchobj:
 start, stop = matchobj.span(1) # start/stop indexes of (...)
 filename = line[start:stop] # slice out of line
 print count, 'include', filename # same as matchobj.group(1)

if len(sys.argv) == 1:
 scan(sys.stdin) # no args: read stdin
else:
 scan(open(sys.argv[1], 'r')) # arg: input file name
```

## ***Parsing languages***

- ◆ For more demanding languages: regular expressions have no “memory”
- ◆ Recursive descent parsers: see *YAPPS* parser generator
- ◆ Parser generators: ‘bison’ wrapper, PyParsing, SPARK, kwParsing, etc. (see the web)
- ◆ *NLTK*: Natural Language Toolkit for Python, AI and statistical tools

## ***Lab Session 10***

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 14. Internet scripting

### Topics

- ◆ Sockets in Python
- ◆ The FTP module
- ◆ Email processing
- ◆ CGI scripts (server)
- ◆ Grail applets (client)
- ◆ Jython: Python for Java systems
- ◆ Active Scripting and COM
- ◆ Other tools: urllib, HTMLgen, XML, Zope

### ***Using sockets in Python***

- ◆ At the heart of Internet communications
- ◆ A standard C extension type: BSD socket wrapper
- ◆ *C functions* become socket object *methods*

### **Socket module details**

- ◆ Supports all types: TCP/IP, UDP datagram, Unix domain
- ◆ Also supports timeouts, non-blocking, SSL
- ◆ Encrypted sockets supported if SSL library enabled
- ◆ Urllib modules do https:// if secure sockets enabled
- ◆ Email also uses secure sockets if enabled
- ◆ Python also supports “select” multiplex processing

## Basic client/server example

- ♦ *Server: echoes all data that it receives back*
- ♦ *Client: sends data to the server*
- ♦ *Client calls: “socket”, “connect”*
- ♦ *Server calls: “socket”, “bind”, “listen”, “accept”*

### file: echoserver.py

```
#####
Server side: open a socket on a port, listen for
a message from a client, and send an echo reply;
this is a simple one-shot listen/reply per client,
but it goes into an infinite loop to listen for
more clients as long as this server script runs;
#####

from socket import * # get socket constructor and constants
myHost = '' # server machine, '' means local host
myPort = 50007 # listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP socket object
sockobj.bind((myHost, myPort)) # bind it to server port number
sockobj.listen(5) # listen, allow 5 pending connects

while 1:
 connection, address = sockobj.accept() # listen until process killed
 print 'Server connected by', address # wait for next client connect
 while 1: # connection is a new socket
 data = connection.recv(1024) # read next line on client socket
 if not data: break # send a reply line to the client
 connection.send('Echo=>' + data) # until eof when socket closed
 connection.close()
```

### file: echoclient.py

```
#####
Client side: use sockets to send data to the server, and
print server's reply to each message line; 'localhost'
means that the server is running on the same machine as
the client, which lets us test client and server on one
machine; to test over the net, run server on a remote
machine, set serverHost to machine's domain name or IP addr;
#####

import sys
```

```

from socket import * # portable socket interface plus constants
serverHost = 'localhost' # server name, or: 'starship.python.net'
serverPort = 50007 # non-reserved port used by the server

message = ['Hello network world'] # text to send to server
sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP/IP socket object
sockobj.connect((serverHost, serverPort)) # connect to server and port

for line in message:
 sockobj.send(line) # send line to server over socket
 data = sockobj.recv(1024) # receive from server: up to 1k
 print 'Client received:', `data`

sockobj.close() # close to send eof to server

```

## Forking socket servers

```

import os, time, sys, signal, signal
from socket import * # get socket constructor and constants
myHost = '' # server machine, '' means local host
myPort = 50007 # listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP socket object
sockobj.bind((myHost, myPort)) # bind to server port number
sockobj.listen(5) # up to 5 pending connects
signal.signal(signal.SIGCHLD, signal.SIG_IGN) # avoid child zombie processes

def now(): # time on server machine
 return time.ctime(time.time())

def handleClient(connection): # child process replies, exits
 time.sleep(5) # simulate a blocking activity
 while 1: # read, write a client socket
 data = connection.recv(1024)
 if not data: break
 connection.send('Echo=>%s at %s' % (data, now()))
 connection.close()
 os._exit(0)

def dispatcher(): # listen until process killed
 while 1: # wait for next connection,
 connection, address = sockobj.accept() # pass to process for service
 print 'Server connected by', address,
 print 'at', now()
 childPid = os.fork() # copy this process
 if childPid == 0: # if in child process: handle
 handleClient(connection) # else: go accept next connect

dispatcher()

```

## Threading socket servers

```
import thread, time
from socket import * # get socket constructor and constants
myHost = '' # server machine, '' means local host
myPort = 50007 # listen on a non-reserved port number

sockobj = socket(AF_INET, SOCK_STREAM) # make a TCP/IP socket object
sockobj.bind((myHost, myPort)) # bind to server port number
sockobj.listen(5) # upto 5 pending connects

def now():
 return time.ctime(time.time()) # current time on the server

def handleClient(connection):
 # in spawned thread: reply
 time.sleep(5) # simulate a blocking activity
 while 1: # read, write a client socket
 data = connection.recv(1024)
 if not data: break
 connection.send('Echo=>%s at %s' % (data, now()))
 connection.close()

def dispatcher():
 # listen until process killed
 while 1: # wait for next connection,
 connection, address = sockobj.accept() # pass to thread for service
 print 'Server connected by', address,
 print 'at', now()
 thread.start_new(handleClient, (connection,))

dispatcher()
```

## Select socket servers

```
event loop: listen and multiplex until server process killed
from select import select
print 'select-server loop starting'
while 1:
 readables, writeables, exceptions = select(readsocks, writesocks, [])
 for sockobj in readables:
 if sockobj in mainsocks: # for ready input sockets
 # port socket: accept new client
 newsock, address = sockobj.accept() # accept should not block
 print 'Connect:', address, id(newsock) # newsock is a new socket
 readsocks.append(newsock) # add to select list, wait
 else:
 # client socket: read next line
 data = sockobj.recv(1024) # recv should not block
 print '\tgot', data, 'on', id(sockobj)
 if not data: # if closed by the clients
 sockobj.close() # close here and remv from
 readsocks.remove(sockobj) # del list else reselected
 else:
 # this may block: should really select for writes too
 sockobj.send('Echo=>%s at %s' % (data, now()))
```



## Standard library server types

- ◆ Threading/Forking TCP/UDP server classes
- ◆ Asyncore select-based server classes
- ◆ See also: HTTP/CGI webserver.py in CGI section below
- ◆ See also: Twisted 3<sup>rd</sup> party system

```
import SocketServer, time # get socket server, handler objects
myHost = '' # server machine, '' means local host
myPort = 50007 # listen on a non-reserved port number
def now():
 return time.ctime(time.time())

class MyClientHandler(SocketServer.BaseRequestHandler):
 def handle(self):
 # on each client connect
 print self.client_address, now() # show this client's addr
 time.sleep(5) # simulate blocking activity
 while 1:
 # self.request is client socket
 data = self.request.recv(1024) # read, write a client socket
 if not data: break
 self.request.send('Echo=>%s at %s' % (data, now()))
 self.request.close()

make a threaded server, listen/handle clients forever
myaddr = (myHost, myPort)
server = SocketServer.ThreadingTCPServer(myaddr, MyClientHandler)
server.serve_forever()
```

## The FTP module

- ◆ *ftplib* library module uses sockets to transfer files
- ◆ Supports both binary and text retrieve/store operations
- ◆ Handles all handshaking with the remote site
- ◆ See Python library manual for more details

## Example: fetch file

### file: getone.py

```
import os, sys
from getpass import getpass

nonpassive = False
filename = 'lawnlake2-jan-03.jpg' # file to download
dirname = '.' # remote directory
sitename = 'ftp.rmi.net' # ftp site to contact
userinfo = ('lutz', getpass('Pswd?')) # use () for anonymous
if len(sys.argv) > 1: filename = sys.argv[1] # file on command-line?

print 'Connecting...'
from ftplib import FTP # socket-based ftp tools
localfile = open(filename, 'wb') # local file to store to
connection = FTP(sitename) # connect to ftp site
connection.login(*userinfo) # default anonymous login
connection.cwd(dirname) # xfer 1k at a time
if nonpassive: # if server requires
 connection.set_pasv(False)

thread me in a GUI
print 'Downloading...'
connection.retrbinary('RETR ' + filename, localfile.write, 1024)
connection.quit()
localfile.close()
```

## Example: simple FTP site mirror (PP book)

```
#!/bin/env python
#####
use ftp to copy all files from a remote site/directory to a local dir;
e.g., run me periodically from a unix cron job to mirror an ftp site;
script assumes this is a flat directory--see the mirror program in
Python's Tools directory for a version that handles subdirectories;
#####

import os, sys, ftplib
from getpass import getpass

remotesite = 'home.rmi.net'
remotedir = 'public_html'
remoteuser = 'lutz'
remotepass = getpass('Please enter password for %s: ' % remotesite)
localdir = (len(sys.argv) > 1 and sys.argv[1]) or '.'
cleanall = raw_input('Clean local directory first? ')[1] in ['y', 'Y']

print 'connecting...'
connection = ftplib.FTP(remotesite) # connect to ftp site
connection.login(remoteuser, remotepass) # login as user/password
connection.cwd(remotedir) # cd to directory to copy

if cleanall:
 for localname in os.listdir(localdir):
 try: # try to delete all locals
 os.remove(localname) # to remove old files
```

```

 print 'deleting local', localname
 os.remove(os.path.join(localdir, localname))
 except:
 print 'cannot delete local', localname

count = 0
remotefiles = connection.nlst()

download remote files
nlst() gives files list
dir() gives all details

for remotename in remotefiles:
 localname = os.path.join(localdir, remotename)
 print 'copying', remotename, 'to', localname
 if remotename[-4:] == 'html' or remotename[-3:] == 'txt':
 # use ascii mode xfer
 localfile = open(localname, 'w')
 callback = lambda line, file=localfile: file.write(line + '\n')
 connection.retrlines('RETR ' + remotename, callback)
 else:
 # use binary mode xfer
 localfile = open(localname, 'wb')
 connection.retrbinary('RETR ' + remotename, localfile.write)
 localfile.close()
 count = count+1

connection.quit()
print 'Done:', count, 'files downloaded.'
```

## Example: upload site by FTP (PP book)

```

#!/bin/env python
#####
use ftp to upload all files from a local dir to a remote site/directory;
e.g., run me to copy an ftp site's files from your machine to your ISP,
especially handy if you only have ftp access to your website, not a
telnet/shell account access (else you could tar up all files and
transfer in a single step to the remote machine and untar there);
to upload subdirectories too, use os.path.isdir(path), FTP().mkd(path),
and recursion--see uploadall.py for a version that supports subdirs.
#####

import os, sys, ftplib, getpass

remotesite = 'starship.python.net' # upload to starship site
remotedir = 'public_html/home' # from win laptop or other
remoteuser = 'lutz'

remotepass = getpass.getpass('Please enter password for %s: ' % remotesite)
localdir = (len(sys.argv) > 1 and sys.argv[1]) or '.'
cleanall = raw_input('Clean remote directory first? ')[0:1] in ['y', 'Y']

print 'connecting...'
connection = ftplib.FTP(remotesite) # connect to ftp site
connection.login(remoteuser, remotepass) # login as user/password
connection.cwd(remotedir) # cd to directory to copy

if cleanall:
 for remotename in connection.nlst():
 # try to delete remotes
 # to remove old files
 try:
 print 'deleting remote', remotename
```

```

 connection.delete(remotename)
 except:
 print 'cannot delete remote', remotename

count = 0
localfiles = os.listdir(localdir) # upload all local files
 # listdir() strips dirpath

for localname in localfiles:
 localpath = os.path.join(localdir, localname)
 print 'uploading', localpath, 'to', localname
 if localname[-4:] == 'html' or localname[-3:] == 'txt':
 # use ascii mode xfer
 localfile = open(localpath, 'r')
 connection.storlines('STOR ' + localname, localfile)
 else:
 # use binary mode xfer
 localfile = open(localpath, 'rb')
 connection.storbinary('STOR ' + localname, localfile, 1024)
 localfile.close()
 count = count+1

connection.quit()
print 'Done:', count, 'files uploaded.'

```

## ***Email processing***

- ◆ POP, IMAP (retrieve) SMTP (send), on top of sockets
- ◆ Newer email.\* module package: parse+compose, attachments, en/decoding ...

### **Reading a POP email account**

```

#!/usr/local/bin/python
#####
use the Python POP3 mail interface module to view
your pop email account messages;
#####

import poplib, getpass, sys, mailconfig

mailserver = mailconfig.popservername # ex: 'pop.rmi.net'
mailuser = mailconfig.popusername # ex: 'lutz'
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print 'Connecting...'
server = poplib.POP3(mailserver)
server.user(mailuser) # connect, login to server
server.pass_(mailpasswd) # pass is a reserved word
try:
 print server.getwelcome() # print greeting message

```

```

msgCount, msgBytes = server.stat()
print 'There are', msgCount, 'mail messages in', msgBytes, 'bytes'
print '-'*80
raw_input('[Press Enter key]')

for i in range(msgCount):
 hdr, message, octets = server.retr(i+1)
 for line in message: print line # retrieve, print all
 print '-'*80 # mbox locked till quit
 if i < msgCount - 1:
 raw_input('[Press Enter key]')
finally:
 server.quit() # make sure to unlock
print 'Bye.'

```

## Sending email via a SMTP server

```

#!/usr/local/bin/python
#####
use the Python SMTP mail interface module to send mail
#####

import smtplib, string, sys, time, mailconfig
mailserver = mailconfig.smtpservername # ex: starship.python.net

From = string.strip(raw_input('From? ')) # ex: lutz@rmi.net
To = string.strip(raw_input('To? ')) # ex: guido@python.org
To = string.split(To, ',') # allow a list of tos
Subj = string.strip(raw_input('Subj? '))

prepend standard headers
date = time.ctime(time.time())
text = ('From: %s\nTo: %s\nDate: %s\nSubject: %s\n'
 % (From, string.join(To, ','), date, Subj))
text = text + '\n' # blank line between hdrs,body

print 'Type message text, end with line=(ctrl + D or Z)'
while 1:
 line = sys.stdin.readline()
 if not line:
 break
 text = text + line
 # exit on ctrl-d
 # servers do this auto

if sys.platform[:3] == 'win': print
print 'Connecting...'
server = smtplib.SMTP(mailserver) # connect, no login step
errors = server.sendmail(From, To, text)
server.quit()
if errors: # smtplib also raises excepts
 print 'Errors:', errors
else:
 print 'No errors.'
print 'Bye.'

```

## The email package

- Parses mail text fetched with poplib
- Generates mail text to send with smtplib

### # Composing a simple message

```
>>> from email.Message import Message
>>> m = Message()
>>> m['from'] = 'Sue Jones <sue@jones.com>'
>>> m['to'] = 'pp3e@earthlink.net'
>>> m.set_payload('The owls are not what they seem...')
>>> s = str(m)
>>> print s
From nobody Sun Jan 22 21:26:53 2006
from: Sue Jones <sue@jones.com>
to: pp3e@earthlink.net
```

The owls are not what they seem...

### # Parsing a simple message

```
>>> from email.Parser import Parser
>>> x = Parser().parsestr(s)
>>> x
<email.Message.Message instance at 0x00A7DA30>
>>> x['From']
'Sue Jones <sue@jones.com>'
>>> x.get_payload()
'The owls are not what they seem...'
>>> x.items()
[('from', 'Sue Jones <sue@jones.com>'), ('to', 'pp3e@earthlink.net')]
```

```
>>> for part in x.walk():
... print x.get_content_type()
... print x.get_payload()
...
text/plain
The owls are not what they seem...
```

### # Composing a multi-part message (attachments)

```
>>> from email.MIMEMultipart import MIMEMultipart
>>> from email.MIMEText import MIMEText
>>>
>>> top = MIMEMultipart()
>>> top['from'] = 'Art <arthur@camelot.org>'
>>> top['to'] = 'pp3e@earthlink.net'
>>>
>>> sub1 = MIMEText('nice red uniforms...\n')
>>> sub2 = MIMEText(open('data.txt').read())
>>> sub2.add_header('Content-Disposition', 'attachment',
... filename='data.txt')
>>> top.attach(sub1)
>>> top.attach(sub2)
```

```
>>> text = top.as_string() # same as str() or print
>>> print text
Content-Type: multipart/mixed; boundary="=====0257358049=="
MIME-Version: 1.0
from: Art <arthur@camelot.org>
to: pp3e@earthlink.net

-----0257358049==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

nice red uniforms...

-----0257358049==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename="data.txt"

line1
line2
line3

-----0257358049=---
```

#### # Parsing a multi-part message

```
>>> from email.Parser import Parser
>>> msg = Parser().parsestr(text)
>>> msg['from']
'Art <arthur@camelot.org>'

>>> for part in msg.walk():
... print part.get_content_type()
... print part.get_payload()
... print
...
multipart/mixed
[<email.Message.Message instance at 0x00A82058>,
<email.Message.Message instance at 0x00A82260>]

text/plain
nice red uniforms...

text/plain
line1
line2
line3
```

## Running the email examples

See also: `pymail.py` and `PyMailGui.py` in examples directory

## PyMailGui is a Python/Tk mail client (screen shots in GUI unit)

```

C:\examples\Part3\Internet\Email>python smtpmail.py
From? lutz@rmi.net
To? lutz@rmi.net
Subj? testing 1 2 3
Type message text, end with line=(ctrl + D or Z)
words
Connecting...
No errors.
Bye.

C:\examples\Part3\Internet\Email>python popmail.py
Password for pop.rmi.net?
Connecting...
+OK Cubic Circle's v1.31 1998/05/13 POP3 ready
There are 1 mail messages in 780 bytes

[Press Enter key]
Received: by chevalier (mbox lutz)
 (with Cubic Circle's cucipop (v1.31 1998/05/13) Sun Feb 13 13:56:44 2000)
X-From_: lutz@rmi.net Sun Feb 13 13:43:01 2000
Return-Path: <lutz@chevalier.rmi.net>
Received: from server.python.net (server.python.net [209.50.192.113])
 by chevalier.rmi.net (8.9.3/8.9.3) with SMTP id NAA16859
 for <lutz@rmi.net>; Sun, 13 Feb 2000 13:43:00 -0700 (MST)
Message-Id: <200002132043.NAA16859@chevalier.rmi.net>
Received: (qmail 31944 invoked from network); 13 Feb 2000 20:43:22 -0000
Received: from dial-67.73.denco.rmi.net (166.93.67.73)
 by server.python.net with SMTP; 13 Feb 2000 20:43:22 -0000
From: lutz@rmi.net
To: lutz@rmi.net
Date: Sun Feb 13 13:44:16 2000
Subject: testing 1 2 3

words

Bye.

```

## Other client-side tools

- ***urllib: fetching web pages***
- ***nntplib: reading and posting usenet news***
- ***httplib: lower-level web conversations***



- *telnet, gopher, imap,...*
- *htmllib, xml package: parsing fetched web pages and data (soap)*
- *Active Scripting: embedded Python in HTML*
- *Jython: client-side applets in Python*

## ***Building web sites with Python***

### ◆ **CGI scripting**

- *Simpler techniques for simpler sites*
- *Augment with state retention: hidden form fields, query parameters, cookies, server-side databases*
- *Embeds HTML in Python (opposite of PSP, etc.)*
- *Cgi module parses inputs, escapes outputs*
- *See also HTMLgen and similar for reply generation*
- *Require web server that can run Python scripts*

### ◆ **Web frameworks and tools**

- ◆ *Zope (OO meets websites: DTML, TAL, ZMI, ZODB, ...)*
- ◆ *Plone (CMS built on top of Zope, workflow)*
- ◆ *TurboGears (multi-tool package: AJAX, MVC, SQLAlchemy, CherryPy,...)*
- ◆ *Django (model view controller implicit)*
- ◆ *CherryPy, WebWare, Quixote (embarrassment of riches!)*
- ◆ *PSP (Webware and mod\_python, like PHP/ASP)*
- ◆ *mod\_python for Apache (speed, sessions, PSP,...)*
- ◆ *Fast CGI (persistent processes for state retention)*
- ◆ *Twisted (network server framework)*

## Writing server-side CGI scripts

- ◆ Server-side scripts referenced from HTML pages
- ◆ Run on server, connected to client/browser via sockets
- ◆ cgi module handles input parsing, output formatting

### Typical operation

- ◆ *Inputs*: fetch info typed into forms (cgi.FieldStorage)
  - ◆ *cgi scripts get input from stdin plus environment info*
- ◆ cgi library module
  - ◆ *provides dictionary interface to parsed form data*
- ◆ *Outputs*: results in browser (cgi.escape, urllib.quote)
  - ◆ *cgi scripts write HTML to stdout to display results*

### Basic CGI interfaces

- ◆ **FieldStorage class**
- ◆ *Reads the form contents from standard input or the environment*

```
print "Content-type: text/html" # HTML text follows
print # blank line: end headers
print "<TITLE>CGI script output</TITLE>"
```

```
form = cgi.FieldStorage()
form_ok = 0
if form.has_key("name") and form.has_key("addr"):
 if (form["name"].value != "" and
 form["addr"].value != ""):
```

```

 form_ok = 1

if not form_ok:
 print "<H1>Error</H1>"
 print "Please fill in the name and addr fields."
 return
...more form processing here...

```

### ◆ FormContent class

- ◆ *A (now) outdated interface: same functionality, but FieldStorage above is preferred scheme*

```

import cgi # see library manual
...
form = cgi.FormContent() # parse input stream
if form.has_key("fieldname"):
 data = form["fieldname"][0]

```

## Basic CGI example

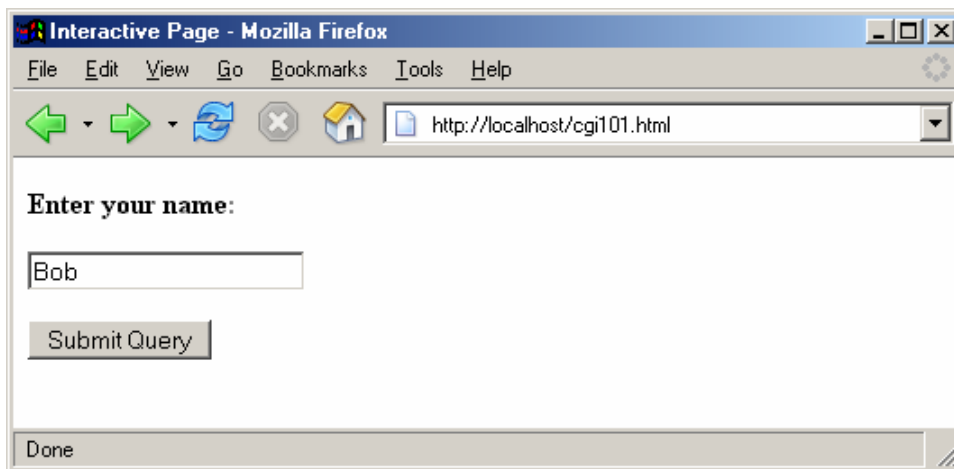
```

input form: cgi101.html

 • use GET instead of POST for some server/browser combos

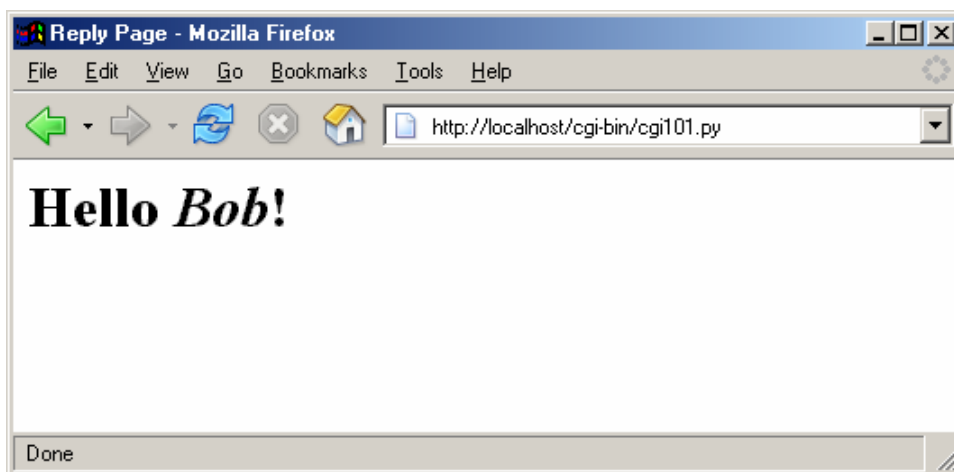
<html><body>
<title>Interactive Page</title>
<form method=GET action="cgi-bin/cgi101.py">
 <P>Enter your name:
 <P><input type=text name=user>
 <P><input type=submit>
</form>
</body></html>

```



```
reply script: cgi-bin/cgi101.py
```

```
#!/usr/bin/python
import cgi
form = cgi.FieldStorage() # parse form data
print "Content-type: text/html\n" # hdr plus blank line
print "<title>Reply Page</title>" # html reply page
if not form.has_key('user'):
 print "<h1>Who are you?</h1>"
else:
 print "<h1>Hello <i>%s</i>!</h1>" % cgi.escape(form['user'].value)
```



```
running a local webserver: Extras\Misc\webserver.py
```

```
#####
implement HTTP server in Python which
knows how to run server-side CGI scripts;
serves files/scripts from current working dir;
scripts must be in webdir\cgi-bin or htbin
#####

webdir = '.'
import os, sys
from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler

hack for Windows: os.environ not propagated
to subprocess by os.popen2, force in-process
if sys.platform[:3] == 'win':
 CGIHTTPRequestHandler.have_popen2 = False
 CGIHTTPRequestHandler.have_popen3 = False

os.chdir(webdir) # run in html root dir
srvraddr = ("", 80) # my hostname, portnumber
srvrobject = HTTPServer(svraddr, CGIHTTPRequestHandler)
srvrobject.serve_forever() # run as perpetual demon

testing with query strings and urllib

>>> from urllib import urlopen
>>> conn = urlopen('http://localhost/cgi-bin/cgi101.py?user=Sue+Smith')
>>> reply = conn.read()
>>> reply
'<title>Reply Page</title>\n<h1>Hello <i>Sue Smith</i>!</h1>\n'

>>> urlopen('http://localhost/cgi-bin/cgi101.py').read()
'<title>Reply Page</title>\n<h1>Who are you?</h1>\n'

>>> urlopen('http://localhost/cgi-bin/cgi101.py?user=Bob').read()
'<title>Reply Page</title>\n<h1>Hello <i>Bob</i>!</h1>\n'
```

## CGI controls: test5.html/.cgi (PP book)

Note: these examples live at <http://starship.python.net/~lutz/PyInternetDemos.html>. Visit that site to view the HTML that generates this and other pages--select 'view source' in your browser.

```
test5b.html
<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: alternative layout</H1>
<P>Use the same test5.cgi server side script, but change the
```

layout of the form itself. Notice the separation of user interface and processing logic here; the CGI script is independent of the HTML used to interact with the user/client.</P><HR>

```
<FORM method=POST
action="http://starship.python.net/~lutz/test5.cgi">
 <H3>Please complete the following form and click Submit</H3>
 <P><TABLE border cellpadding=3>
 <TR>
 <TH align=right>Name:
 <TD><input type=text name=name>
 <TR>
 <TH align=right>Shoe size:
 <TD><input type=radio name=shoesize value=small>Small
 <input type=radio name=shoesize value=medium>Medium
 <input type=radio name=shoesize value=large>Large
 <TR>
 <TH align=right>Occupation:
 <TD><select name=job>
 <option>Developer
 <option>Manager
 <option>Student
 <option>Evangelist
 <option>Other
 </select>
 <TR>
 <TH align=right>Political affiliations:
 <TD><P><input type=checkbox name=language value=Python>Pythonista
 <P><input type=checkbox name=language value=Perl>Perlmonger
 <P><input type=checkbox name=language value=Tcl>Tcler
 <TR>
 <TH align=right>Comments:
 <TD><textarea name=comment cols=30 rows=2>Enter spam here</textarea>
 <TR>
 <TD colspan=2 align=center>
 <input type=submit value="Submit">
 <input type=reset value="Reset">
 </TD>
 </TR>
 </TABLE>
</FORM>
</BODY></HTML>
```

**Please complete the following form and click Submit**

<b>Name:</b>	<input type="text" value="Guido"/>
<b>Shoe size:</b>	<input type="radio"/> Small <input type="radio"/> Medium <input checked="" type="radio"/> Large
<b>Occupation:</b>	<input type="text" value="Developer"/>
<b>Political affiliations:</b>	<input checked="" type="checkbox"/> Pythonista <input type="checkbox"/> Perlmonger <input type="checkbox"/> Tcler
<b>Comments:</b>	<input type="text" value="Python rules!"/>
<input type="button" value="Submit"/> <input type="button" value="Reset"/>	

Document: Done

```

test5.cgi
#!/usr/local/bin/python
runs on the server, reads form input, prints html;
executable privileges, stored in ~/public_html,

import cgi, sys, string
form = cgi.FieldStorage() # parse form data
print "Content-type: text/html" # plus blank line

html = """
<TITLE>test4.cgi</TITLE>
<H1>Greetings</H1>
<HR>
<H4>Your name is %(name)s</H4>
<H4>You wear rather %(shoesize)s shoes</H4>
<H4>Your current job: %(job)s</H4>
<H4>You program in %(language)s</H4>
<H4>You also said:</H4>
<P>%(comment)s</P>
<HR>"""

data = {}
for field in ['name', 'shoesize', 'job', 'language', 'comment']:
 if not form.has_key(field):

```

```

 data[field] = '(unknown)'
 else:
 if type(form[field]) != type([]):
 data[field] = form[field].value
 else:
 values = map(lambda x: x.value, form[field])
 data[field] = string.join(values, ' and ')
print html % data

```



In typical interactions, a user directs a browser to a HTML file, which includes a form action that automatically invokes the server-side CGI script on submit button press. It's also possible to invoke a CGI script directly, provided that input fields are given values within the referencing URL address itself:

### test5c.html

```

<HTML><BODY>
<TITLE>CGI 101</TITLE>
<H1>Common input devices: URL parameters</H1>

```

```
<P>
```

This demo invokes the test5.cgi server-side script again, but hardcodes input data to the end of the script's URL, within a simple hyperlink (instead of packaging up a form's inputs). Click your browser's "show page source" button to view the links associated with each list item below.



<P>This is really more about CGI than Python, but notice that Python's cgi module handles both this form of input (which is also produced by GET form actions), as well as POST-ed forms; they look the same to the Python CGI script. In other words, cgi module users are independent of the method used to submit data.

<P>Also notice that URLs with appended input values like this can be generated as part of the page output by another CGI script, to direct a next user click to the right place and context; together with type 'hidden' input fields, they provide one way to save state between clicks.

</P><HR>

<UL>

<LI><A href=

"http://starship.python.net/~lutz/test5.cgi?name=Bob&shoesize=small">

Send Bob, small</A>

<LI><A href=

"http://starship.python.net/~lutz/test5.cgi?name=Tom&language=Python">

Send Tom, Python</A>

<LI><A href=

"http://starship.python.net/~lutz/test5.cgi?job=Evangelist&comment=spam">

Send Evangelist, spam</A>

</UL>

</UL>

<HR>

</BODY></HTML>

## The Grail web browser

- ♦ *Portable: written in Python, uses Tkinter as browser GUI*
- ♦ *Browser downloads/runs applets referenced in HTML*
- ♦ *Applets more powerful than HTML+CGI: a full GUI API*
- ♦ *Python also supports ActiveX, Netscape plug-ins,...*

### HTML file

<HEAD>

<TITLE>Grail Applet Test Page</TITLE>

</HEAD>

<BODY>

<H1>Test an Applet Here!</H1>

Click this button!

<APP CLASS=Question>

</BODY>

### file: Question.py

```
Python applet file: Question.py
in the same location (URL) as the html file
that references it; adds widgets to browser;

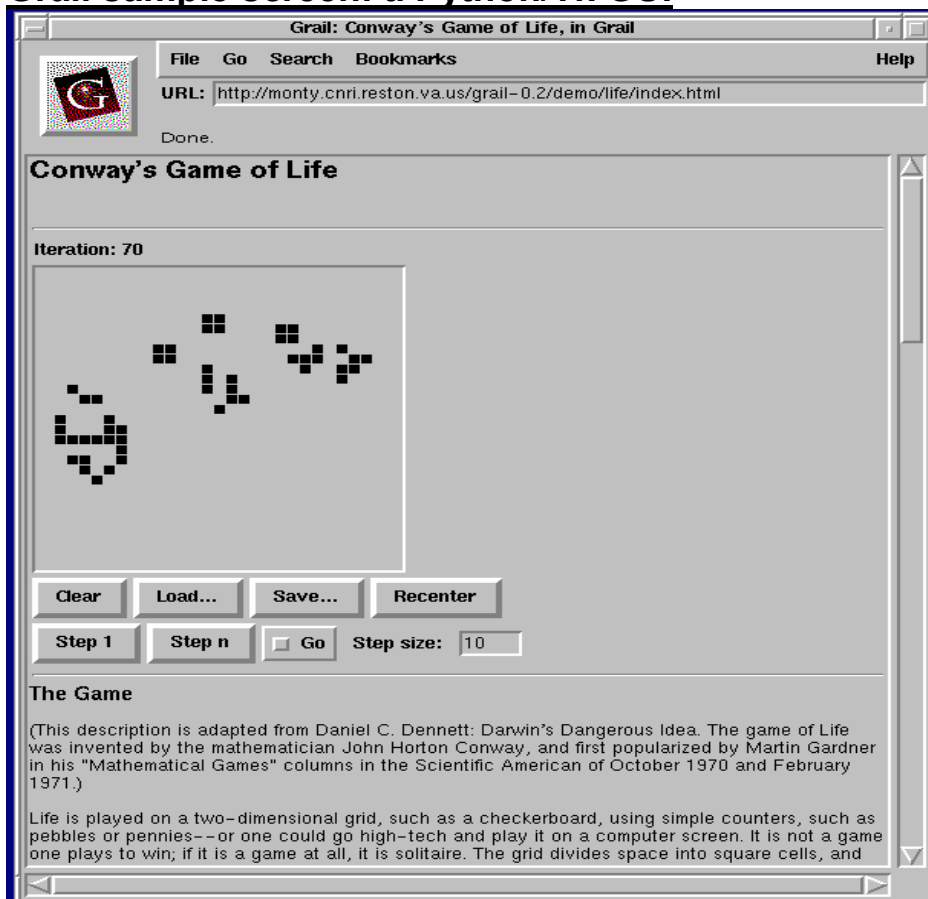
from Tkinter import *

class Question:
 # run by grail?
 def __init__(self, parent):
 # parent=browser
 self.button = Button(parent,
 bitmap='question',
 command=self.action)
 self.button.pack()

 def action(self):
 if self.button['bitmap'] == 'question':
 self.button.config(bitmap='questhead')
 else:
 self.button.config(bitmap='question')

if __name__ == '__main__':
 root = Tk()
 button = Question(root)
 root.mainloop()
 # run stand-alone?
 # parent=Tk: top-level
```

## Grail sample screen: a Python/Tk GUI



## ***Jython: Python for Java systems***

- ◆ An independent implementation of Python
- ◆ Python for Java-based applications
- ◆ Python *scripting* complements Java *systems* language
- ◆ Renamed from “JPython” to “Jython” (2000: copyright)

### **Structure**

- ***A collection of Java classes that compile and run Python code***
- ***“jython” program: equivalent to “python”, interactive, scripts***
- ***“jythonc” compiler/packager program: makes .jar, .class files***
- ***Embedding: “PythonInterpreter” class runs Python from Java***
- ***Extending: automatic, Python coded imports, uses Java classes***

### **Advantages**

- ◆ Compiles Python code to JVM byte-code
- ◆ ***“100% Pure Java” (written in Java, generates JVM)***
- ◆ ***More seamless than Tcl and Perl approaches***
- ◆ ***Can run as client-side applets in Java-aware browsers***
- ◆ Includes Python/Java integration support
- ◆ ***Python can access Java classes***
- ◆ ***Java can embed/call Python scripts***
- ◆ ***Access to AWT and Swing from Python for GUIs***

## Examples

### Hello World

```
from java.applet import Applet

class HelloWorld(Applet):
 def paint(self, gc):
 gc.drawString("Hello world", 20, 30)
```

### A simple calculator

```
from java import awt
from pawt import swing

labels = ['7', '8', '9', '+',
 '4', '5', '6', '-',
 '1', '2', '3', '*',
 '0', '.', '=', '/']

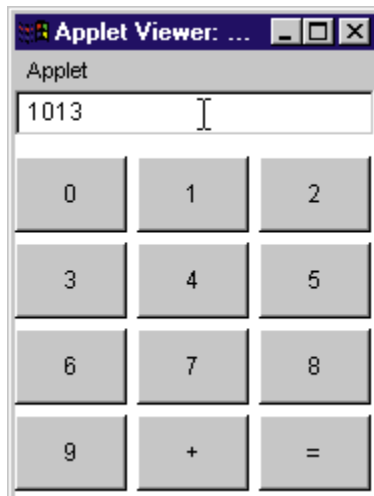
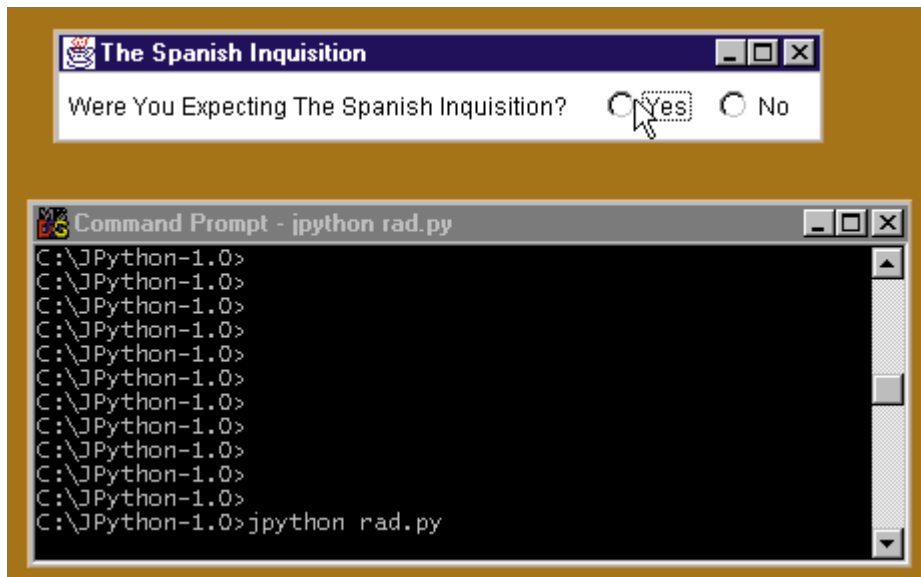
keys = swing.JPanel(awt.GridLayout(4, 4))
display = swing.JTextField()

def push(event):
 # Callback for regular keys
 display.replaceSelection(event.actionCommand)

def enter(event):
 # Callback for '=' key
 display.text = str(eval(display.text))
 display.selectAll()

for label in labels:
 key = swing.JButton(label)
 if label == '=':
 key.actionPerformed = enter
 else:
 key.actionPerformed = push
 keys.add(key)

panel = swing.JPanel(awt.BorderLayout())
panel.add("North", display)
panel.add("Center", keys)
swing.test(panel)
```



## Jython downsides

- ◆ Not yet fully compatible with standard Python

- ◆ Only applicable where a JVM is installed or shipped
- ◆ Requires users to learn Java too (environment, libs)
- ◆ Doesn't support standard C/C++ extension modules
- ◆ Slower than C Python (1.7x, 10x, 100X), for now?

## Active Scripting and COM

- Active Scripting: Python embedded in HTML, like JavaScript or VBScript
- Extracted and run on client (browser object module) or server (form and reply object models)
- Windows only, IE and IIS only today, requires Python + PyWin32 (formerly win32all) package installs
- More useful on server side: only one machine requires Python + Pywin32 installs, any browser will work
- COM and DCOM: Microsoft component object models; DCOM allows client and server to be remote
- Host uses Active Scripting to export COM object interfaces for use in Python code
- Future: Python port to C#/.Net framework; see also Python Soap, XML-RPC support for web services

### Active Scripting example

**➔ Embedded Python in HTML currently disabled on client due to rexec security issue, though Javascript can still call registered Python COM objects there; see also ASP, PSP, Zope,...**

```
<HTML>
<BODY>
<H1>Embedded code demo: Python</H1>
<SCRIPT Language=Python>

embedded python code shows three alert boxes
as page is loaded; any Python code works here,
and uses auto-imported global funcs and objects
```

```

def message(i):
 if i == 2:
 alert("Finished!")
 else:
 alert("A Python-generated alert => %d" % i)

for count in range(3): message(count)

</SCRIPT>
</BODY></HTML>

```

## A Python COM client

```

from sys import argv
docdir = 'C:\\temp\\'
if len(argv) == 2: docdir = argv[1] # ex: comclient.py a:\

from win32com.client import Dispatch # early or late binding
word = Dispatch('Word.Application') # connect/start word
word.Visible = 1 # else word runs hidden

create and save new doc file
newdoc = word.Documents.Add() # call word methods
spot = newdoc.Range(0,0)
spot.InsertBefore('Hello COM client world!') # insert some text
newdoc.SaveAs(docdir + 'pycom.doc') # save in doc file
newdoc.SaveAs(docdir + 'copy.doc')
newdoc.Close()

```

## A Python COM server

```

import sys
from win32com.server.exception import COMException # what to raise
import win32com.server.util # server tools
globhellos = 0

class MyServer:

 # com info settings
 _reg_clsid_ = '{1BA63CC0-7CF8-11D4-98D8-BB74DD3DDE3C}'
 _reg_desc_ = 'Example Python Server'
 _reg_progid_ = 'PythonServers.MyServer' # external name
 _reg_class_spec_ = 'comserver.MyServer' # internal name
 _public_methods_ = ['Hello', 'Square']
 _public_attrs_ = ['version']

 # python methods
 def __init__(self):
 self.version = 1.0
 self.hellos = 0
 def Square(self, arg): # exported methods
 return arg ** 2
 def Hello(self): # global variables
 # retain state, but
 # self vars don't
 global globhellos
 globhellos = globhellos + 1
 self.hellos = self.hellos + 1
 return 'Hello COM server world [%d, %d]' % (globhellos, self.hellos)

```

```

registration functions
def Register(pyclass=MyServer):
 from win32com.server.register import UseCommandLine
 UseCommandLine(pyclass)
def Unregister(classid=MyServer._reg_clsid_):
 from win32com.server.register import UnregisterServer
 UnregisterServer(classid)

if __name__ == '__main__':
 Register() # register server if file run or clicked
 # unregisters if --unregister cmd-line arg

```

## Other Internet-related tools

See [Extras\Internet](#) directory on the class CD for more examples

- ◆ XML: SAX/DOM parsers, XML-RPC, SOAP
  - ◆ See [Extras\XML](#) DOM/SAX examples on class CD
  - ◆ 3<sup>rd</sup>-party extensions for XPath,... (see [xml-sig](#) page)
  - ◆ O'Reilly book: *Python & XML*
  - ◆ *xmlrpclib* in Python std lib
  - ◆ *PySoap*, *Soapy*: SOAP protocol
  - ◆ *ElementTree*: XML parser/generator in Python 2.5 std lib
- ◆ HTMLgen
  - ◆ generates HTML from class-based page descriptions ([see CD](#))
- ◆ HTML, SGML parsers
  - ◆ useful for extracting information from web pages
- ◆ rfc822 module, email package
  - ◆ parses standard message headers
- ◆ ILU, fnorb, and OmniORB packages
  - ◆ can be used for CORBA networking



- ◆ **urllib module**
  - ◆ *opens URL, returns file-like object: read, readline ([see CD](#))*
  - ◆ *Parse fetched page with `htmllib`, `string.find`, `xml` package*
- ◆ **Restricted execution mode (security issue)**
  - ◆ *See [Extras\Internet](#) directory on CD*
  - ◆ *for running code fetched from untrusted sources*
  - ◆ *modules “rexec” and “bastion”*
  - ◆ *➔ Withdrawn in Python 2.4: security concerns*
- ◆ **Other Internet protocol support**
  - ◆ *web server classes (see [Extras\Internet](#) on CD)*
  - ◆ *`telnetlib`, `nntplib` (news), SSL sockets,...*
- ◆ **Zope & Plone ([www.zope.org](http://www.zope.org))**
  - ◆ *An open-source web application framework*
  - ◆ *Written in and customized with Python*
  - ◆ *Plone runs on Zope: workflow-based content management*
  - ◆ *“`http://server/module/func?arg1=val1&arg2=val2`” URL becomes  
“`module.func(arg1=val1, arg2=val2)`” call on server-side Python*
  - ◆ *Supplemental Zope/Plone CD available*

## Lab Session 10

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)


## 15. Extending Python in C/C++

- ◆ C/C++ registers functions to Python
- ◆ For optimization and integration

### Extending topics

- ◆ Integration overview
- ◆ C extension modules
- ◆ C extension types
- ◆ Wrapper classes
- ◆ SWIG glue code generator

## *Review: Python tool-set layers*

- ◆ Built-ins
  - ◆ *Lists, dictionaries, strings, library modules, etc.*
  - ◆ *High-level tools for simple, fast programming*
- ◆ Python extensions
  - ◆ *Functions, classes, modules*
  - ◆ *For adding extra features, and new object types*
- ◆ C extensions and embedding 
  - ◆ *C modules, C types, runtime API*
  - ◆ *For integrating external systems, optimizing components, customization*

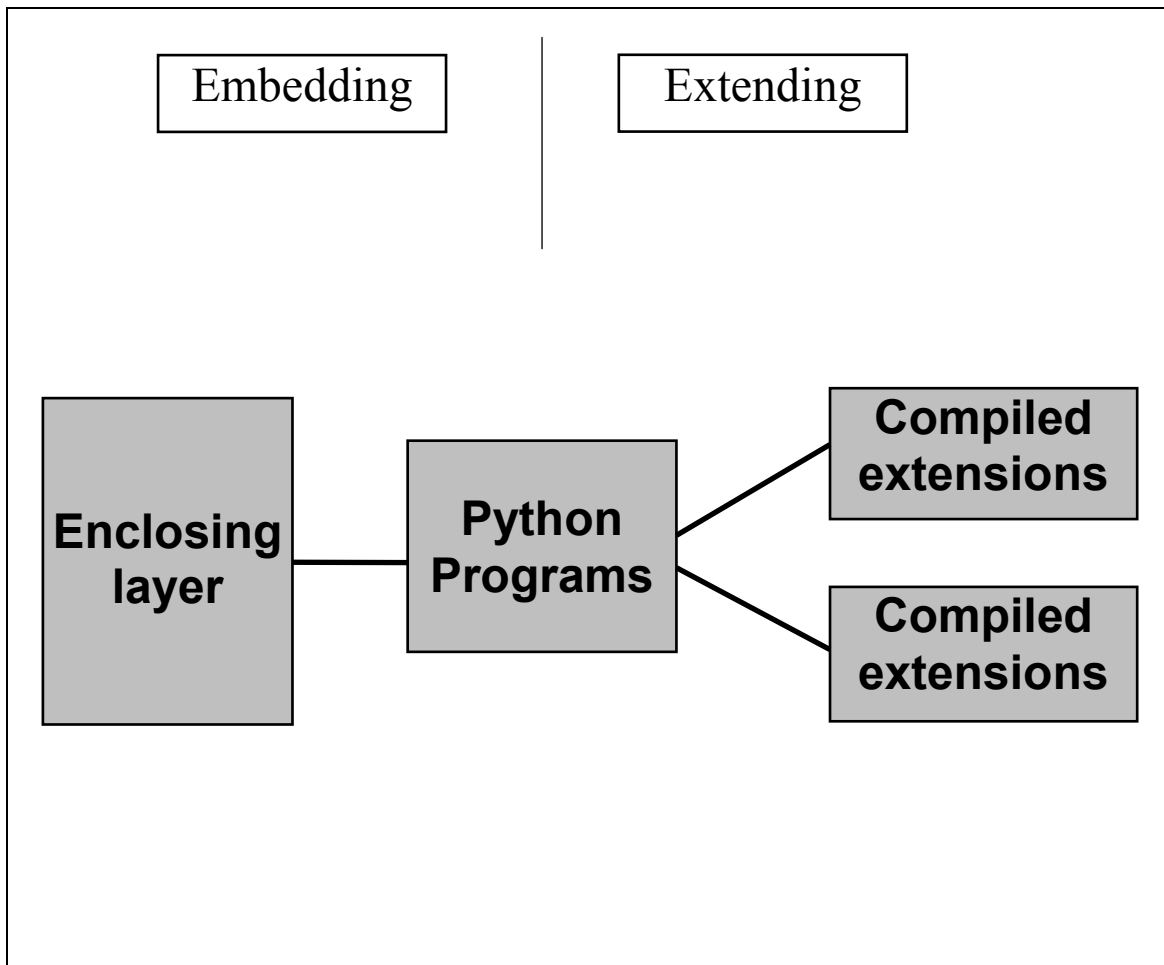
## ***Why integration?***

- ◆ **Optimization**
  - ◆ *time-critical components*
- ◆ **Customization**
  - ◆ *on-site/end-user changes*
- ◆ **Component reuse**
  - ◆ *wrapping C libraries in a Python front-end*

### **“Hybrid development”**

- ◆ *Python is optimized for speed-of-development*
- ◆ *Python is designed for multi-language systems*
- ◆ *Leverages both development and execution speed*

## *The 'big picture' revisited*



## Integration modes

### ◆ Extending

- ◆ *Python  $\Rightarrow$  C/C++*
- ◆ *For optimization, library integration, special tasks*
- ◆ *Via registering C function pointers*

### ◆ Embedding

- ◆ *C/C++  $\Rightarrow$  Python*
- ◆ *For system customizations, callbacks*
- ◆ *Via calling Python API functions*

### ◆ May be mixed arbitrarily

- ◆ *Python  $\Rightarrow$  C/C++  $\Rightarrow$  Python  $\Rightarrow$  C/C++ . . .*

## Extending basics

- Extending requires glue code between Python and C library for data translation
- Glue code can be written in C, or generated in C with tools like SWIG
- Glue code can also be written in Python with ctypes, std lib module in Python 2.5+

## Extending Tools

- ◆ SWIG: generate glue code from C/C++ declarations
- ◆ SIP: a smaller SWIG
- ◆ Ctypes: write glue code in Python (std lib in 2.5)
- ◆ Boost.Python: write glue with C++ template code
- ◆ PyRex: a Python/C combo for new extensions
- ◆ CXX: Python API for C++
- ◆ F2py, pyfort: glue code for Fortran

## A simple C extension module

- ◆ C extension module for Python, called “environ”
- ◆ Wraps (exports) the C library’s ‘getenv’ function

### file: environ.c

```
#include <Python.h>
#include <stdlib.h>

/* Functions */
static PyObject *
wrap_getenv(PyObject *self, PyObject *args)
{
 /* args from python */
 char *varName, *varValue;
 PyObject *returnObj = NULL; /* null=exception */

 if (PyArg_Parse(args, "s", &varName)) { /*P->C*/
 varValue = getenv(varName);
 if (varValue != NULL)
 returnObj = Py_BuildValue("s", varValue); /*C->P*/
 else
 PyErr_SetString(PyExc_SystemError, "bad getenv");
 }
 return returnObj;
}

/* Registration */
static struct PyMethodDef environ_methods[] = {
 {"getenv", wrap_getenv}, /* name, address */
 {NULL, NULL}
};

/* Initialization */
void initempty()
{
 /* called by Python */
 /* on first import */
 (void) Py_InitModule("environ", environ_methods);
}
```

## Using the C module in Python

- ◆ Used just like Python modules
- ◆ Serve same roles: name-space

- ◆ C functions handle all data conversions
- ◆ C raises exceptions by returning 'NULL'

```
% python
>>> import environ
>>> environ.getenv("USER")
'mlutz'

>>> environ.getenv("PYTHONPATH")
'./opt/local/src/Python-1.4/Lib'

>>> dir(environ)
['__doc__', '__file__', '__name__', 'getenv']

>>> environ.__file__
'./environ.so'
```

## C module structure

- ◆ API tools
  - ◆ *Python API symbols start with a 'Py' prefix*
  - ◆ *'PyArg\_Parse' converts arguments to C*
  - ◆ *'Py\_BuildValue' C data to Python return object*
  - ◆ *'PyObject\*' for generic Python objects in C*
- ◆ Module method functions
  - ◆ *'self' argument (not used for modules)*
  - ◆ *'args'—Python tuple containing passed arguments*
- ◆ Name-to-address registration table
  - ◆ *Maps name strings to C function address*
  - ◆ *Null table terminator*

- ◆ **Module initialization function**
  - ◆ *Called by Python when the module is first imported*
  - ◆ *'Py\_InitModule' initializes module attributes dictionary*
  - ◆ *'initenvron' called by name (non-static)*

## ***Binding C extensions to Python***

- ◆ **Static binding...**                      **rebuild Python**
- ◆ **Dynamic binding...**                **load when imported**

### **Static binding**

#### **1. Add file to Python tree**

*Put (or link to) the source or object file in Modules directory of Python source tree*

#### **2. Add line to Modules/Setup**

*Add a line to the Modules/Setup file in the Python source tree: "environ environ.c" (config table)*

#### **3. Rebuild Python**

*Rerun the 'make' command at the top-level of Python's source tree directory*



## Dynamic Binding

- ◆ Loaded into process on first import
- ◆ Doesn't require access to Python source
- ◆ Doesn't require rebuilding Python
- ◆ Makes for smaller executable/process
- ◆ Compile/link details are platform-specific

### 1. Compile into shareable

*Compile C module into a sharable (dynamic-load) object file: .so, .sl, .dll, etc. (see makefiles below and in embedding unit)*

### 2. Add to module search path

*Put the shareable object file in a directory named on the '\$PYTHONPATH' environment variable*

## Makefile example, dynamic binding on Linux

### File makefile.environ

```
#####
Compile environ.c into a shareable object file on
Linux, to be loaded dynamically when first imported,
whether from interactive, stand-alone, or embedded code.
To use, type make -f makefile.environ, at your shell;
to run, make sure environ.so is on a dir on PYTHONPATH.
#
To link statically with Python instead, add line like:
environ ~/examples/Part1/Preview/Integrate/environ.c
to Modules/Setup (or add a link to environ.c in Modules,
and add a line like environ environ.c) and re-make
python itself; this works on any platform, and no extra
makefile like this one is needed;
#
To make a shareable on Solaris, you might instead say:
cc xxx.c -c -KPIC -o xxx.o
ld -G xxx.o -o xxx.so
rm xxx.o
On other platforms, it's more different still; see
your c or c++ compiler's documentation for details
#####

PY = /home/mark/python1.5.2-ddjcd/Python-1.5.2
```

```

environ.so: environ.c
 gcc environ.c -g -I$(PY)/Include -I$(PY) -fpic -shared -o environ.so

clean:
 rm -f environ.so

#or-- environ.so: environ.c
gcc environ.c -c -g -fpic -I$(PY)/Include -I$(PY) -o environ.o
gcc -shared environ.o -o environ.so
rm -f environ.o

```

## Data conversions: Python $\Leftrightarrow$ C

- ◆ ‘PyArg\_Parse’ converts arguments to C
- ◆ ‘Py\_BuildValue’ C data to Python return object
- ◆ ‘PyArg\_ParseTuple’ assumes it’s converting a tuple
- ◆ Other API tools handle type-specific conversions

### Common conversion codes

Format code	C data-type	Python object-type
“s”	char*	string
“s#”	char*, int	string, length
“i”	int	integer
“l”	long int	integer
“c”	char (or int for build)	string
“f”	float	floating-point
“d”	double	floating-point
“O”	PyObject*	a raw object
“(items)”	targets or values	nested tuple
“[items]”	series of arg/value	list
“{items}”	series of “key,value”	dictionary

- ◆ **Warning:**
- ◆ *“s” returns pointer to string in Python: use it or lose it*

## ***C extension types***

- ◆ **Creation of multiple instances**  
*Each is a new C ‘struct’, not a dictionary*
- ◆ **Overloading operators and type operations**  
*Via type-descriptor tables, not method names*
- ◆ **Types also now support inheritance like classes**

### **Type file components**

- 1. A C ‘struct’ used to hold per-instance data**
- 2. Instance method functions and registration table**
- 3. Functions to handle general type operations**
- 4. Functions to handle specific type category operations**
- 5. Type-descriptor tables: register operation handlers**
- 6. A C extension module: exports instance constructor**

## A 'skeleton' C extension type

- ◆ A C string-stack type
- ◆ Implements push/pop methods
- ◆ Overloads sequence and type operators

### file: stacktyp.c

```

/*****
 * stacktyp.c: a character-string stack data-type;
 * a C extension type, for use in Python programs;
 * stacktype module clients can make multiple stacks;
 * similar to stackmod, but 'self' is the instance,
 * and we can overload sequence operators here;
 *****/

#include "Python.h"

static PyObject *ErrorObject; /* local exception */
#define onError(message) \
 { PyErr_SetString(ErrorObject, message); return NULL; }

/*****
 * STACK-TYPE INFORMATION
 *****/

#define MAXCHARS 2048
#define MAXSTACK MAXCHARS

typedef struct { /* stack instance object */
 PyObject_HEAD /* python header */
 int top, len; /* + per-instance info */
 char *stack[MAXSTACK];
 char strings[MAXCHARS];
} stackobject;

staticforward PyTypeObject Stacktype; /* type descriptor */

#define is_stackobject(v) ((v)->ob_type == &Stacktype)

/*****
 * INSTANCE METHODS
 *****/

static PyObject *
stack_push(self, args) /* on "instance.push(arg)" */
 stackobject *self; /* 'self' is the instance */
 PyObject *args; /* 'args' passed to method */
{
 char *pstr;
 if (!PyArg_ParseTuple(args, "s", &pstr))

```

```

 return NULL;
 [. . .]
 Py_INCREF(Py_None);
 return Py_None;
}

static PyObject *
stack_pop(self, args)
 stackobject *self;
 PyObject *args; /* on "instance.pop()" */
{
 [. . .]
 return Py_BuildValue("s", "not implemented");
}

static PyObject *
stack_top(self, args)
 stackobject *self;
 PyObject *args;
{
 [. . .]
}

static PyObject *
stack_empty(self, args)
 stackobject *self;
 PyObject *args;
{
 [. . .]
}

/* instance methods */
static struct PyMethodDef stack_methods[] = {
 {"push", stack_push, 1}, /* name, addr */
 {"pop", stack_pop, 1},
 {"top", stack_top, 1},
 {"empty", stack_empty, 1},
 {NULL, NULL}
};

/*****
* BASIC TYPE-OPERATIONS
*****/

static stackobject * /* on "x = stacktype.Stack()" */
newstackobject() /* instance constructor */
{
 stackobject *self;
 self = PyObject_NEW(stackobject, &Stacktype);
 if (self == NULL)
 return NULL; /* raise exception */
 self->top = 0;
 self->len = 0;
 return self; /* a new type-instance */
}

static void /* instance destructor */
stack_dealloc(self) /* frees instance struct */
 stackobject *self;
{
 PyMem_DEL(self);
}

static int

```

```

stack_print(self, fp, flags)
 stackobject *self;
 FILE *fp;
 int flags; /* print self to file */
{
 [. . .]
}

static PyObject *
stack_getattr(self, name) /* on "instance.attr" */
 stackobject *self; /* bound-method or member */
 char *name;
{
 if (strcmp(name, "len") == 0)
 return Py_BuildValue("i", self->len);
 return
 Py_FindMethod(stack_methods, (PyObject *)self, name);
}

static int
stack_compare(v, w)
 stackobject *v, *w; /* return -1, 0 or 1 */
{
 [. . .]
}

/*****
* SEQUENCE TYPE-OPERATIONS
*****/

static int
stack_length(self)
 stackobject *self; /* on "len(instance)" */
{
 [. . .]
}

static PyObject *
stack_concat(self, other)
 stackobject *self; /* on "instance + other" */
 PyObject *other;
{
 [. . .] /* return new stack instance */
}

static PyObject *
stack_repeat(self, n) /* on "instance * N" */
 stackobject *self;
 int n;
{
 [. . .]
}

static PyObject *
stack_item(self, index) /* on x[i], for, in */
 stackobject *self;
 int index;
{
 if (index < 0 || index >= self->top) {
 PyErr_SetString(PyExc_IndexError, "out-of-bounds");
 return NULL;
 }
 else
 return Py_BuildValue("s", self->stack[index]);
}

```

```

static PyObject *
stack_slice(self, ilow, ihigh)
 stackobject *self;
 int ilow, ihigh;
{
 /* return ilow..ihigh slice of self--new object */
 onError("slicing not yet implemented")
}

/*****
 * TYPE DESCRIPTORS: MORE REGISTRATION
 *****/

static PySequenceMethods stack_as_sequence = {
 (inquiry) stack_length,
 (binaryfunc) stack_concat,
 (intargfunc) stack_repeat,
 (intargfunc) stack_item,
 (intintargfunc) stack_slice,
 (intobjargproc) 0, /* setitem */
 (intintobjargproc) 0, /* setslice */
};

static PyTypeObject Stacktype = { /* type descriptor */
 /* type header */
 PyObject_HEAD_INIT(&PyType_Type)
 0, /* ob_size */
 "stack", /* name */
 sizeof(stackobject), /* basicsize */
 0, /* itemsize */

 /* standard methods */
 (destructor) stack_dealloc, /* dealloc */
 (printfunc) stack_print, /* print */
 (getattrfunc) stack_getattr, /* getattr */
 (setattrfunc) 0, /* setattr */
 (cmpfunc) stack_compare, /* compare */
 (reprfunc) 0, /* repr */

 /* type categories */
 0, /* number ops */
 &stack_as_sequence, /* sequence ops */
 0, /* mapping ops */

 /* more methods */
 (hashfunc) 0, /* "dict[x]" */
 (binaryfunc) 0, /* "x()" */
 (reprfunc) 0, /* "str(x)" */
}; /* plus others: see Include/object.h */

```

```

/*****
* MODULE LOGIC: CONSTRUCTOR FUNCTION
*****/

static PyObject *
stacktype_new(self, args) /* on "x = stacktype.Stack()" */
 PyObject *self;
 PyObject *args;
{
 if (!PyArg_ParseTuple(args, "")) /* Module func */
 return NULL;
 return (PyObject *)newstackobject();
}

static struct PyMethodDef stacktype_methods[] = {
 {"Stack", stacktype_new, 1},
 {NULL, NULL}
};

void
initstacktype() /* on first "import stacktype" */
{
 PyObject *m, *d;
 m = Py_InitModule("stacktype", stacktype_methods);
 d = PyModule_GetDict(m);
 ErrorObject = Py_BuildValue("s", "stacktype.error");
 PyDict_SetItemString(d, "error", ErrorObject);
 if (PyErr_Occurred())
 Py_FatalError("can't initialize module stacktype");
}

```

## Using C extension types in Python

### Basic usage

```

% python
>>> import stacktype # load the type's module
>>> x = stacktype.Stack() # make a type-instance
>>> x.push('new') # call type-instance methods
>>> x # call the print handler

```

### Sequence operators

```

>>> x[0] # stack_item
'new'
>>> x[1] # raise IndexError
Traceback (innermost last):
 File "<stdin>", line 1, in ?

```



IndexError: out-of-bounds

```
>>> x[0:1] # stack_slice
>>> y = stacktype.Stack() # stacktype_new
>>> for c in 'SPAM': y.push(c) # stack_getattr ->
... # stack_push
>>> z = x + y # stack_concat
>>> z * 4 # stack_repeat
```

## Comparisons, exceptions

```
>>> t = stacktype.Stack()
>>> t == y, t is y, t > y, t >= y # stack_compare
>>> for i in range(1000): y.push('hello' + `i`)
...
Traceback (innermost last):
 File "<stdin>", line 1, in ?
stacktype.error: string-space overflow
```

## Wrapping C extensions in Python

- ◆ In older Python releases, C types don't support inheritance
- ◆ Wrapper classes add inheritance
- ◆ Wrappers can be specialized in Python

### file: oopstack.py

```
import stacktype # get C type
class Stack:
 def __init__(self, start=None): # wrap | make
 self._base = start or stacktype.Stack()
 def __getattr__(self, name):
 return getattr(self._base, name) # attributes
 def __cmp__(self, other):
 return cmp(self._base, other)
 def __repr__(self): # 'print'
 print self._base,; return ''
 def __add__(self, other): # operators
 return Stack(self._base + other._base)
 def __mul__(self, n):
 return Stack(self._base * n) # new Stack
 def __getitem__(self, i):
 return self._base[i] # [i],in,for
 def __len__(self):
 return len(self._base)
```

**file: substack.py**

```

from oopstack import Stack # get wrapper class

class Substack(Stack):
 def __init__(self, start=[]): # extend it in Python
 Stack.__init__(self)
 [. . .]

```

## Writing extensions in C++

- ◆ *Python is coded in portable ANSI C*
- ◆ *Normal C  $\Rightarrow$  C++ mixing rules apply*
- ◆ **Python header files**
  - ◆ *Automatically wrapped in extern "C", and may be included in C++ extensions freely*
- ◆ **Exported functions**
  - ◆ *Wrap functions to be called by Python in extern "C" declarations*
  - ◆ *Includes module initialization functions and (usually) method functions*
- ◆ **Static initializers**
  - ◆ *C++ global or static object constructors (initializers) may not work correctly, if main program is linked by C*

**See Also:**

- ◆ Wrapper class techniques: stubs for C++ class types
- ◆ C++ class  $\Leftrightarrow$  Python type integration work underway
- ◆ SWIG code generator: <http://www.swig.org/>

## SWIG example (PP book)

SWIG is designed to export ('wrap') existing C/C++ components to Python programs. It generates complete extension modules with type conversion code based on C/C++ type signatures. It can also export C global variables, and do Python shadow class generation for C++ classes.

*Also see "Extras" directory on the class CD for more examples*

## Module definition file

```
/* File : hellolib.i */

/*****
 * Swig module description file, for a C lib file.
 * Generate by saying "swig -python hellolib.i".
 * - %module sets name as known to Python importers
 * - %{...%} encloses code added to wrapper verbatim
 * - extern stmts declare exports in ANSI C syntax
 * You could parse the whole header file by using a
 * %include directive instead of the extern here,
 * but externs let you select what is wrapped/exported;
 * use '-Idir' swig args to specify .h search paths;
 *****/

%module hellowrap

%{
#include <hellolib.h>
%}

extern char *message(char*); /* or: %include "../HelloLib/hellolib.h" */
 /* or: %include hellolib.h, and use -I arg */
```

## Makefile, dynamic binding

```
#####
Use SWIG to integrate the hellolib.c examples for use
in Python programs. Using type signature information
in .h files (or separate .i input files), SWIG generates
the sort of logic we manually coded in the earlier example's
hellolib_wrapper.c. SWIG creates hellolib_wrap.c when run;
this makefile creates a hellowrap.so extension module file.
#
To build, we run SWIG on hellolib.i, then compile and
```

```

link with its output file. Note: you may need to first
get and build the 'swig' executable if it's not already
present on your machine: unpack, and run a './configure'
and 'make', just like building Python from its source.
You may also need to modify and source ./setup-swig.csh if
you didn't 'make install' to put swig in standard places.
See HelloLib/ makefiles for more details; the hellolib
.c and .h files live in that dir, not here.
#####

unless you've run make install
SWIG = ./myswig

PY = /home/mark/python1.5.2-ddjcd/Python-1.5.2
LIB = ../HelloLib

hellowrap.so: hellolib_wrap.o $(LIB)/hellolib.o
 ld -shared hellolib_wrap.o $(LIB)/hellolib.o -o hellowrap.so

generated wrapper module code
hellolib_wrap.o: hellolib_wrap.c $(LIB)/hellolib.h
 gcc hellolib_wrap.c -c -g -I$(LIB) -I$(PY)/Include -I$(PY)

hellolib_wrap.c: hellolib.i
 $(SWIG) -python -I$(LIB) hellolib.i

C library code in another directory
$(LIB)/hellolib.o:
 cd $(LIB); make -f makefile.hellolib-o hellolib.o

clean:
 rm -f *.o *.so core

force:
 rm -f *.o *.so core hellolib_wrap.c hellolib_wrap.doc

```

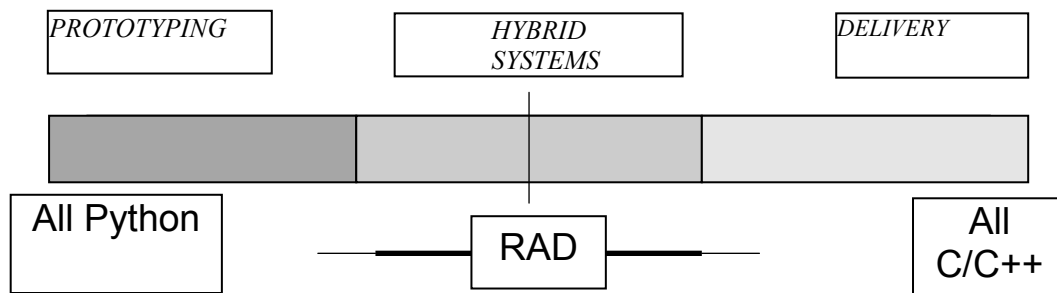
## Python and rapid development

- ◆ **Prototype-and-migrate**
  - ◆ *Code in Python initially*
  - ◆ *Move selected Python modules to C/C++*
  - ◆ *Use profiler to pick components to migrate*
- ◆ **Seamless migration path**
  - ◆ *C modules look just like Python modules*
  - ◆ *C types look almost like Python classes*
  - ◆ *Wrappers add inheritance to types*

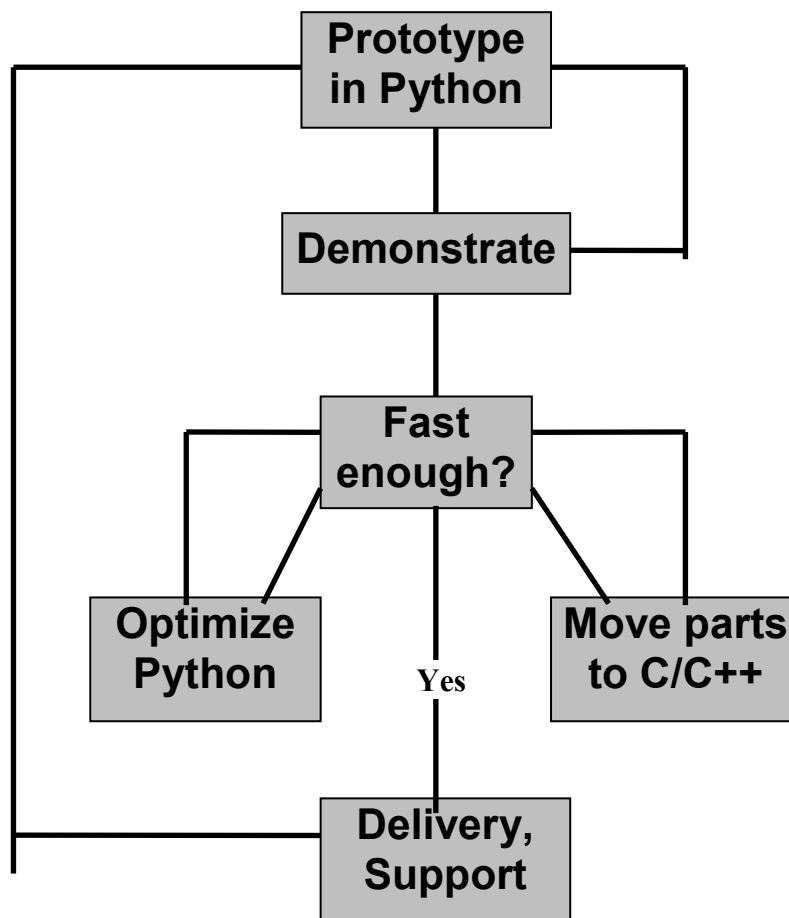
◆ Hybrid designs

- ◆ *Python front-end, C/C++ back-end*
- ◆ *C/C++ application, Python customizations*

The RAD slider



## Prototyping with Python



## ***Lab Session 11***

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 16. Embedding Python in C/C++

- ♦ C/C++ runs Python code via API calls
- ♦ For dynamic system customization

### Embedding topics

- ♦ Calling objects
- ♦ Running code strings
- ♦ Registration techniques
- ♦ Other topics: errors, tools, etc.

## *General embedding concepts*

### Embedded code forms

- ♦ Code strings
  - ♦ *Running expressions, statements*
- ♦ Callable objects
  - ♦ *Calling functions, classes, methods*
- ♦ Code files
  - ♦ *Importing modules, executing scripts*

## Embedded code sources

<i>Source</i>	<i>Description</i>
<b>Modules</b>	fetching code by importing modules
<b>Text files</b>	fetching code from simple text files
<b>Registration</b>	letting Python pass code to a C extension
<b>HTML tags</b>	extracting code from web pages
<b>Databases</b>	fetching code from a database table
<b>Processes</b>	receiving code over sockets
<b>Construction</b>	building Python code in C at runtime
<b>And so on</b>	system registries, etc.

## Common code sources

- ◆ Callable objects... modules, registration
- ◆ Code strings... files, registration, HTML,...
- ◆ Code files... files, modules, scripts

## Data communication techniques

<i>Code form</i>	<i>Technique</i>	<i>Mode</i>
Objects	function arguments	In/out
Objects	function return values	Out
Strings	expression results	Out
Strings, Objects	global module-level variables	In/out



Strings, Objects	C extension get/set functions	In/out
Strings, Objects	files, stdin/stdout, sockets,...	In/out

## Running simple code strings

- ◆ 'Py\_Initialize' initializes Python libraries
- ◆ 'PyRun\_SimpleString' runs Python statements
- ◆ Runs in module '\_\_main\_\_', no result from code
- ◆ Python code uses 'environ' *extension* module too

### file: main1.c

```
#include <Python.h>

main(argc, argv)
int argc;
char **argv;
{
 /* This is the simplest embedding mode. */
 /* Other API functions return results, */
 /* accept namespace arguments, allow */
 /* access to real Python objects, etc. */
 /* Strings may be precompiled for speed. */

 Py_Initialize(); /* init python */
 PyRun_SimpleString("print 'Hello embedded world!'");

 /* use C extension module above */
 PyRun_SimpleString("from environ import *");
 PyRun_SimpleString(
 "for i in range(4):\n"
 "\tprint i,\n"
 "\tprint 'Hello, %s' % getenv('USER')\n\n");

 PyRun_SimpleString("print 'Bye embedded world!'");
}
```

## Running the C program

```
% main1
Hello embedded world!
0 Hello, mlutz
1 Hello, mlutz
2 Hello, mlutz
3 Hello, mlutz
Bye embedded world!
```

Same as typing at “>>>” prompt:

```
print 'Hello embedded world!'
from environ import *
for i in range(4):
 print i,
 print 'Hello, %s' % getenv('USER')
print 'Bye embedded world!'
```

## **Building programs that embed Python**

- ◆ Link with Python library (1), and your main()
- ◆ Add any libs referenced by Python lib (Modules/Setup)
- ◆ Older scheme (1.4): 4 Python .a libs + 2 Python .o files

### **file: Makefile.main1**

```
this file builds a C executable that embeds Python
assuming no external module libs must be linked in
works on Linux with a custom Python build tree

PY = /home/mark/python1.5.2-ddjcd/Python-1.5.2
PYLIB = $(PY)/libpython1.5.a
PYINC = -I$(PY)/Include -I$(PY)

basic1: basic1.o
 cc basic1.o $(PYLIB) -g -export-dynamic -lm -ldl -o basic1

basic1.o: basic1.c
 cc basic1.c -c -g $(PYINC)
```

## Calling objects and methods

### The example task

- ◆ Make an instance of a Python class in a module file
- ◆ Call a method of that instance by name
- ◆ Python equivalent:

```
import <module>
object = <module>.<class>()
result = object.<method>(..args..)
```

### Python API tools to be used

<i>Tool</i>	<i>Description</i>
PyObject*	The type of a generic Python object in C
PyImport_ImportModule	Imports a Python module, much like the Python 'import'
PyObject_GetAttrString	Performs attribute qualifications: 'object.name', like 'getattr'
PyEval_CallObject	Calls any callable object (class, function, method), like 'apply'
Py_BuildValue	Converts C data to Python form, based on a format string
PyArg_Parse	Converts Python data to C form, based on a format string
Py_DECREF	Release ownership of object passed to C from the API

## The task implementation

file: module.py (on \$PYTHONPATH)

```
class klass:
 def method(self, x, y):
 return "brave %s %s" % (x, y) # run me from C
```

file: objects1.c

```
#include <Python.h>
#include <stdio.h>

main() {
 char *arg1="sir", *arg2="robin", *cstr;
 PyObject *pmod, *pclass, *pargs, *pinst, *pmeth, *pres;

 /* instance = module.klass() */
 Py_Initialize();
 pmod = PyImport_ImportModule("module");
 pclass = PyObject_GetAttrString(pmod, "klass");
 Py_DECREF(pmod);

 pargs = Py_BuildValue("()");
 pinst = PyEval_CallObject(pclass, pargs);
 Py_DECREF(pclass);
 Py_DECREF(pargs);

 /* result = instance.method(x,y) */
 pmeth = PyObject_GetAttrString(pinst, "method");
 Py_DECREF(pinst);
 pargs = Py_BuildValue("(ss)", arg1, arg2);
 pres = PyEval_CallObject(pmeth, pargs);
 Py_DECREF(pmeth);
 Py_DECREF(pargs);

 PyArg_Parse(pres, "s", &cstr); /* convert to C */
 printf("%s\n", cstr);
 Py_DECREF(pres);
}

% objects1
brave sir robin
```

## With full error checking (!)

file: objects1err.c

```
#include <Python.h>
#include <stdio.h>
#define error(msg) do { printf("%s\n", msg); exit(1); } while (1)

main() {
 char *arg1="sir", *arg2="robin", *cstr;
 PyObject *pmod, *pclass, *pargs, *pinst, *pmeth, *pres;

 /* instance = module.klass() */
 Py_Initialize();
 pmod = PyImport_ImportModule("module");
 if (pmod == NULL)
 error("Can't load module");
 pclass = PyObject_GetAttrString(pmod, "klass");
 Py_DECREF(pmod);
 if (pclass == NULL)
 error("Can't get module.klass");

 pargs = Py_BuildValue("()");
 if (pargs == NULL) {
 Py_DECREF(pclass);
 error("Can't build arguments list");
 }
 pinst = PyEval_CallObject(pclass, pargs);
 Py_DECREF(pclass);
 Py_DECREF(pargs);
 if (pinst == NULL)
 error("Error calling module.klass()");

 /* result = instance.method(x,y) */
 pmeth = PyObject_GetAttrString(pinst, "method");
 Py_DECREF(pinst);
 if (pmeth == NULL)
 error("Can't fetch klass.method");
 pargs = Py_BuildValue("(ss)", arg1, arg2);
 if (pargs == NULL) {
 Py_DECREF(pmeth);
 error("Can't build arguments list");
 }
 pres = PyEval_CallObject(pmeth, pargs);
 Py_DECREF(pmeth);
 Py_DECREF(pargs);
 if (pres == NULL)
 error("Error calling klass.method");

 if (!PyArg_Parse(pres, "s", &cstr)) /* convert to C */
 error("Can't convert klass.method result");
 printf("%s\n", cstr);
 Py_DECREF(pres);
}
```

## The easy way: an extended API

- ◆ Higher-level interface from “*Programming Python*”
- ◆ Automates errors, reference counts, conversions
- ◆ Supports reloading and debugging embedded code
- ◆ Link with Python libs/objects + “pyembed\*” files

### file: objects2.c

```
#include <stdio.h>
#include "pyembed.h"

main () {
 int failflag;
 PyObject *pinst;
 char *arg1="sir", *arg2="robin", *cstr;

 failflag =
 Run_Function("module", "klass", /* module.klass() */
 "O", &pinst, "()") /* result, args */
 ||
 Run_Method(pinst, "method", /* pinst.method() */
 "s", &cstr, /* result fmt/ptr */
 "(ss)", arg1, arg2); /* args fmt/values*/

 printf("%s\n", (!failflag) ? cstr : "Can't call objects");
 Py_XDECREF(pinst);
}

% objects2
brave sir robin
```

## ***Running strings: results & name-spaces***

- ◆ Runs: [ upper('spam') + '!' ] in string module
- ◆ Expression result comes back as a Python object
- ◆ 'PyModule\_GetDict': module's name-space dictionary
- ◆ 'PyRun\_String': runs a code string in name-spaces
- ◆ Parse-mode flag: 'Py\_eval\_input' = expression

### **file codestring1.c**

```
#include <Python.h> /* standard API defs */

main() {
 /* error checking omitted! */
 char *cstr;
 PyObject *pstr, *pmod, *pdict;
 Py_Initialize();

 /* result = string.upper('spam') + '!' */
 pmod = PyImport_ImportModule("string"); /* namespace */
 pdict = PyModule_GetDict(pmod);
 pstr = PyRun_String("upper('spam') + '!',",
 Py_eval_input, pdict, pdict);

 /* convert result to C */
 PyArg_Parse(pstr, "s", &cstr);
 printf("%s\n", cstr);
 Py_DECREF(pmod);
 Py_DECREF(pstr); /* free exported objects */
}

% codestring1
SPAM!
```

## **The easy way, part II: extended API**

- ◆ Automates code strings, object calls, and methods
- ◆ Also supports running strings without modules

### **file: codestring2.c**

```
#include "pyembed.h"
#include <stdio.h>

main() {
 char *cstr;
 int err =
 Run_Codestr(
 PY_EXPRESSION, /* expr|stmt? */
 "upper('spam') + '!', "string", /* code,module */
 "s", &cstr); /* expr result */
 printf("%s\n", (!err) ? cstr : "Can't run string");
}

% codestring2
SPAM!
```

## ***Other code string possibilities***

- ◆ Making new dictionaries for string name-spaces
- ◆ Fetching strings from files, modules, HTML, databases
- ◆ Exporting C extension functions for communication
- ◆ Global variables for communication: ‘copy-in-copy-out’

## **Copy-in-copy-out**



**validate.py**

```
use QUANTITY, ORDERTYPE
set QUANTITY, MESSAGES
[. . .]
```

**C code...**

```
/* set input vars */
char *string = . . .
Set_Global("validate", "QUANTITY", "i", quantity);
Set_Global("validate", "ORDERTYPE", "s", ordertype);

/* run code string */
status = Run_CodeStr(PY_STATEMENT,
 string, "validate", "", NULL);
if (status == -1)
 PyErr_Print(); /* stack traceback */
else {
 /* fetch output vars */
 Get_Global("validate", "QUANTITY", "i", &quantity);
 Get_Global("validate", "MESSAGES", "s", &messages);
}
```

## Making namespace dictionaries

- ◆ Shamelessly stolen from “*Programming Python*”
- ◆ Same as using built-in ‘eval’/‘exec’, ‘{}’, ‘X[key]’:

```
dict = {}
dict['Y'] = 2
exec 'X = 99' in dict, dict
exec 'X = X+Y' in dict, dict
print dict['X'] # => 101
```

### file: basic4.c

```
#include <Python.h>

main() {
 int cval;
 PyObject *pdict, *pval;
 Py_Initialize();

 /* make a new namespace */
 pdict = PyDict_New();
 PyDict_SetItemString(pdict, "__builtins__",
 PyEval_GetBuiltins());

 /* dict['Y'] = 2 */
 PyDict_SetItemString(pdict, "Y", PyInt_FromLong(2));

 PyRun_String("X = 99", Py_file_input, pdict, pdict);
 PyRun_String("X = X+Y", Py_file_input, pdict, pdict);

 /* fetch dict['X'] */
 pval = PyDict_GetItemString(pdict, "X");
 PyArg_Parse(pval, "i", &cval); /* convert to C */
 printf("%d\n", cval); /* result=101 */
 Py_DECREF(pdict);
}
```

## ***Registering Python objects and strings***

- ♦ A strategy for code location/source
- ♦ Python passes code to a C extension
- ♦ C saves the code and runs it later
- ♦ May register objects or code strings
- ♦ Works best if Python is 'on top'

### **Components: extending + embedding**

- ♦ A C extension module
  - ♦ *Exports a registration function to Python ('setHandler')*
- ♦ A C event routing function
  - ♦ *Calls the registered Python object in response to events*
- ♦ A Python client program
  - ♦ *Registers functions, triggers events*

## Registration implementation

### file: cregister.c

```
#include <Python.h>
#include <stdlib.h>

/*****
/* 1) code to route events to Python object */
/* note that we could run strings here instead */
*****/

static PyObject *Handler = NULL; /* Python object in C */

void Route_Event(char *label, int count)
{
 char *cres;
 PyObject *args, *pres;

 /* call Python handler */
 args = Py_BuildValue("(si)", label, count);
 pres = PyEval_CallObject(Handler, args);
 Py_DECREF(args);

 if (pres != NULL) {
 /* use and decref handler result */
 PyArg_Parse(pres, "s", &cres);
 printf("%s\n", cres);
 Py_DECREF(pres);
 }
}

/*****
/* 2) python extension module to register handlers */
/* python imports this module to set handler objects */
*****/

static PyObject *
Register_Handler(PyObject *self, PyObject *args)
{
 /* save Python callable object */
 Py_XDECREF(Handler); /* called before? */
 PyArg_Parse(args, "O", &Handler); /* one argument? */
 Py_XINCRREF(Handler); /* add reference */
 Py_INCREF(Py_None); /* None=success */
 return Py_None;
}

static PyObject *
Trigger_Event(PyObject *self, PyObject *args)
{
 /* let Python simulate event caught by C */
 static count = 0;
 Route_Event("spam", count++);
 Py_INCREF(Py_None);
 return Py_None;
}

static struct PyMethodDef cregister_methods[] = {
 {"setHandler", Register_Handler}, /* name, addr */
 {"triggerEvent", Trigger_Event},
 {NULL, NULL}
}
```

```
};

void initcregister() /* this is called by Python */
{
 /* on first "import cregister" */
 (void) Py_InitModule("cregister", cregister_methods);
}
```

## Python client program

- ◆ Combines extending and embedding
- ◆ Extending:            Python encloses C
- ◆ Embedding:          C calls Python on events
- ◆ C may also call embedded python code to register

### file: register.py

```
handle an event, return a result (or None)

def function1(label, count):
 return "%s number %i..." % (label, count)

def function2(label, count):
 return label * count

register handlers, trigger events

import cregister
cregister.setHandler(function1)
for i in range(3):
 cregister.triggerEvent() # simulate events caught by C

cregister.setHandler(function2)
for i in range(3):
 cregister.triggerEvent() # routes events to function2

% python register.py
spam number 0...
spam number 1...
spam number 2...
spamspamspam
spamspamspamspam
spamspamspamspamspam
```

## Registration tradeoffs

- ◆ **+: Granularity**

*Associating actions with objects without external files*

- ◆ **-: Application structure**

*Requires Python on top, or extra embedding logic*

- ◆ **-: Complexity**

*Might require extra coding step to register code*

- ◆ **-: Reloading code**

*Difficult to reload without going through modules*

## Building the example

### Solaris

```
add the extension modules as dynamically linked modules;
when first imported, they are located in a dir on
$PYTHONPATH, and loaded into the process.
```

```
cregister.so: cregister.c
 $(Cc) cregister.c $(CFLAGS) -DDEBIG -KPIC -o cregister.o
 ld -G cregister.o -o $@
```

```
environ.so: environ.c
 $(Cc) environ.c $(CFLAGS) -KPIC -o environ.o
 ld -G environ.o -o $@
```

### Linux

```
PY = /home/mark/python1.5.2-ddjcd/Python-1.5.2
PYINC = -I$(PY)/Include -I$(PY)
cregister.so: cregister.c
 gcc cregister.c -g $(PYINC) -fpic -shared
```

## Accessing C variables in Python

### ◆ Using an extension module

```
import cvars # import C variable wrapper module
cvars.setX(24) # set C's X from Python
print cvars.getY() # fetch C's Y from Python
```

### ◆ Using an extension type

```
import cvars # import C variable wrapper module
c = cvars.interface() # make interface object instance
c.X = (24) # set attributes -> C variable
print c.Y # get attributes -> C variable
```

### ◆ Using copy-in-copy-out

1. C copies X and Y values to variables in module M
2. C runs embedded code in module M's name-space
3. C fetches the final values of the Python variables

## C API equivalents in Python

- ◆ Python may be embedded in C or Python
- ◆ See section “Dynamic coding” for Python tools

<i>C API tool</i>	<i>Python tool</i>
PyImport_ImportModule	import
PyEval_CallObject	apply
PyRun_String	exec/eval
PyObject_GetAttrString	getattr

## ***Running code files from C***

- ◆ **'PyRun\_File', 'PyRun\_SimpleFile'**
- ◆ **Module imports and reloads**
- ◆ **system, popen, fork/exec**

## ***Precompiling strings into byte-code***

- ◆ *Modules are compiled once, on first import*
- ◆ *Raw strings compiled when run, unless precompile*

- ◆ **Compile to byte code object**

```
PyCodeObject*
Py_CompileString(char *string,
 char *filename, int parsemode);
```

- ◆ **Run byte-code object**

```
PyObject*
PyEval_EvalCode(PyCodeObject *code,
 PyObject *globalnamesdict,
 PyObject *localnamesdict);
```



## ***Embedding under C++***

- ♦ *Python is coded in portable ANSI C*
- ♦ *Normal C++  $\Rightarrow$  C mixing rules apply*
- ♦ *No special considerations when embedding*
  
- ♦ **Python header files**
  - ♦ *Automatically wrapped in extern "C", and may be included in C++ programs freely*
  
- ♦ **Python libraries**
  - ♦ *Don't need to be recompiled by the C++ compiler to link: API entry points are all extern "C"*

## ***More on object reference counts***

- ♦ Python uses reference-count garbage collection
- ♦ New objects returned to C with a reference
- ♦ C must INCRREF other objects to retain them
- ♦ C must DECRREF objects it no longer needs
- ♦ XINCRREF/XDECRREF ignore NULL pointers

### **Resources:**

- ♦ New API documentation
  - ♦ *<http://www.python.org/doc>, *Python/C API manual**

- ◆ Older API documentation
  - ◆ <http://www.python.org/doc/ext/ext.html>
- ◆ Other references
  - ◆ ***Programming Python, Editions 2+***
- ◆ Abstract object API
  - ◆ ***Tools in "abstract.h" header file are well-defined***
- ◆ Extended API in chapter 15 of “Programming Python”
  - ◆ ***Run\_Function, etc., handle most details***
- ◆ Embedding examples in books, on the ‘net, etc.
  - ◆ <http://rmi.net/~lutz/newex.html>
- ◆ Check the C source code of the API
  - ◆ ***Not as hard as you may think!***

### Common API calls

New Name (1.3+)	INCREFs?	Python Equivalent
PyImport_AddModule	No	n/a
PyImport_ImportModule	Yes	import module
PyImport_ReloadModule	Yes	reload(module)
PyImport_GetModuleDict	No	sys.modules
PyModule_GetDict	No	module.__dict__
PyDict_GetItemString	No	dict[key]
PyDict_SetItemString	n/a	dict[key]=val
PyObject_GetAttrString	Yes	getattr(obj, attr)
PyObject_SetAttrString	n/a	setattr(obj, attr, val)
PyArg_ParseTuple	n/a	n/a
Py_BuildValue	Yes	n/a
PyEval_CallObject	Yes	apply(func, argtuple)
PyRun_String	Yes	eval(expr), exec stmt
PyErr_Print	n/a	traceback.print_exc
PyDict_New	Yes	{ }

## Integration error handling

- ◆ **Extending: sending errors to Python**
  - ◆ *C extensions raise exceptions by returning NULL*
  - ◆ *'PyErr\_SetString' sets exception's name and data*
  - ◆ *C may propagate error instead of setting*
- ◆ **Embedding: catching Python errors**
  - ◆ *API return values: NULL, or integer status codes*
  - ◆ *'PyErr\_Fetch' fetches exception info*
  - ◆ *'PyErr\_Print' shows Python stack traceback on stderr*
  - ◆ *Extended API functions return -1 or NULL on first error*

### Fetching exception information in C

#### file: pyerrors.c

```
#include <Python.h>
#include <stdio.h>
char save_error_type[1024], save_error_info[1024];

PyerrorHandler(char *msgFromC)
{
 /* process Python-related errors */
 /* call after Python API raises an exception */

 PyObject *errobj, *errdata, *errtraceback, *pystring;
 printf("%s\n", msgFromC);

 /* get latest python exception info */
 PyErr_Fetch(&errobj, &errdata, &errtraceback);

 pystring = NULL;
 if (errobj != NULL &&
 (pystring = PyObject_Str(errobj)) != NULL &&
 (PyString_Check(pystring))
)
 strcpy(save_error_type, PyString_AsString(pystring));
 else
```

```

 strcpy(save_error_type, "<unknown exception type>");
 Py_XDECREF(pystring);

 pystring = NULL;
 if (errdata != NULL &&
 (pystring = PyObject_Str(errdata)) != NULL &&
 (PyString_Check(pystring))
)
 strcpy(save_error_info, PyString_AsString(pystring));
 else
 strcpy(save_error_info, "<unknown exception data>");
 Py_XDECREF(pystring);

 printf("%s\n%s\n", save_error_type, save_error_info);
 Py_XDECREF(errobj);
 Py_XDECREF(errdata); /* caller owns all 3 */
 Py_XDECREF(errtraceback); /* already NULL'd out */
 }

```

## Automated integration tools

### ◆ SWIG

- ◆ *Generates Python interfaces to external C/C++ libraries*
- ◆ *Uses C/C++ declarations and interface description files*
- ◆ *For C++ class: generates C type + Python wrapper class*
- ◆ *See unit 15, Programming Python 2<sup>nd</sup> edition, and <http://www.swig.org/>*

### ◆ ILU

- ◆ *Implements CORBA distributed-object systems*
- ◆ *Uses interface description files*
- ◆ *Platform and language independent*

### ◆ Abstract object API

- ◆ *C API to create/process Python objects generically*
- ◆ *In Python's "abstract.h": exports slicing, concatenation, etc.*

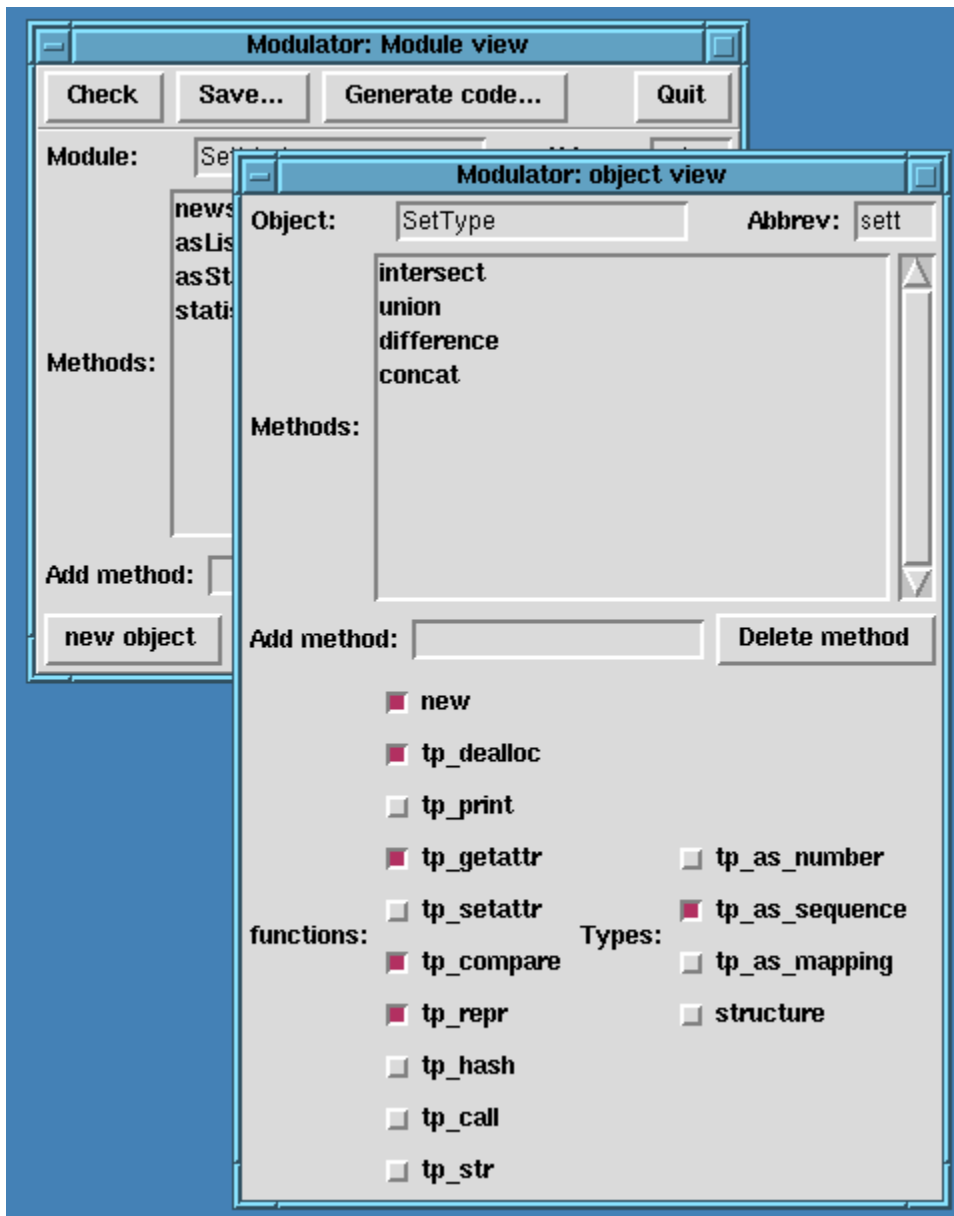
### ◆ Embedded call API

- ◆ *C API to simplify common embedding tasks*
- ◆ *In chapter 15 of "Programming Python" 1<sup>st</sup> Edition*

### ◆ Modulator

- ♦ *Tkinter GUI: generates skeleton C module/type files*
- ♦ *Users fill in the blanks with application-specific logic*
- ♦ **Other: ActiveX/COM interfaces**

## Modulator in action



## ***Lab Session 12***

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

[Click here to go to lecture example files](#)

## 17. Resources

### Topics

- ♦ Internet resources
- ♦ Books
- ♦ Conferences and services

### *Internet resources*

- [www.python.org](http://www.python.org)
- [www.google.com](http://www.google.com)

### **Others**

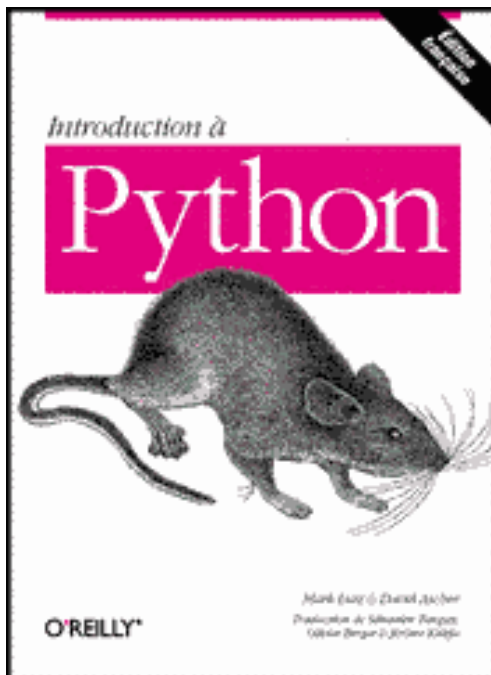
Python's support mail-list	<a href="mailto:help@python.org">mailto:help@python.org</a>
Instructor's web site	<a href="http://www.rmi.net/~lutz">http://www.rmi.net/~lutz</a>
O'Reilly's web site	<a href="http://www.oreilly.com/">http://www.oreilly.com/</a>
Python's support mail-list	<a href="mailto:help@python.org">mailto:help@python.org</a>
Python online docs	<a href="http://www.python.org/doc">http://www.python.org/doc</a>
Python starship site	<a href="http://starship.python.net/">http://starship.python.net/</a>
Vaults of Parnassus site	<a href="http://www.vex.net/parnassus/">http://www.vex.net/parnassus/</a>
PyPi site	<a href="http://cheeseshop.python.org/pypi/">http://cheeseshop.python.org/pypi/</a>

## Python books

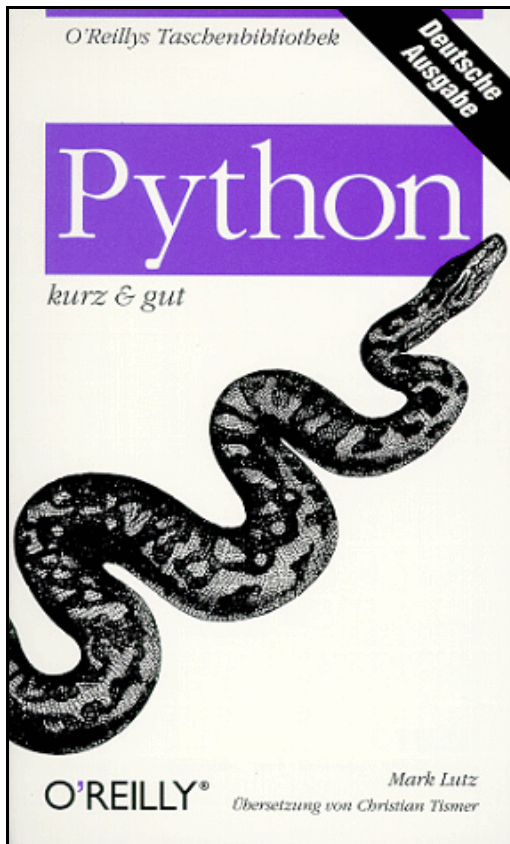
- Learning Python
- Programming Python
- Python Pocket Reference
- Python Cookbook
- Python in a Nutshell
- Python Essential Reference
- ...

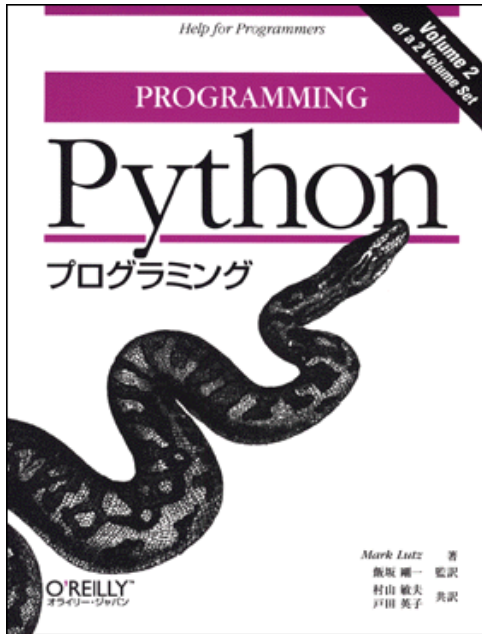
*I stopped updating the original list here when there were 50 books in 2002 (!). For more details, search for Python at [amazon.com](http://amazon.com), or see the book pages at [www.python.org/doc](http://www.python.org/doc).*

### O'Reilly Translations









## ***Python conferences and services***

- ◆ Annual gatherings (PyCon, formerly IPCn)
- ◆ O'Reilly Python Conference, Open Source Convention
- ◆ Local Python user groups ("PIGgies")
- ◆ European Python Conference (annual)
- ◆ Others: Brazil, Korean Python convention
- ◆ Commercial support, consulting: see [www.python.org](http://www.python.org)

## Lab Session 12

[Click here to go to lab exercises](#)

[Click here to go to exercise solutions](#)

[Click here to go to solution source files](#)

## And finally...

Click the link below to play audio file “sousa.au”—the Monty Python theme song, provided your machine supports audio playback.

*[“Roll the closing credits.”](#)*



## Laboratory exercises

The lab exercises below ask students to write original programs, as well as modify pre-coded examples. Later exercises demonstrate more advanced concepts and tools: simple Tkinter GUIs, C integration, etc. The final lab also points users to online Python resources (the web page, etc.), provided the lab machines have Internet access. [Lecture examples source-code](#), as well as [answers to lab exercises](#), are provided on this CD; click on the links in this paragraph to go to these resources, or ask the instructor for assistance.

Note: don't expect to finish all the questions in each lab. Some of these exercises are more involved than others, and most of the labs have more exercises than you'll probably be able to finish in a half or full hour. Students are encouraged to skip around if they find they need more or less time for some questions. Feel free to return to prior labs at any time, regardless of where we are at in the lecture schedule.

## Lab 1: Using the interpreter

[Go to solutions](#)

[Go to solution files](#)

1. *Interaction.* Using a system command-line, IDLE, or other, start the Python interactive command line (`>>>` prompt), and type the expression: `"Hello World!"` (including the quotes). The string should be echoed back to you. The purpose of this exercise is to get your environment configured to run Python. In some scenarios, you may need to first run a `cd` shell command, type the full path to the `python` executable, or add its path to your `PATH` environment variable. Set it in your `.cshrc` or `.kshrc` file to make Python permanently available on Unix systems; use a `setup.bat`, `autoexec.bat`, or the environment variable GUI on Windows.
2. *Programs.* With the text editor of your choice, write a simple module file—a file containing the single statement: `print 'Hello module world!'`. Store this statement in a file named `module1.py`. Now, run this file by using any launch option you like: running it in IDLE, clicking on its file icon, passing it to the Python interpreter program on the system shell's command line, and so on. In fact, experiment by running your file with as many of the launch techniques we've seen in this unit as you can. Which technique seems easiest (there is no right answer to this one, of course)?
3. *Modules.* Next, start the Python interactive command line (`>>>` prompt) and import the module you wrote in the prior exercise. Does your `PYTHONPATH` setting need to include the directory where the file is stored? Try moving the file to a different directory and importing it again from its original directory; what happens? (Hint: is there still a file named `module1.pyc` in the original directory?)
4. *Scripts.* If your platform supports it, add the `#!` line to the top of your `module1.py` module, give the file executable privileges, and run it directly as an executable. What does the first line need to contain? Skip that if you are working on a Windows machine (`#!` usually only has meaning on Unix and Linux); instead try running your file by listing just its name in a DOS console window (this works on recent flavors of Windows), or the “Start/Run...” dialog box.
5. *Errors.* Experiment with typing mathematical expressions and assignments at the Python interactive command line. First type the expression: `1 / 0`; what happens? Next, type a variable name you haven't assigned a value to yet; what happens this time?

You may not know it yet, but you're doing exception processing, a topic we'll explore in depth in a later unit. As we'll learn then, you are technically triggering what's known as the *default exception handler*—logic that prints a standard error message.

For full-blown source-code *debugging* chores, IDLE includes a GUI debugging interface (select `Debug` before running your script), and a Python standard library module named `pdb` provides a command-line debugging interface (more on `pdb` later). When first starting out, though, Python's default error messages will probably be as much error handling as you need—they give the cause of the error, as well as the lines in your code were active when the error occurred.

6. *Breaks.* At the Python command line, type:

```
L = [1, 2]
L.append(L)
L
```

What happens? If you're using a Python newer than release 1.5, you'll probably see a strange output. If you're using a Python version older than 1.5.1, a Ctrl-C key combination will probably help on most platforms. Why do you think this occurs? What does Python report when you type the Ctrl-C key combination? Warning: if you do have a Python older than release 1.5.1, make sure your machine can stop a program with a break-key combination of some sort before running this test, or you may be waiting a long time.

7. *Documentation.* Spend at least 6 minutes browsing the Python library and language manuals before moving on, to get a feel for the available tools in the standard library, and the structure of the documentation set. It takes at least this long to become familiar with the location of major topics in the manual set; once you do, though, it's easy to find what you need. You can find this manual in the Python "Start" button entry on Windows, in the "Help" pulldown menu in IDLE, or online at [www.python.org](http://www.python.org). We'll also have a few more words to say about the manuals, and other documentation sources available (including PyDoc and the `help` function), in the statements unit. If you still have time to kill, go explore the Python website ([www.python.org](http://www.python.org)), and the Vaults of Parnassus site. Especially check out the python.org documentation and search pages; they can be crucial resources in practice.

## Lab 2: Types and operators

[Go to solutions](#)

[Go to solution files](#)

*If you have limited time, start with exercise 11 (the most practical), and then work from first to last as time allows. This is all fundamental material, though, so try to do as many of these as you can.*

1. *The basics.* Experiment interactively with the common type operations found in this unit's tables. To get you started, bring up the Python interactive interpreter, type the expressions below, and try to explain what's happening in each case:

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)

('x',)[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
```

```

([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
D.keys(), D.values(), D.has_key((1,2,3))

[[]], ["", [], (), {}, None]

```

2. *Indexing and slicing.* At the interactive prompt, define a list named `L` that contains four strings or numbers (e.g., `L=[0, 1, 2, 3]`). Now, let's experiment with some boundary cases.
  - a) What happens when you try to index out of bounds (e.g., `L[4]`)?
  - b) What about slicing out of bounds (e.g., `L[-1000:100]`)?
  - c) Finally, how does Python handle it if you try to extract a sequence in reverse—with the lower bound greater than the higher bound (e.g., `L[3:1]`)? Hint: try assigning to this slice (`L[3:1]='?'`) and see where the value is put. Do you think this may be the same phenomenon you saw when slicing out of bounds?
3. *Indexing, slicing, and del.* Define another list `L` with four items again, and assign an empty list to one of its offsets (e.g., `L[2]=[]`): what happens? Then try assigning an empty list to a slice (`L[2:3]=[]`): what happens now? Recall that slice assignment deletes the slice and inserts the new value where it used to be. The `del` statement deletes offsets, keys, attributes, and names: try using it on your list to delete an item (e.g., `del L[0]`). What happens if you `del` an entire slice (`del L[1:]`)? What happens when you assign a nonsequence to a slice (`L[1:2]=1`)?
4. *Tuple assignment.* What do you think is happening to `x` and `y` when you type this sequence? We'll return to this construct in the next unit, but it has something to do with the tuples we've seen here.

```

>>> x = 'spam'
>>> y = 'eggs'
>>> x, y = y, x

```

5. *Dictionary keys.* Consider the following code fragments:

```

>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'

```

We learned that dictionaries aren't accessed by offsets; what's going on here? Does the following shed any light on the subject? (Hint: strings, integers, and tuples share which type category?)

```

>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}

```

6. *Dictionary indexing.* Create a dictionary named `D` with three entries, for keys `'a'`, `'b'`, and `'c'`. What happens if you try to index a nonexistent key (`D['d']`)? What does Python do if you try to assign to a nonexistent key (e.g., `D['d']='spam'`)? How does this compare to out-of-bounds assignments and references for lists? Does this sound like the rule for variable names?

7. *Generic operations.* Run interactive tests to answer the following questions.
  - a) What happens when you try to use the `+` operator on different/mixed types (e. g., string + list, list + tuple)?
  - b) Does `+` work when one of the operands is a dictionary?
  - c) Does the `append` method work for both lists and strings? How about the using the `keys` method on lists? (Hint: What does `append` assume about its subject object?)
  - d) Finally, what type of object do you get back when you slice or concatenate two lists or two strings?
8. *String indexing.* Define a string `S` of four characters: `S = "spam"`. Then type the following expression: `S[0][0][0][0][0]`. Any clues as to what's happening this time? (Hint: recall that a string is a collection of characters, but Python characters are one-character strings.) Does this indexing expression still work if you apply it to a list such as: `['s', 'p', 'a', 'm']`? Why?
9. *Immutable types.* Define a string `S` of 4 characters again: `S = "spam"`. Write an assignment that changes the string to `"slam"`, using only slicing and concatenation. Could you perform the same operation using just indexing and concatenation? How about index assignment?
10. *Nesting.* Write a data-structure that represents your personal information: name (first, middle, last), age, job, address, email ID, and phone number. You may build the data structure with any combination of built-in object types you like: lists, tuples, dictionaries, strings, numbers. Then access the individual components of your data structures by indexing. Do some structures make more sense than others for this object?
11. *Files.* Write a script that creates a new output file called `myfile.txt` and writes the string `"Hello file world!"` in it. Then write another script that opens `myfile.txt`, and reads and prints its contents. Run your two scripts from the system command line. Does the new file show up in the directory where you ran your scripts? What if you add a different directory path to the filename passed to open? Note: file write methods do not add newline characters to your strings; add an explicit `'\n'` at the end of the string if you want to fully terminate the line in the file.
12. *The `dir` function.* Try typing the following expressions at the interactive prompt. Starting with Version 1.5, the `dir` builtin function has been generalized to list all attributes of any Python object you're likely to be interested in. If you're using an earlier version than 1.5, the `__methods__` scheme has the same effect. If you're using Python 2.2, `dir` is probably the only of these that will work.

```

[].__methods__ # 1.4 or 1.5
dir([]) # 1.5 and later
{ }.__methods__
dir({ })

```



## Lab 3: Basic statements

[Go to solutions](#)

[Go to solution files](#)

### 1. Coding basic loops.

- Write a `for` loop which prints the ASCII code of each character in a string named `S`. Use the built-in function `ord(character)` to convert each character to an ASCII integer (test it interactively to see how it works, or see the Python library manual).
- Next, change your loop to compute the *sum* of the ASCII codes of all the characters in a string.
- Finally, modify your code again to return a *new list*, which contains the ASCII codes of each character in the string. Does this expression have a similar effect—`map(ord, S)`? (Hint: `map` applies a function to all nodes of a sequence in turn, and collects all the results.)

### 2. Backslash characters. What happens on your machine when you type the following code interactively?

```
for i in range(50):
 print 'hello %d\n\a' % i
```

*Warning:* Outside IDLE, this example may beep at you, so you may not want to run it in a crowded lab (unless you happen to enjoy getting lots of attention). Within IDLE, you'll get odd characters instead. Hint: see the full set of backslash escape characters in the string literals section of Python's language reference manual.

### 3. Sorting dictionaries. In lecture 2, we saw that dictionaries are *unordered* collections. Write a `for` loop which prints a dictionary's items in sorted (ascending) order. Hint: use the dictionary `keys` and `list sort` methods.

### 4. Program logic alternatives. Part of learning to program in Python is learning which coding alternatives work better than others. Consider the following code, which uses a `while` loop and `found` flag to search a list of powers-of-2, for the value of 2 raised to the power 5 (32). It's stored in a module file called `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = i = 0
while not found and i < len(L):
 if 2 ** X == L[i]:
 found = 1
 else:
 i = i+1

if found:
 print 'at index', i
else:
 print X, 'not found'

C:\book\tests> python power.py
at index 5
```

As is, the example doesn't follow normal Python coding techniques. Follow the steps below to improve it; for all of the transformations, you may type your code interactively, or store it in a script file run from the system command line (though using a file will make this exercise much easier).

- a) First, rewrite this code with a `while` loop `else`, to eliminate the `found` flag and final `if` statement.
- b) Next, rewrite the example to use a `for` loop with an `else`, to eliminate the explicit list indexing logic. Hint: to get the index of an item, use the list *index* method (`L.index(X)` returns the offset of the first `X` in list `L`).
- c) Now, remove the loop completely by rewriting the examples with a simple `in` operator membership expression (to see how, interactively type this: `2 in [1, 2, 3]`).
- d) Finally, use a `for` loop and the list *append* method to generate the powers-of-2 list (`L`), instead of hard-coding a list constant.
- e) Deeper thoughts: (1) Do you think it would improve performance to move the `2**X` expression outside the loops? How would you code that? (2) As we saw in exercise 1, Python also includes a `map(function, list)` tool which could be used to generate the powers-of-2 list too, as follows: `map((lambda x: 2**x), range(7))`. Try typing this code interactively; we'll meet `lambda` more formally in the next lecture.

## Lab 4: Functions

[Go to solutions](#)

[Go to solution files](#)

1. *Basics*. At the Python interactive prompt, write a function which prints its single argument to the screen, and call it interactively, passing a variety of object types: string, integer, list, dictionary. Then try calling it without passing any argument: what happens? What happens when you pass two arguments?
2. *Arguments*. Write a function called `adder` in a Python module file. `adder` should accept two arguments, and return the sum (or concatenation) of its two arguments. Then add code at the bottom of the file to call the function with a variety of object types (two strings, two lists, two floating-points), and run this file as a script from the system command line. Do you have to print the call statement results to see results on the screen?
3. *Varargs*. Generalize the `adder` function you wrote in the last exercise to compute the sum of an arbitrary number of arguments, and change the calls to pass more or less than two. What type is the return value `sum`? (Hints: a slice like `S[:0]` returns an empty sequence of the same type as `S`, and the `type` built-in function can be used to test types, but the `max` function we wrote in class provides an easier approach). What happens if you pass in arguments of different types? What about passing in dictionaries?

4. *Keywords.* Change the `adder` function from exercise (2) to accept and add three arguments: `"def adder(good, bad, ugly)"`. Now, provide default values for each argument, and experiment with calling the function interactively. Try passing 1, 2, 3, and 4 arguments. Then, try passing keyword arguments. Does the call `"adder(ugly=1, good=2)"` work? Why? Finally, generalize the new `adder` to accept and add an arbitrary number of keyword arguments, much like exercise (3), but you'll need to iterate over a dictionary, not a tuple (hint: the `dictionary.keys()` method returns a list you can step through with a `for` or `while`).
5. Write a function called `copyDict(dict)`, which copies its dictionary argument. It should return a new dictionary with all the items in its argument. Use the dictionary `keys` method to iterate. Copying sequences is easy (`X[:]` makes a top-level copy); does this work for dictionaries too?
6. Write a function called `addDict(dict1, dict2)` which computes the union of two dictionaries. It should return a new dictionary, with all the items in both its arguments (assumed to be dictionaries). If the same key appears in both arguments, feel free to pick a value from either. Test your function by writing it in a file and running the file as a script. What happens if you pass lists instead of dictionaries? How could you generalize your function to handle this case too? (Hint: see the `type` built-in function used earlier). Does the order of arguments passed matter?
7. *Argument matching.* First, define the following six functions (either interactively, or in an importable module file):

```
def f1(a, b): print a, b # normal args
def f2(a, *b): print a, b # positional varargs
def f3(a, **b): print a, b # keyword varargs
def f4(a, *b, **c): print a, b, c # mixed modes
def f5(a, b=2, c=3): print a, b, c # defaults
def f6(a, b=2, *c): print a, b, c # defaults + positional varargs
```

Now, test the following calls interactively, and try to explain each result; in some cases, you'll probably need to fall back on the matching algorithm shown in the lecture. Do you think mixing matching modes is a good idea in general? Can you think of cases where it would be useful anyhow?

```
>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)

>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)
```

8. *List comprehensions.* Write code to build a new list containing the square roots of all the numbers in this list: `[2, 4, 9, 16, 25]`. Code this as a `for` loop first, then as a `map` call, and finally as a list comprehension. Use the `sqrt` function in the built-in `math` module to do the calculation (i.e., `import math`, and say `math.sqrt(x)`). Of the three, which approach do you like best?

## Lab 5: Modules

[Go to solutions](#)

[Go to solution files](#)

1. *Basics, import.* Write a program that counts lines and characters in a file (similar in spirit to “wc” on Unix). With your favorite text editor, code a Python module called `mymod.py`, which exports three top-level names:

- a) A `countLines(name)` function that reads an input file and counts the number of lines in it (hint: `file.readlines()` does most of the work for you, and `len` does the rest)
- b) A `countChars(name)` function that reads an input file and counts the number of characters in it (hint: `file.read()` returns a single string)
- c) A `test(name)` function that calls both counting functions with a given input filename. Such a filename generally might be passed-in, hard-coded, input with `raw_input`, or pulled from a command-line via the `sys.argv` list; for now, assume it’s a passed-in function argument.

All three `mymod` functions should expect a filename string to be passed in. If you type more than two or three lines per function, you’re working much too hard—use the hints listed above!

Now, test your module interactively, using `import` and name qualification to fetch your exports. Does your `PYTHONPATH` need to include the directory where you created `mymod.py`? Try running your module on itself: e.g., `test("mymod.py")`. Note that `test` opens the file twice; if you’re feeling ambitious, you may be able to improve this by passing an open file object into the two count functions (hint: `file.seek(0)` is a file rewind).

2. *from/from\*.* Test your `mymod` module from Exercise 1 interactively, by using `from` to load the exports directly, first by name, then using the `from*` variant to fetch everything.
3. *\_\_main\_\_.* Now, add a line in your `mymod` module that calls the `test` function automatically only when the module is run as a script, not when it is imported. The line you add will probably test the value of `__name__` for the string `"__main__"`, as shown in this unit. Try running your module from the system command line; then, import the module and test its functions interactively. Does it still work in both modes?
4. *Nested imports.* Write a second module, `myclient.py`, which imports `mymod` and tests its functions; run `myclient` from the system command line. If `myclient` uses `from` to fetch from `mymod`, will `mymod`’s functions be accessible from the top level of `myclient`? What if it imports with `import` instead? Try coding both variations in `myclient` and test interactively, by importing `myclient` and inspecting its `__dict__`.
5. *Package imports.* Finally, import your file from a package. Create a subdirectory called `mypkg` nested in a directory on your module import search path, move the `mymod.py` module file you created in exercises 1 or 3 into the new directory, and try to import it with a package import of the form: `import mypkg.mymod`.

You'll need to add an `__init__.py` file in the directory your module was moved to in order to make this go, but it should work on all major Python platforms (that's part of the reason Python uses "." as a path separator). The package directory you create can be simply a subdirectory of the one you're working in; if it is, it will be found via the home directory component of the search path, and you won't have to configure your path. Add some code to your `__init__.py`, and see if it runs on each import.

6. *Reload*. Experiment with module reloads: perform the tests in the `changer.py` example, changing the called function's message and/or behavior repeatedly, without stopping the Python interpreter. Depending on your system, you might be able to edit `changer` in another window, or suspend the Python interpreter and edit in the same window (on Unix, a Ctrl-Z key combination usually suspends the current process, and a `fg` command later resumes it).
7. *[Optional] Circular imports* (and other acts of cruelty). In the section on recursive import gotchas, importing `recur1` raised an error. But if we restart Python and import `recur2` interactively, the error doesn't occur: test and see this for yourself. Why do you think it works to import `recur2`, but not `recur1`? (Hint: Python stores new modules in the built-in `sys.modules` table (a dictionary) before running their code; later imports fetch the module from this table first, whether the module is "complete" yet or not.) Now try running `recur1` as a top-level script file: `% python recur1.py`. Do you get the same error that occurs when `recur1` is imported interactively? Why? (Hint: when modules are run as programs they aren't imported, so this case has the same effect as importing `recur2` interactively; `recur2` is the first module imported.) What happens when you run `recur2` as a script?

## Lab 6: Classes

[Go to solutions](#)

[Go to solution files](#)

*Note: The last 2 (or 3) are the most fun of this bunch; you might want to start with these first, and then come back to work top down.*

1. *Inheritance*. Write a class called `Adder` that exports a method `add(self, x, y)` that prints a "Not Implemented" message. Then define two subclasses of `Adder` that implement the `add` method:
  - a) `ListAdder`, with an `add` method that returns the concatenation of its two list arguments
  - b) `DictAdder`, with an `add` method that returns a new dictionary with the items in both its two dictionary arguments (any definition of addition will do)

Experiment by making instances of all three of your classes interactively and calling their `add` methods.

Now, extend your `Adder` superclass to save an object in the instance with a constructor (e.g., assign `self.data` a list or a dictionary) and overload the `+` operator with an `__add__` to automatically dispatch to your `add` methods (e.g., `X+Y` triggers `X.add(X.data, Y)`). Where is the best place to

put the constructors and operator overload methods (i.e., in which classes)? What sorts of objects can you add to your class instances?

In practice, you might find it easier to code your `add` methods to accept just one real argument (e.g., `add(self, y)`), and add that one argument to the instance's current data (e.g., `self.data+y`). Does this make more sense than passing two arguments to `add`? Would you say this makes your classes more "object-oriented"?

2. *Operator overloading.* Write a class called `Mylist` that shadows ("wraps") a Python list: it should overload most list operators and operations—`+`, indexing, iteration, slicing, and list methods such as `append` and `sort`. See the Python reference manual for a list of all possible methods to support. Also provide a constructor for your class that takes an existing list (or a `Mylist` instance) and copies its components into an instance member. Experiment with your class interactively. Things to explore:
  - a) Why is copying the initial value important here?
  - b) Can you use an empty slice (e.g., `start[:]`) to copy the initial value if it's a `Mylist` instance?
  - c) Is there a general way to route list method calls to the wrapped list?
  - d) Can you add a `Mylist` and a regular list? How about a list and a `Mylist` instance?
  - e) What type of object should operations like `+` and slicing return; how about indexing?
  - f) If you are working with a more recent Python release (version 2.2 or later), you may implement this sort of wrapper class either by embedding a real list in a stand-alone class, or by extending the built-in list type with a subclass. Which is easier and why?
3. *Subclassing.* Now, make a subclass of `Mylist` from Exercise 2 called `MylistSub`, which extends `Mylist` to print a message to `stdout` before each overloaded operation is called and counts the number of calls. `MylistSub` should inherit basic method behavior from `Mylist`. For instance, adding a sequence to a `MylistSub` should print a message, increment the counter for `+` calls, and perform the superclass's method. Also introduce a new method that displays the operation counters to `stdout` and experiment with your class interactively. Do your counters count calls per instance, or per class (for all instances of the class)? How would you program both of these? (Hint: it depends on which object the count members are assigned to: class members are shared by instances, `self` members are per-instance data.)
4. *Metaclass methods.* Write a class called `Meta` with methods that intercept every attribute qualification (both fetches and assignments) and prints a message with their arguments to `stdout`. Create a `Meta` instance and experiment with qualifying it interactively. What happens when you try to use the instance in expressions? Try adding, indexing, and slicing the instance of your class.
5. *Set objects.* Experiment with the set class described in this unit. Run commands to do the following sorts of operations:
  - a) Create two sets of integers, and compute their intersection and union by using `&` and `|` operator expressions.
  - b) Create a set from a string, and experiment with indexing your set; which methods in the class are called?
  - c) Try iterating through the items in your string set using a `for` loop; which methods run this time?
  - d) Try computing the intersection and union of your string set and a simple Python string; does it work?
  - e) Now, extend your set by subclassing to handle arbitrarily many operands using a `*args` argument form (hint: see the function versions of these algorithms in the functions unit). Compute intersections and unions of multiple operands with your set subclass. How can you intersect three or more sets, given that `&` has only two sides?

- f) How would you go about emulating other list operations in the set class? (Hints: `__add__` can catch concatenation, and `__getattr__` can pass most list method calls off to the wrapped list.)

6. *Composition*. Simulate a fast-food ordering scenario by defining four classes:

- a) Lunch: a container and controller class
- b) Customer: the actor that buys food
- c) Employee: the actor that a customer orders from
- d) Food: what the customer buys

To get you started, here are the classes and methods you'll be defining:

```
class Lunch:
 def __init__(self) # make/embed Customer and Employee
 def order(self, foodName) # start a Customer order simulation
 def result(self) # ask the Customer what kind of Food it has

class Customer:
 def __init__(self) # initialize my food to None
 def placeOrder(self, foodName, employee) # place order with an Employee
 def printFood(self) # print the name of my food

class Employee:
 def takeOrder(self, foodName) # return a Food, with requested name

class Food:
 def __init__(self, name) # store food name
```

The order simulation works as follows:

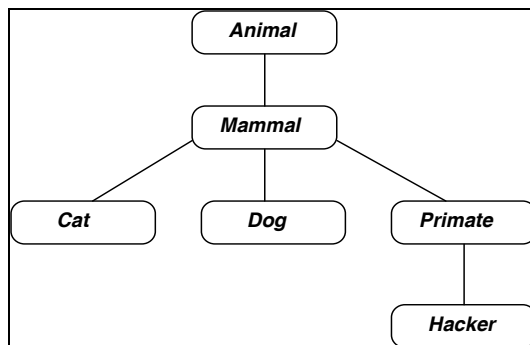
- a) The `Lunch` class's constructor should make and embed an instance of `Customer` and `Employee`, and export a method called `order`. When called, this `order` method should ask the `Customer` to place an order, by calling its `placeOrder` method. The `Customer`'s `placeOrder` method should in turn ask the `Employee` object for a new `Food` object, by calling the `Employee`'s `takeOrder` method.
- b) `Food` objects should store a food name string (e.g., "burritos"), passed down from `Lunch.order` to `Customer.placeOrder`, to `Employee.takeOrder`, and finally to `Food`'s constructor. The top-level `Lunch` class should also export a method called `result`, which asks the customer to print the name of the food it received from the `Employee` via the order (this can be used to test your simulation).
- c) Note that `Lunch` needs to either pass the `Employee` to the `Customer`, or pass itself to the `Customer`, in order to allow the `Customer` to call `Employee` methods.

Experiment with your classes interactively by importing the `Lunch` class, calling its `order` method to run an interaction, and then calling its `result` method to verify that the `Customer` got what he or she ordered. If you prefer, you can also simply code test cases as self-test code in the file where your classes are defined, using the module `__name__` trick we met in the modules unit. In this simulation, the `Customer` is the active agent; how would your classes change if `Employee` were the object that initiated customer/ employee interaction instead?

7. *Zoo Animal Hierarchy: (If this example was skipped in class)* Consider the class tree sketched in the figure below. Code a set of 6 class statements to model this taxonomy with Python inheritance. Then, add a `speak` method to each of your classes which prints a unique message, and a `reply` method in your top-level `Animal` superclass which simply calls `self.speak` to invoke the category-specific message printer in a subclass below (remember, this will kick off an independent

inheritance search from `self`). Finally, remove the `speak` method from your `Hacker` class, so that it picks up the default above it. When you're finished, your classes should work this way:

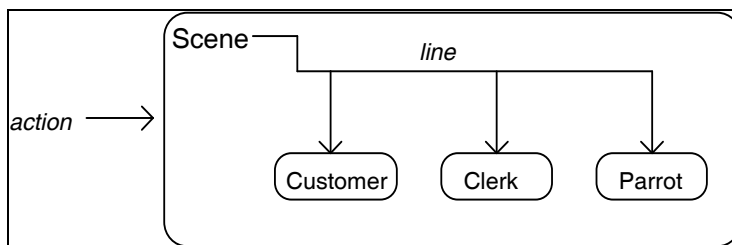
```
% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply() # Animal.reply, calls Cat.speak
meow
>>> data = Hacker() # Animal.reply, calls Primate.speak
>>> data.reply()
Hello world!
```



*A Zoo Hierarchy*

8. *The Dead Parrot Skit: (If this example was skipped in class)* Consider the object embedding structure captured in the figure below. Code a set of Python classes to implement this structure with composition. Code your `Scene` object to define an `action` method, and embed instances of `Customer`, `Clerk`, and `Parrot` classes—all three of which should define a `line` method which prints a unique message. The embedded objects may either inherit from a common superclass that defines `line` and simply provide message text, or define `line` themselves. In the end, your classes should operate like this:

```
% python
>>> import parrot
>>> parrot.Scene().action() # activate nested objects
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```



*A Scene Composite*



## Lab 7: Exceptions and built-in tools

[Go to solutions](#)

[Go to solution files](#)

1. *try/except*. Write a function called `oops` that explicitly raises an `IndexError` exception when called. Then write another function that calls `oops` inside a `try/except` statement to catch the error. What happens if you change `oops` to raise `KeyError` instead of `IndexError`? Where do the names `KeyError` and `IndexError` come from? (Hint: recall that all unqualified names come from one of four scopes, by the LEGB rule.)
2. *Exception objects and lists*. Change the `oops` function you just wrote to raise an exception you define yourself, called `MyError`, and pass an extra data item along with the exception. You may identify your exception with either a string or a class. Then, extend the `try` statement in the catcher function to catch this exception and its data in addition to `IndexError`, and print the extra data item. Finally, if you used a string for your exception, go back and change it to be a class instance; what now comes back as the extra data to the handler?
3. *Error handling*. Write a function called `safe(func, *args)` that runs any function using `apply`, catches any exception raised while the function runs, and prints the exception using the `exc_info()` call in the `sys` module. Then, use your `safe` function to run the `oops` function you wrote in Exercises 1 and/or 2. Put `safe` in a module file called `tools.py`, and pass it the `oops` function interactively. What sort of error messages do you get? Finally, expand `safe` to also print a Python stack trace when an error occurs by calling the built-in `print_exc()` function in the standard `traceback` module (use `exc_info()[2]`, and see the Python library reference manual for details).
4. *Error handling: the Pdb debugger*. Run the “oops” function from (1) or (2) under the `pdb` debugger interactively. Import `pdb` and “oops”, run a call string, and type “c” (continue) commands till the error occurs. Where are you when the debugger stops? Type a “where” command to find out. Now, type “quit” and rerun the program: set a break point on the “oops” function, and single-step up to the error. Experiment with “up” and “down” commands—they climb and descend the Python call stack. See the library manual for more details on `pdb` debugging, or use the point-and-click debugger interface in the IDLE GUI.
5. *Built-in tools*. Study the tables of built-in tools in the Python library manual: *functions*, *exceptions*, *modules*, and special *attribute* names. Experiment interactively with some of the built-in functions we didn’t cover in the lectures yet (e.g., use “`raw_input`” to prompt for an input string, call “`type`” to inspect the type of objects, and so on). Python’s library reference manual provides the most up-to-date listing of available tools, but you may need to hunt for some of them the first time around.
6. *The sys module*. The “`sys.path`” list is initialized from the `PYTHONPATH` environment variable. Is it possible to change it from within Python? If so, does it effect where Python looks for module files? Run some interactive tests to find out.

## Lab 8: System interfaces and GUIs

[Go to solutions](#)

[Go to solution files](#)

1. *Shell tools.* (1) Test the *pack/unpack* example scripts: run *pack* to pack files, and *unpack* to unpack in another directory. (2) Now, modify *pack* and *unpack* to package their functionality as *functions*, instead of top-level code. For instance, add a *'pack\_files'* function in the *pack* script, which encloses the packing logic, and accepts two arguments: a list of input file-names and an output file-name. Wrap the *unpack* logic in a function that takes an input file name as its argument. (3) Finally, write code at the bottom of your new *pack/unpack* files which calls the new functions only when the file is run as a script. Test your new files by running them as scripts, and then by importing them and calling their functions interactively.
2. *Shell tools.* Set up a directory structure, and run the regression tester (example “*regtest.py*”) on some of the scripts you’ve developed so far.
3. *Basic GUIs.* Write a simple script that creates a GUI with three buttons—‘curly’, ‘moe’, and ‘larry’, each of which prints a different message on the ‘stdout’ stream. Use simple Tkinter class calls to build the GUI, but don’t code your GUI as a new class. Run this file as a script, from the system shell’s command line.
4. *Frames.* Now, rewrite your 3-button GUI from (3) as a subclass of the *Frame* container class, and test it again.
5. *Extending GUIs.* Extend the 3-button GUI class from (4) by subclassing it, to include a *Label*. Pack your *Label* before the superclass’s widgets; then try packing it after. What happens?
6. *Attaching GUIs.* Extend the 3-button GUI class from (4) to include a *Label*, by attaching it to an enclosing frame. You can code the enclosing *Frame* as another class, or by using simple widget creation calls. Try attaching the original class instance both before and after the label in the containing *Frame*; what happens this time?
7. *File processing: readline.* Write a script called “*cat.py*”, which defines a function called “*lister*”. The script should take 1 *command-line* argument—a text file name—and pass it as a function argument to “*lister*” only when the file is run as a script. The

“lister” function should read a text file line-by-line with the “*readline*” method, and print each line in upper-case, with a line number prefix before each output line. Use the ‘string’ module to do the case conversion. Test your script file by running it on itself: “python cat.py cat.py”. Then import and test lister interactively: “cat.lister(‘cat.py’)”.

8. *File processing: readlines, read.* Change your “lister” function from (7) to use the “*readlines*” file method instead of “*readline*”, and retest. Then change “lister” to read the file all at once with “*read*”, and detect line boundaries manually. You may either scan for the end-of-line characters yourself by indexing or slicing, use the “string.index/find” functions to find each one, or call “string.split” (see the library manual for more details). Retest with this “read” version. Which of the approaches for finding end-of-lines is easiest? Which may be fastest?
9. *Environment variables.* Change your ‘USER’ environment variable, by assigning to “os.environ”, and spawn another Python program that prints the value of ‘USER’ (using fork, system, or popen). Is the modified value exported to the spawned process? Try changing a different environment variable and repeating the test.
10. *File globbing.* Write a script which accepts a directory name as a command-line argument, and prints a listing of all the files in that directory. Then enhance your script to also accept a search-pattern string on the command line, and use it to display matching file names in the directory.
11. *Timing programs.* Try changing the call *order* in “timeinline.py” to see if it has any effect: in you test code, call inline2 before inline1. Any effect? (Hint: might it depend on your ‘malloc’?) Then, change the example to allow the number of pushes and pops to be passed in to the “test” function instead of using global variables, and test the inline and inline2 functions *interactively* as follows:

```
>>> from testinline import *
>>> test(1000, 1000, inline1)
[results]
>>> test(1000, 1000, inline2)
[results]
```

Any impact on the results? (Hint: the only way it could, is if you’re also measuring the time it takes to compile the scripts.). See also the CD’s Extras\Misc\timerseqs.py

## Lab 9: Persistence

[Go to solutions](#)

[Go to solution files](#)

1. *Shelve basics.* (a) Write a *function* which creates a persistent shelve of lists, from a dictionary-of-lists passed in as an argument. (b) Then write a *script* file that accepts a shelve file name on the command line, and dumps the shelve to the screen, as “key => value” lines. (c) Now test both components: from the *interactive* command line, import and call the first function to load the shelve from a dictionary-of-lists you create, then change one of its entries interactively by opening the shelve yourself. Exit the interactive session, and run the script (b) to dump the updated shelve.
2. *Shelve processing scripts.* Write a script that updates (changes) all the entries in the shelve you created in exercise (1). Open and scan the shelve file, deleting the first node in each of the stored lists, and storing the lists back to the shelve as you go. Use your shelve dumping script from exercise (1) to verify your changes. Can you update a shelve by something like: ‘del file[key][0]’?
3. *Storing class instances.* Write a class called “*Student*” in a module called “*student.py*”. “*Student*” should model you: it’s constructor should create instance members for ‘name’, ‘age’, ‘job’, and so on, and define a method called ‘info’ which lists the member values in ‘key=>value’ form to stdout. Now, make an instance of this class interactively, and store it in a new shelve file. Exit Python, reopen your shelve, fetch back the object you stored, and inspect it by calling its ‘info’ method. Then, store a new “*Student*” instance object in the shelve, with fictitious member details. Did you have to import the “*student*” module in order to use “*Student*” instances fetched from the shelve? Why?
4. *Changing classes.* Change the “*Student*” class you wrote in (3), such that its ‘info’ method now lists members in ‘key...value’ form to stdout. Reopen your shelve, and iterate over all the stored Students (use the dictionary ‘keys’ method to get an index list), calling the ‘info’ method for each. What happens if you repeat this experiment after renaming the “*student.py*” module that contains the “*Student*” class?

## Lab 10: Text processing and the Internet

[Go to solutions](#)

[Go to solution files](#)

1. *String processing.* Write a function which returns the *reverse* of a string passed in as an argument. You may use iteration and concatenation, or any other approach you think works best. (Suggestion: what about using the built-in `list(sequence)` conversion function to leverage the list ‘reverse’ method for this job?) Can you generalize your function to reverse *any* type of sequence? What type should the result be?
2. *String conversions.* Write a function which *converts* a string of digit characters (‘0’ through ‘9’) to an integer object with the corresponding numeric value. For instance, “123” should create the integer 123. Hint: use “`ord(C) - ord(‘0’)`” to convert each digit. Experiment with these tools interactively. Now, how would you go about doing the inverse—converting an integer to a string? Hint ‘`X % 10`’ returns the remainder of division by 10, and ‘`X / 10`’ returns the integer part, but built-in tools can help here too.
3. *String splits.* Write a text file containing columns of numbers, and run the “*summer.py*” example to tally its columns. Now, write a file of numeric *expressions* without embedded blanks (e.g. ‘`2+1`’), and try running *summer* on that. What happens? Why? Finally, change ‘*summer.py*’ to split lines on “\$” column separators, change your text file to separate columns by “\$”, and rerun the tests. Can your expressions contain blanks and tabs now?
4. *Regular expressions.* Run the “*cheader1.py*” example on a C file (e.g., ‘.c’ or ‘.h’) of your choosing. For instance, you might run it over a Python include file in the Python source tree. Does it handle both ‘<’ and quoted file names in `#include` directives? How about `#define` macros with arguments and continuation lines? As is, the script doesn’t support transitively included files (it doesn’t also scan files referenced by a `#include` line). How would you go about adding this too? (Hint: you’ll probably need to make a recursive call, and keep track of files you’ve scanned so far; see “Tools/scripts/h2py.py” in the Python source tree.)
5. *The FTP module.* Write a program that fetches the file “sousa.au” from site “ftp.python.org”, in directory “pub/python/misc”. Use anonymous FTP, and binary transfer mode, and store the file on your local machine under name “mysousa.au”. If you have an audio filter, you can play this file on your machine after your script finishes. How would you go about playing it automatically from within the FTP script? (Hint: a DOS “start” command can be run by `os.system`, and spawned web browsers may help too.)

## Lab 11: Extending Python in C/C++

[Go to solutions](#)

[Go to solution files](#)

*Note:* most of the exercises in the next two labs (extending and embedding Python in C) are complex, and require access to a C compiler and build environment. They're also almost never worked on during the course itself. Feel free to use these last two lab sessions to go back and finish prior lab exercises you've skipped, especially if you'll be using Python in stand-alone mode (i.e., without coding any C integrations of your own). These exercises can also be tackled after the course is over, at your convenience. Be sure to at least have a look at the last two exercises in these two labs, though; they don't require C compilation.

1. *Extension modules.* Add another function to the extension module example, which calls the C library's **'putenv'** function. It should accept a single string, and return the **'None'** object, or a NULL pointer to report errors. Install the modified module in Python using *static* binding, and test your new extension function from the Python command line. How does your **putenv** differ from the built-in **'os.environ'** and **'os.putenv'**? (Hint: does your function update **'os'**?).
2. *Dynamic binding.* If your platform supports dynamic load libraries, bind the modified **'environ'** module from (1) to Python using dynamic binding: remove the Modules/Setup entry and remake Python, compile **'environ.c'** into a shared library, and put the result in a directory names on \$PYTHONPATH. Test the binding by importing the module and calling your **'putenv'** function interactively.
3. *Extension types.* Code a **'print'** handler function in the C stack extension type (stacktyp.c), which simply prints a "not implemented" message whenever it is called. Compile your extension type's module and bind it to Python, either statically or dynamically, and test the handler interactively.
4. *Extension types.* Code the **'length'** handler function, and the **'push'** and **'pop'** instance methods, and rebuild Python with these extensions. Test the new method interactively: import the stack module, make a Stack instance, and call the instance's push, pop, length, and item (indexing) functionality. Does the type respond to *'for'* loop iteration as is?
5. *Extension types.* Add a **'set\_item'** (index assignment) handler to the stack type. Code a handler function (it may simply print a message), register your function in the type descriptors, and rebuild. Test your new handler interactively, by assignments (**"x[i] = value"**).
6. *Extension wrappers.* Expand the extension type wrapper presented in the lecture to include a wrapper for the new **'set\_item'** method you implemented in exercise (5).
7. *Inspecting objects.* Import your extension module from exercise (1), and call the built-in **'dir'** function on the module (**'dir(M)'**). What do you see? Now, inspect the

module's `'__dict__'` attribute. How does this differ? Next, import your stack *type*, make an instance, and inspect the instance's `'__members__'` and `'__methods__'` attributes. What happens when you do a `'dir'` on a type instance? How do these findings compare to Python modules, classes and class instances? Write a simple class in a module file and run interactive tests to find out.

8. *Inspecting objects.* Repeat all of the inspections performed in exercise (7) on the built-in **'regex'** regular expression module: import and inspect the 'regex' module itself, then make a compiled expression object (by running an assignment like `"x = regex.compile('hello')"`) and inspect its members and methods. Regex is a pre-coded C extension module/type which is a standard part of Python; compiling expressions makes a type instance object. The module's source code is available in the Python source tree's 'Module' directory. Find and study the 'regexmodule.c' source file.
9. *Built-in type examples.* Study the implementations of built-in lists, integers, and dictionaries, in the "Object" directory of the Python source tree. See the source files 'listobject.c', 'intobject.c', and 'mappingobject.c'. Can you locate the major type components we studied in the lecture? Why isn't there a constructor module? (Hint: how do you create lists, integers, and dictionaries?).

## Lab 12: Embedding Python in C/C++

[Go to solutions](#)

[Go to solution files](#)

1. *Running code strings.* Write a C 'main' program which runs a Python code string, to display the value of the "sys.path" module search path in Python. You may implement the embedding in one of three ways:
  - By running simple strings like `"import sys; print sys.path"` (PyRun\_SimpleString)
  - By running `"print path"` in module `"sys"` (PyImport\_ImportModule, PyRun\_String)
  - By using the extended API functions (Run\_Codestr)

Build an executable by linking your program with Python libraries and object files, and run it from the system shell's command line. If you choose to use the extended API tools, also compile with the `"ppembed*.c"` files from the CD-ROM shipped with the book Programming Python 2<sup>nd</sup> Edition. Experiment with the other modes above as time allows.

2. *Calling objects.* Write a C ‘main’ program which calls the “os.getcwd()” function, and prints the returned string to stdout. “os.getcwd()” takes no arguments, and returns the name of the current directory as a Python string (see p.788). Alternative approaches:

- Call getcwd manually (PyImport\_ImportModule, PyEval\_CallObject)
- Call getcwd with an extended API function (Run\_Function)

Build your program and test it from the system command line.

3. *Dynamic debugging.* If you used the extended API functions in exercises (1) and/or (2), experiment with dynamic debugging of embedded code, by setting the variable “PY\_DEBUG = 1” in your C program, before calling the API functions. Where are you in your embedded code, when control stops in the pdb debugger? Would dynamic reloading of the embedded code help in these programs?
4. *Registration.* In the registration example, suppose we wanted to change the event handlers without stopping the C program. Is there any way to force C to reload the registered function’s module at run-time? (Hint: how does C know the name of the enclosing module? And how would reloading the module update the object?) Does this sound similar to the “from” gotcha for “reload” in Python?
5. *Error handling.* Add code to the “objects1err.c” example to print a Python stack traceback (PyErr\_Print) and fetch exception information using the “pyerrors.c” support file whenever API errors occur. Now, change your PYTHONPATH so it doesn’t include the Python module which holds the “klass” class, and rebuild/rerun your executable to trigger an API error. What happens?
6. *Web resources.* If you have access to a web browser, point it to URL “<http://www.python.org>”, Python’s home page. Then click on the ‘Search’ link at the top to invoke the Python locator system. Then search the newsgroup/ mailing list archives for topic “Larry Wall”, to see what you can find out about Perl’s creator from old Python posts.
7. *FTP resources.* If you have access to a FTP client, use anonymous FTP to manually fetch the file “sousa.au” from URL “<ftp://ftp.python.org/pub/python/misc>” (doing it from Python was a prior exercise). This is an audio file—the Monty Python television show’s theme song—which you can play on your machine if you have an audio filter program. You may also be able to fetch and play the file with your web browser.



## Selected exercise solutions

This section lists solutions to some of the earlier lab's exercises, taken from the book *Learning Python*. Feel free to consult these answers if you get stuck, and also for pointers on alternative solutions.

Also see the [solution file directories](#) for answers to additional lab sessions.

### Lab 1: Using the Interpreter

1. *Interaction.* Assuming your Python is configured properly, you should participate in an interaction that looks something like the following. You can run this any way you like: in IDLE, from a shell prompt, and so on:

```
% python
...copyright information lines...
>>> "Hello World!"
'Hello World!'
>>> # Ctrl-D, Ctrl-Z, or window close to exit
```

2. *Programs.* Here's what your code (i.e., module) file and shell interactions should look like; again, feel free to run this other ways—by clicking its icon, by IDLE's Edit/RunScript menu option, and so on:

```
File: module1.py
print 'Hello module world!'

% python module1.py
Hello module world!
```

3. *Modules.* The following interaction listing illustrates running a module file by importing it. Remember that you need to reload it to run again without stopping and restarting the interpreter. The bit about moving the file to a different directory and importing it again is a trick question: if Python generates a module1.pyc file in the original directory, it uses that when you import the module, even if the source code file (.py) has been moved to a directory not on Python's search path. The .pyc file is written automatically if Python has access to the source file's directory and contains the compiled bytecode version of a module. We look at how this works again in the modules unit.

```
% python
>>> import module1
Hello module world!
>>>
```

4. *Scripts.* Assuming your platform supports the `#!` trick, your solution will look like the following (though your `#!` line may need to list another path on your machine):

```

File: module1.py
#!/usr/local/bin/python (or #!/usr/bin/env python)
print 'Hello module world!'

% chmod +x module1.py

% module1.py
Hello module world!
```

5. *Errors.* The interaction below demonstrates the sort of error messages you get if you complete this exercise. Really, you're triggering Python exceptions; the default exception handling behavior terminates the running Python program and prints an error message and stack trace on the screen. The stack trace shows where you were at in a program when the exception occurred (it's not very interesting here, since the exceptions occur at the top level of the interactive prompt; no function calls were in progress). In the exceptions unit, you will see that you can catch exceptions using "try" statements and process them arbitrarily; you'll also see that Python includes a full-blown source-code debugger for special error detection requirements. For now, notice that Python gives meaningful messages when programming errors occur (instead of crashing silently):

```

% python
>>> 1 / 0
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo
>>>
>>> x
Traceback (innermost last):
 File "<stdin>", line 1, in ?
NameError: x
```

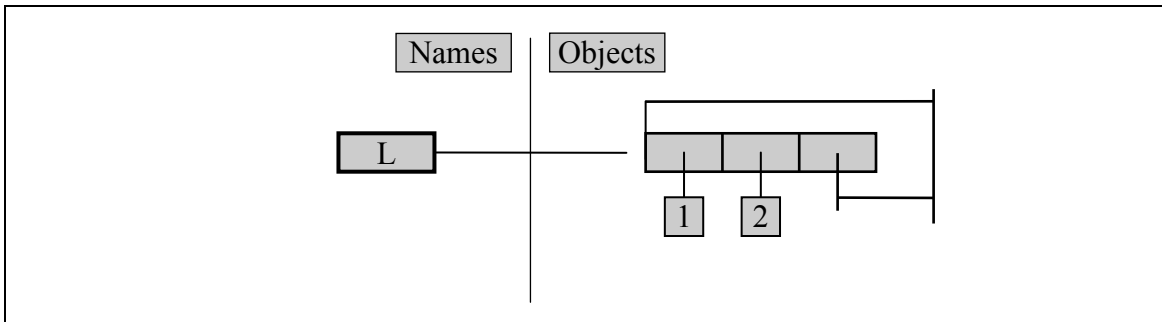
6. *Breaks.* When you type this code:

```

L = [1, 2]
L.append(L)
```

you create a cyclic data-structure in Python. In Python releases before Version 1.5.1, the Python printer wasn't smart enough to detect cycles in objects, and it would print an unending stream of `[1, 2, [1, 2, [1, 2, [1, 2, ...]` and so on, until you hit the break key combination on your machine (which, technically, raises a keyboard-interrupt exception that prints a default message at the top level unless you intercept it in a program). Beginning with Python Version 1.5.1, the printer is clever enough to detect cycles and prints `[...]` instead to let you know.

The reason for the cycle is subtle and requires information you'll gain in the next unit. But in short, assignment in Python always generates references to objects (which you can think of as implicitly followed pointers). When you run the first assignment above, the name `L` becomes a named reference to a two-item list object. Now, Python lists are really arrays of object references, with an `append` method that changes the array in-place by tacking on another object reference. Here, the `append` call adds a reference to the front of `L` at the end of `L`, which leads to the cycle illustrated in the figure below. Believe it or not, cyclic data structures can sometimes be useful (but maybe not when printed!). Today, Python can also reclaim (garbage collect) such objects cyclic automatically.



A cyclic list

## Lab 2: Types and Operators

1. *The basics.* Here are the sort of results you should get, along with a few comments about their meaning. Note that “;” is used in a few of these to squeeze more than one statement on a single line; as we’ll learn in the next unit, the `;` is a statement separator.

### Numbers

```
>>> 2 ** 16 # 2 raised to the power 16
65536
>>> 2 / 5, 2 / 5.0 # integer / truncates, float / doesn't
(0, 0.40000000000000002)
```

### Strings

```
>>> "spam" + "eggs" # concatenation
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5 # repetition
'hamhamhamhamham'
>>> S[:0] # an empty slice at the front--[0:0]
''
>>> "green %s and %s" % ("eggs", S) # formatting
'green eggs and ham'
```

### Tuples

```
>>> ('x',)[0] # indexing a single-item tuple
'x'
>>> ('x', 'y')[1] # indexing a 2-item tuple
'y'
```

### Lists

```
>>> L = [1,2,3] + [4,5,6] # list operations
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
```

```

[3, 4]
>>> [L[2], L[3]] # fetch from offsets, store in a list
[3, 4]
>>> L.reverse(); L # method: reverse list in-place
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L # method: sort list in-place
[1, 2, 3, 4, 5, 6]
>>> L.index(4) # method: offset of first 4 (search)
3

Dictionaries

>>> {'a':1, 'b':2}['b'] # index a dictionary by key
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0 # create a new entry
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4 # a tuple used as a key (immutable)
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}
>>> D.keys(), D.values(), D.has_key((1,2,3)) # methods
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], 1)

Empties

>>> [], [], (), {}, None # lots of nothings: empty objects
([], [], [], (), {}, None)

```

## 2. Indexing and slicing.

Indexing out-of-bounds (e.g., `L[4]`) raises an error; Python always checks to make sure that all offsets are within the bounds of a sequence (unlike C, where out-of-bound indexes will happily crash your system).

On the other hand, slicing out of bounds (e.g., `L[-1000:100]`) works, because Python scales out-of-bounds slices so that they always fit (they're set to zero and the sequence length, if required).

Extracting a sequence in reverse—with the lower bound > the higher bound (e.g., `L[3:1]`)—doesn't really work. You get back an empty slice (`[]`), because Python scales the slice limits to make sure that the lower bound is always less than or equal to the upper bound (e.g., `L[3:1]` is scaled to `L[3:3]`, the empty insertion point at offset 3). Python slices are always extracted from left to right, even if you use negative indexes (they are first converted to positive indexes by adding the length).

```

>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
 File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]

```

## 3. Indexing, slicing, and del. Your interaction with the interpreter should look something like that listed below. Note that assigning an empty list to an offset stores an empty list object there, but assigning an

empty list to a slice deletes the slice. Slice assignment expects another sequence, or you'll get a type error; it assigns items *inside* the sequence assigned, not the sequence itself:

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

4. *Tuple assignment.* The values of *X* and *Y* are swapped. When tuples appear on the left and right of an assignment symbol (=), Python assigns objects on the right to targets on the left, according to their positions. This is probably easiest to understand by noting that targets on the left aren't a real tuple, even though they look like one; they are simply a set of independent assignment targets. The items on the right are a tuple, which get unpacked during the assignment (the tuple provides the temporary assignment needed to achieve the swap effect).

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'
```

5. *Dictionary keys.* Any immutable object can be used as a dictionary key—integers, tuples, strings, and so on. This really is a dictionary, even though some of its keys look like integer offsets. Mixed type keys work fine too.

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Dictionary indexing.* Indexing a nonexistent key (`D['d']`) raises an error; assigning to a nonexistent key (`D['d']='spam'`) creates a new dictionary entry. On the other hand, out-of-bounds indexing for lists raises an error too, but so do out-of-bounds assignments. Variable names work like dictionary keys: they must have already been assigned when referenced, but are created when first assigned. In fact, variable names can be processed as dictionary keys if you wish (they're made visible in module namespace or stack-frame dictionaries).

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
```

```

Traceback (innermost last):
 File "<stdin>", line 1, in ?
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0,1]
>>> L[2]
Traceback (innermost last):
 File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
 File "<stdin>", line 1, in ?
IndexError: list assignment index out of range

```

### 7. *Generic operations.*

Question answers: The + operator doesn't work on different/mixed types (e.g., string + list, list + tuple).

+ doesn't work for dictionaries, because they aren't sequences.

The append method works only for lists, not strings, and keys works only on dictionaries. append assumes its target is mutable, since it's an in-place extension; strings are immutable.

Slicing and concatenation always return a new object of the same type as the objects processed.

```

>>> "x" + 1
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> {} + {}
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
 File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> {}.keys()
[]
>>> [].keys()
Traceback (innermost last):
 File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> [][::]
[]
>>> ""[::]
''

```

8. *String indexing.* Since strings are collections of one-character strings, every time you index a string, you get back a string, which can be indexed again. `S[0][0][0][0][0]` just keeps indexing the first character over and over. This generally doesn't work for lists (lists can hold arbitrary objects), unless the list contains strings.

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. *Immutable types.* Either of the solutions below work. Index assignment doesn't, because strings are immutable.

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

10. *Nesting.* Your mileage will vary.

```
>>> me = {'name':('mark', 'e', 'lutz'), 'age':'?', 'job':'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'lutz'
```

11. *Files.* Here's one way to create and read back a text file in Python (ls is a Unix command; use dir on Windows):

```
File: maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')
file.close()
or: open().write()
close not always needed

File: reader.py
file = open('myfile.txt', 'r')
print file.read()
or: print open().read()

% python maker.py
% python reader.py
Hello file world!

% ls -l myfile.txt
-rwxrwxrwa 1 0 0 19 Apr 13 16:33 myfile.txt
```

12. *The dir function:* Here's what you get for lists; dictionaries do the same (but with different method names). Note that the dir result expanded in Python 2.2—you'll see a large set of additional underscore names that implement expression operators, and support the subclassing we'll meet in the classes unit. The `__methods__` attribute disappeared in 2.2 as well, because it wasn't consistently implemented—use `dir` to to fetch attribute lists today instead:

```
>>> [].__methods__
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort',...]
>>> dir([])
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort',...]
```

## Lab 3: Basic Statements

1. *Coding basic loops.* If you work through this exercise, you'll wind up with code that looks something like the following:

```
>>> S = 'spam'
>>> for c in S:
... print ord(c)
...
115
112
97
109

>>> x = 0
>>> for c in S: x = x + ord(c) # or: x += ord(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> map(ord, S)
[115, 112, 97, 109]
```

2. *Backslash characters.* The example prints the bell character (`\a`) 50 times; assuming your machine can handle it, you'll get a series of beeps (or one long tone, if your machine is fast enough). Hey—I warned you.
3. *Sorting dictionaries.* Here's one way to work through this exercise; see lecture 3 if this doesn't make sense. Remember, you really do have to split the `keys` and `sort` calls up like this, because `sort` returns `None`.

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = D.keys()
>>> keys.sort()
>>> for key in keys:
... print key, '=>', D[key]
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
```



```
g => 7
```

4. *Program logic alternatives.* Here's how we coded the solutions; your results may vary a bit. This exercise is mostly just designed to get you playing with code alternatives, so anything reasonable gets full credit:

- a) First, rewrite this code with a while loop else, to eliminate the found flag and final if statement.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
 if 2 ** X == L[i]:
 print 'at index', i
 break
 i = i+1
else:
 print X, 'not found'
```

- b) Next, rewrite the example to use a for loop with an else, to eliminate the explicit list indexing logic. Hint: to get the index of an item, use the list *index* method (`list.index(X)` returns the offset of the first X).

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
 if (2 ** X) == p:
 print (2 ** X), 'was found at', L.index(p)
 break
else:
 print X, 'not found'
```

- c) Now, remove the loop completely by rewriting the examples with a simple *in* operator membership expression (see lecture 2 for more details, or type this: `2 in [1, 2, 3]`).

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
 print (2 ** X), 'was found at', L.index(2 ** X)
else:
 print X, 'not found'
```

- d) Finally, use a for loop and the list *append* method to generate the powers-of-2 list (L), instead of hard-coding a list constant.

```
X = 5
L = []
for i in range(7): L.append(2 ** i)
print L

if (2 ** X) in L:
 print (2 ** X), 'was found at', L.index(2 ** X)
else:
 print X, 'not found'
```

- e) Deeper thoughts: (2) As we saw in exercise 1, Python also provides a `map(function, list)` built-in tool which could be used to generate the powers-of-2 list too. Consider this a preview of the next lecture.

```
X = 5
L = map(lambda x: 2**x, range(7))
print L

if (2 ** X) in L:
 print (2 ** X), 'was found at', L.index(2 ** X)
else:
 print X, 'not found'
```

## Lab 4: Functions

1. *Basics.* There's not much to this one, but notice that your using the big “P” word—`print` (and hence your function) is technically a *polymorphic* operation, which does the right thing for each type of object:

```
% python
>>> def func(x): print x
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}
```

2. *Arguments.* Here's what one solution looks like. Remember that you have to use `print` to see results in the test calls, because a file isn't the same as code typed interactively; Python doesn't normally echo the results of expression statements in files.

```
File: mod.py

def adder(x, y):
 return x + y

print adder(2, 3)
print adder('spam', 'eggs')
print adder(['a', 'b'], ['c', 'd'])

% python mod.py
5
spameggs
['a', 'b', 'c', 'd']
```

3. *Varargs.* Two alternative `adder` functions are shown in the following code. The hard part here is figuring out how to initialize an accumulator to an empty value of whatever type is passed in. In the

first solution, we use manual type testing to look for an integer and an empty slice of the first argument (assumed to be a sequence) otherwise. In the second solution, we just use the first argument to initialize and scan items 2 and beyond, much like one of the `max` function coded in class.

The second solution is better (and frankly, comes from students in a Python course I taught, who were frustrated with trying to understand the first solution). Both of these assume all arguments are the same type and neither works on dictionaries; as we saw a *priore* unit, `+` doesn't work on mixed types or dictionaries. We could add a type test and special code to add dictionaries too, but that's extra credit.

```

File adders.py

def adder1(*args):
 print 'adder1',
 if type(args[0]) == type(0): # integer?
 sum = 0 # init to zero
 else: # else sequence:
 sum = args[0][:0] # use empty slice of arg1
 for arg in args:
 sum = sum + arg
 return sum

def adder2(*args):
 print 'adder2',
 sum = args[0] # init to arg1
 for next in args[1:]:
 sum = sum + next # add items 2..N
 return sum

for func in (adder1, adder2):
 print func(2, 3, 4)
 print func('spam', 'eggs', 'toast')
 print func(['a', 'b'], ['c', 'd'], ['e', 'f'])

% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']

```

4. **Keywords.** Here is our solution to the first part of this one. To iterate over keyword arguments, use a `**args` form in the function header and use a loop like: `for x in args.keys(): use args[x]`.

```

File: mod.py

def adder(good=1, bad=2, ugly=3):
 return good + bad + ugly

print adder()
print adder(5)
print adder(5, 6)
print adder(5, 6, 7)
print adder(ugly=7, good=6, bad=5)

% python mod.py
6
10
14
18
18

```

5. and 6. Here are our solutions to Exercises 5 and 6. These are just coding exercises, though, because Guido has already made them superfluous—Python 1.5 added new dictionary methods, to do things like copying and adding (merging) dictionaries: `D.copy()`, and `D1.update(D2)`. See Python's library manual or the Python Pocket Reference for more details. `X[:]` doesn't work for dictionaries, since they're not sequences. Also remember that if we assign (`e = d`) rather than copy, we generate a reference to a *shared* dictionary object; changing `d` changes `e` too.

```

File: dicts.py

def copyDict(old):
 new = {}
 for key in old.keys():
 new[key] = old[key]
 return new

def addDict(d1, d2):
 new = {}
 for key in d1.keys():
 new[key] = d1[key]
 for key in d2.keys():
 new[key] = d2[key]
 return new

% python
>>> from dicts import *
>>> d = {1:1, 2:2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1:1}
>>> y = {2:2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}

```

7. *Argument matching.* Here is the sort of interaction you should get, along with comments that explain the matching that goes on:

```

def f1(a, b): print a, b # normal args
def f2(a, *b): print a, b # positional varargs
def f3(a, **b): print a, b # keyword varargs
def f4(a, *b, **c): print a, b, c # mixed modes
def f5(a, b=2, c=3): print a, b, c # defaults
def f6(a, b=2, *c): print a, b, c # defaults + positional varargs

% python
>>> f1(1, 2) # matched by position (order matters)
1 2
>>> f1(b=2, a=1) # matched by name (order doesn't matter)
1 2
>>> f2(1, 2, 3) # extra positionals collected in a tuple

```

```

1 (2, 3)

>>> f3(1, x=2, y=3) # extra keywords collected in a dictionary
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3) # extra of both kinds
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1) # both defaults kick in
1 2 3
>>> f5(1, 4) # only one default used
1 4 3

>>> f6(1) # one argument: matches "a"
1 2 ()
>>> f6(1, 3, 4) # extra positional collected
1 3 (4,)

```

8. *List comprehensions.* Here is the sort of code you should write; we may have a preference, but we're not telling.

```

>>> values = [2, 4, 9, 16, 25]
>>> import math
>>>
>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
>>>
>>> map(math.sqrt, values)
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
>>>
>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

```

## Lab 5: Modules

1. *Basics, import.* This one is simpler than you may think. When you're done, your file and interaction should look close to the following code; remember that Python can read a whole file into a string or lines list, and the `len` built-in returns the length of strings and lists:

```

File: mymod.py

def countLines(name):
 file = open(name, 'r')
 return len(file.readlines())

def countChars(name):
 return len(open(name, 'r').read())

def test(name):
 return countLines(name), countChars(name)
or pass file object
or return a dictionary

```

```
% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

On Unix, you can verify your output with a `wc` command; on Windows, right-click on your file to view its properties. (But note that your script may report fewer characters than Windows does—for portability, Python converts Windows “`\r\n`” line-end markers to “`\n`”, thereby dropping one byte (character) per line. To match byte counts with Windows exactly, you have to open in binary mode—“`rb`”, or add back the number of lines.)

Incidentally, to do the “ambitious” part (passing in a file object, so you only open the file once), you’ll probably need to use the `seek` method of the built-in file object. We didn’t cover it in the text, but it works just like C’s `fseek` call (and calls it behind the scenes): `seek` resets the current position in the file to an offset passed in. After a `seek`, future input/output operations are relative to the new position. To rewind to the start of a file without closing and reopening, call `file.seek(0)`; the file read methods all pick up at the current position in the file, so you need to rewind to reread. Here’s what this tweak would look like:

```
File: mymod2.py

def countLines(file):
 file.seek(0) # rewind to start of file
 return len(file.readlines())

def countChars(file):
 file.seek(0) # ditto (rewind if needed)
 return len(file.read())

def test(name):
 file = open(name, 'r') # pass file object
 return countLines(file), countChars(file) # only open file once

>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. `from/from*`. Here’s the `from*` bit; replace `*` with `countChars` to do the rest:

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
291
```

3. `__main__`. If you code it properly, it works in either mode (program run or module import):

```
File: mymod.py

def countLines(name):
 file = open(name, 'r')
 return len(file.readlines())

def countChars(name):
 return len(open(name, 'r').read())

def test(name):
 return countLines(name), countChars(name) # or pass file object
 # or return a dictionary

if __name__ == '__main__':
 print test('mymod.py')
```

```
% python mymod.py
(13, 346)
```

4. *Nested imports.* Our solution for this appears below:

```
File: myclient.py

from mymod import countLines, countChars
print countLines('mymod.py'), countChars('mymod.py')

% python myclient.py
13 346
```

As for the rest of this one: `mymod`'s functions are accessible (that is, importable) from the top level of `myclient`, since `from` simply assigns to names in the importer (it works almost as though `mymod`'s defs appeared in `myclient`). For example, another file can say this:

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

If `myclient` used `import` instead of `from`, you'd need to use a path to get to the functions in `mymod` through `myclient`:

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

In general, you can define *collector* modules that import all the names from other modules, so they're available in a single convenience module. Using the following code, you wind up with three different copies of `somename`: `mod1.somename`, `collector.somename`, and `__main__.somename`; all three share the same integer object initially, and only the name `somename` exists at the interactive prompt as is:

```
File: mod1.py

somename = 42

File: collector.py

from mod1 import * # collect lots of names here
from mod2 import * # from assigns to my names
from mod3 import *

>>> from collector import somename
```

5. *Package imports.* For this, we put the `mymod.py` solution file listed for exercise 3 into a directory package. The following is what we did to set up the directory and its required `__init__.py` file in a Windows console interface; you'll need to interpolate for other platforms (e.g., use `mv` and `vi` instead of `move` and `edit`). This works in any directory (we just happened to run our commands in Python's install directory), and you can do some of this from a file explorer GUI too.

When we were done, we had a `mypkg` subdirectory, which contained files `__init__.py` and `mymod.py`. You need an `__init__.py` in the `mypkg` directory, but not in its parent; `mypkg` is

located in the home directory component of the module search path. Notice how a `print` statement we coded in the directory's initialization file only fires the first time it is imported, not the second:

```
C:\python22> mkdir mypkg
C:\Python22> move mymod.py mypkg\mymod.py
C:\Python22> edit mypkg__init__.py
...coded a print statement...

C:\Python22> python
>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg\mymod.py')
346
```

6. *Reload*. This exercise just asks you to experiment with changing the `changer.py` example in the book, so there's not much for us to show here. If you had some fun with it, give yourself extra points.
7. *Circular imports*. The short story is that importing `recur2` first works, because the recursive import then happens at the import in `recur1`, not at a `from` in `recur2`.

The long story goes like this: importing `recur2` first works, because the recursive import from `recur1` to `recur2` fetches `recur2` as a whole, instead of getting specific names. `recur2` is incomplete when imported from `recur1`, but because it uses `import` instead of `from`, you're safe: Python finds and returns the already created `recur2` module object and continues to run the rest of `recur1` without a glitch. When the `recur2` import resumes, the second `from` finds name `Y` in `recur1` (it's been run completely), so no error is reported. Running a file as a script is not the same as importing it as a module; these cases are the same as running the first `import` or `from` in the script interactively. For instance, running `recur1` as a script is the same as importing `recur2` interactively, since `recur2` is the first module imported in `recur1`. (E-I-E-I-O!)

## Lab 6: Classes

1. *Inheritance*. Here's the solution we coded up for this exercise, along with some interactive tests. The `__add__` overload has to appear only once, in the superclass, since it invokes type-specific `add` methods in subclasses.

```
File: adder.py

class Adder:
 def add(self, x, y):
 print 'not implemented!'
 def __init__(self, start=[]):
 self.data = start
 def __add__(self, other):
 return self.add(self.data, other)

class ListAdder(Adder):
 def add(self, x, y):
 return x + y
```



```

class DictAdder(Adder):
 def add(self, x, y):
 new = {}
 for k in x.keys(): new[k] = x[k]
 for k in y.keys(): new[k] = y[k]
 return new

% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands

```

Notice in the last test that you get an error for expressions where a class instance appears on the right of a +; if you want to fix this, use `__radd__` methods as described in this unit's operator overloading section.

As we suggested, if you are saving a value in the instance anyhow, you might as well rewrite the `add` method to take just one arguments, in the spirit of other examples in this unit:

```

class Adder:
 def __init__(self, start=[]):
 self.data = start
 def __add__(self, other): # pass a single argument
 return self.add(other) # the left side is in self
 def add(self, y):
 print 'not implemented!'

class ListAdder(Adder):
 def add(self, y):
 return self.data + y

class DictAdder(Adder):
 def add(self, y):
 pass # change me to use self.data instead of x

x = ListAdder([1,2,3])
y = x + [4,5,6]
print y # prints [1, 2, 3, 4, 5, 6]

```

Because values are attached to objects rather than passed around, this version is arguably more object-oriented. And once you've gotten to this point, you'll probably see that you could get rid of `add`

altogether, and simply define type-specific `__add__` methods in the two subclasses. They're called exercises for a reason!

2. *Operator overloading.* Here's what we came up with for this one. It uses a few operator overload methods we didn't say much about, but they should be straightforward to understand. Copying the initial value in the constructor is important, because it may be mutable; you don't want to change or have a reference to an object that's possibly shared somewhere outside the class. The `__getattr__` method routes calls to the wrapped list. For hints on an easier way to code this as of Python 2.2, see this unit's section on extending built-in types with subclasses.

```

File: mylist.py

class MyList:
 def __init__(self, start):
 #self.wrapped = start[:] # copy start: no side effects
 self.wrapped = [] # make sure it's a list here
 for x in start: self.wrapped.append(x)
 def __add__(self, other):
 return MyList(self.wrapped + other)
 def __mul__(self, time):
 return MyList(self.wrapped * time)
 def __getitem__(self, offset):
 return self.wrapped[offset]
 def __len__(self):
 return len(self.wrapped)
 def __getslice__(self, low, high):
 return MyList(self.wrapped[low:high])
 def append(self, node):
 self.wrapped.append(node)
 def __getattr__(self, name): # other members: sort/reverse/etc.
 return getattr(self.wrapped, name)
 def __repr__(self):
 return `self.wrapped`

if __name__ == '__main__':
 x = MyList('spam')
 print x
 print x[2]
 print x[1:]
 print x + ['eggs']
 print x * 3
 x.append('a')
 x.sort()
 for c in x: print c,

% python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s

```

3. *Subclassing.* Our solution appears below. Your solution should appear similar.

```

File: mysub.py

from mylist import MyList

class MyListSub(MyList):
 calls = 0 # shared by instances

```

```

def __init__(self, start):
 self.adds = 0 # varies in each instance
 MyList.__init__(self, start)

def __add__(self, other):
 MyListSub.calls = MyListSub.calls + 1 # class-wide counter
 self.adds = self.adds + 1 # per instance counts
 return MyList.__add__(self, other)

def stats(self):
 return self.calls, self.adds # all adds, my adds

if __name__ == '__main__':
 x = MyListSub('spam')
 y = MyListSub('foo')
 print x[2]
 print x[1:]
 print x + ['eggs']
 print x + ['toast']
 print y + ['bar']
 print x.stats()

% python mysub.py
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 'toast']
['f', 'o', 'o', 'bar']
(3, 2)

```

4. *Metaclass methods.* We worked through this exercise as follows. Notice that operators try to fetch attributes through `__getattr__` too; you need to return a value to make them work.

```

>>> class Meta:
... def __getattr__(self, name):
... print 'get', name
... def __setattr__(self, name, value):
... print 'set', name, value
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1:5]
get __len__
Traceback (innermost last):
 File "<stdin>", line 1, in ?
TypeError: call of non-function

```

5. *Set objects.* Here's the sort of interaction you should get; comments explain which methods are called.

```
% python
>>> from setwrapper import Set
>>> x = Set([1,2,3,4]) # runs __init__
>>> y = Set([3,4,5])

>>> x & y # __and__, intersect, then __repr__
Set:[3, 4]
>>> x | y # __or__, union, then __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello") # __init__ removes duplicates
>>> z[0], z[-1] # __getitem__
('h', 'o')

>>> for c in z: print c, # __getitem__
...
h e l l o
>>> len(z), z # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])

>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

Our solution to the multiple-operand extension subclass looks like the class below. It needs only to replace two methods in the original set. The class's documentation string explains how it works:

```
File: multiset.py

from setwrapper import Set

class MultiSet(Set):
 """
 inherits all Set names, but extends intersect
 and union to support multiple operands; note
 that "self" is still the first argument (stored
 in the *args argument now); also note that the
 inherited & and | operators call the new methods
 here with 2 arguments, but processing more than
 2 requires a method call, not an expression:
 """

 def intersect(self, *others):
 res = []
 for x in self:
 for other in others:
 if x not in other: break
 else:
 res.append(x)
 return Set(res)

 def union(*args):
 res = []
 for seq in args:
 for x in seq:
 if not x in res:
 res.append(x)
 return Set(res)
```

Your interaction with the extension will be something along the following lines. Note that you can intersect by using `&` or calling `intersect`, but must call `intersect` for three or more operands; `&` is a binary (two-sided) operator. Also note that we could have called `MultiSet` simply `Set` to make this change more transparent. if we used `setwrapper.Set` to refer to the original within `multiset`:

```
>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])

>>> x & y, x | y # 2 operands
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z) # 3 operands
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]

>>> x.intersect([1,2,3], [2,3,4], [1,2,3]) # 4 operands
Set:[2, 3]
>>> x.union(range(10)) # non-MultiSets work too
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
```

6. *Composition.* Our solution is below, with comments from the description mixed in with the code. This is one case where it's probably easier to express a problem in Python than it is in English:

```
File: lunch.py

class Lunch:
 def __init__(self): # make/embed Customer and Employee
 self.cust = Customer()
 self.empl = Employee()
 def order(self, foodName): # start a Customer order simulation
 self.cust.placeOrder(foodName, self.empl)
 def result(self): # ask the Customer about its Food
 self.cust.printFood()

class Customer:
 def __init__(self): # initialize my food to None
 self.food = None
 def placeOrder(self, foodName, employee): # place order with Employee
 self.food = employee.takeOrder(foodName)
 def printFood(self): # print the name of my food
 print self.food.name

class Employee:
 def takeOrder(self, foodName): # return a Food, with requested name
 return Food(foodName)

class Food:
 def __init__(self, name): # store food name
 self.name = name

if __name__ == '__main__':
 x = Lunch() # self-test code
 x.order('burritos') # if run, not imported
 x.result()
 x.order('pizza')
 x.result()

% python lunch.py
```

```

burritos
pizza

```

7. *Zoo Animal Hierarchy*. Here is the way we coded the taxonomy on Python; it's artificial, but the general coding pattern applies to many real structures—form GUIs to employee databases. Notice that the `self.speak` reference in `Animal` triggers an independent inheritance search, which finds `speak` in a subclass. Test this interactively per the exercise description. For more fun, try extending this hierarchy with new classes, and making instances of various classes in the tree:

```

File: zoo.py

class Animal:
 def reply(self): self.speak() # back to subclass
 def speak(self): print 'spam' # custom message

class Mammal(Animal):
 def speak(self): print 'huh?'

class Cat(Mammal):
 def speak(self): print 'meow'

class Dog(Mammal):
 def speak(self): print 'bark'

class Primate(Mammal):
 def speak(self): print 'Hello world!'

class Hacker(Primate): pass # inherit from Primate

```

8. *The Dead Parrot Skit*. Here's how we implemented this one. Notice how the `line` method in the `Actor` superclass works: by accessing `self` attributes twice, it sends Python back to the instance twice, and hence invokes *two* inheritance searches—`self.name` and `self.says()` find information in the specific subclasses. We'll leave rounding this out to include the complete text of the Monty Python skit as a suggested exercise:

```

File: parrot.py

class Actor:
 def line(self): print self.name + ':', `self.says()`

class Customer(Actor):
 name = 'customer'
 def says(self): return "that's one ex-bird!"

class Clerk(Actor):
 name = 'clerk'
 def says(self): return "no it isn't..."

class Parrot(Actor):
 name = 'parrot'
 def says(self): return None

class Scene:
 def __init__(self):
 self.clerk = Clerk() # embed some instances
 self.customer = Customer() # Scene is a composite
 self.subject = Parrot()

 def action(self):
 self.customer.line() # delegate to embedded

```

```
self.clerk.line()
self.subject.line()
```

## Lab 7: Exceptions and built-in tools

1. *try/except*. Our version of the `oops` function follows. As for the noncoding questions, changing `oops` to raise `KeyError` instead of `IndexError` means that the exception won't be caught by our `try` handler (it “percolates” to the top level and triggers Python's default error message). The names `KeyError` and `IndexError` come from the outermost built-in names scope. If you don't believe us, import `__builtin__` and pass it as an argument to the `dir` function to see for yourself.

```
File: oops.py
def oops():
 raise IndexError

def doomed():
 try:
 oops()
 except IndexError:
 print 'caught an index error!'
 else:
 print 'no error caught...'

if __name__ == '__main__': doomed()

% python oops.py
caught an index error!
```

2. *Exception objects and lists*. Here's the way we extended this module for an exception of our own (here a string, at first):

```
File: oops.py
MyError = 'hello'

def oops():
 raise MyError, 'world'

def doomed():
 try:
 oops()
 except IndexError:
 print 'caught an index error!'
 except MyError, data:
 print 'caught error:', MyError, data
 else:
 print 'no error caught...'

if __name__ == '__main__':
 doomed()
```

```
% python oops.py
caught error: hello world
```

To identify the exception with a class, we just changed the first part of the file to this:

```
File: oop_oops.py

class MyError: pass

def oops():
 raise MyError()

...rest unchanged...
```

Like all class exceptions, the instance comes back as the extra data; our error message now shows both the class, and its instance (<...>).

```
% python oop_oops.py
caught error: __main__.MyError <__main__.MyError instance at 0x00867550>
```

Remember, to make this look nicer, you can define a `__repr__` method in your class to return a custom print string; see the unit for details.

3. *Error handling.* Finally, here's one way to solve this one; we decided to do our tests in a file, rather than interactively, but the results are about the same.

```
File: safe2.py

import sys, traceback

def safe(entry, *args):
 try:
 apply(entry, args)
 except:
 traceback.print_exc()
 print 'Got', sys.exc_info()[0], sys.exc_info()[1] # type, value

import oops
safe(oops.oops)

% python safe2.py
Traceback (innermost last):
 File "safe2.py", line 5, in safe
 apply(entry, args)
 File "oops.py", line 4, in oops
 raise MyError, 'world'
hello: world
Got hello world
```

Also see the [solution file directories](#) for answers to additional lab sessions.