Python for [Perl] Programmers

Aahz
aahz@pythoncraft.com
http://pythoncraft.com/

Powered by PythonPoint
http://www.reportlab.com/

Meta Tutorial

- I'm hearing-impaired
  - Please write questions if at all possible (also helps with my book)
- Two breaks
  - 15 minutes each, class will resume ***promptly***
- Slides and scripts on web
  - Resources at end of slideshow (including all URLs)

This Class Is

- Fast
  - First two sections double-fast
- Intermediate level
  - Skim or skip lots of basics
  - See on-line tutorial for intro
- Focused Tour / Survey
  - Based on common questions from `comp.lang.python`

This Class Is Not

- Advocacy
- Bashing

  "P/P" indicates compare/contrast

  Perl and Python have same fundamental goal: programmer productivity

First code: Python 2.2.1

```
from __future__ import generators
import sys, re
import fileinput

def grep(seq, regex):
    "yields items from seq that match regex"

    regex = re.compile(regex)
    for line in seq:
        if regex.search(line):
            yield line

if __name__ == '__main__':
    regex = sys.argv.pop(1)
    input = fileinput.input()
    for line in grep(input, regex):
        sys.stdout.write(line)
```

Interactive Python

- Quick demo

Contents

- Background
- Core Language I
  - Quick start
- Core II
  - In-depth
- Library
- Resources

Background

- History
- Dev process
- Philosophy
- Documentation

History

- Started 1989 on Mac
- Based on ABC and Modula-3
- Interface to C
  - Platform-neutral except for C's roots to Unix

Dev History

- 1.5.2 released 4/1999
  - Red Hat 7.x still defaults :-(
- 2.0 released 10/2000
  - Move to SourceForge
- 2.2.1 released 4/2002
  - Current release; some apps / extensions require 2.1.x (notably Zope 2.4.x)
- 2.3 by end of 2002
  - Small release, but busy developers

Platforms

- CPython
  - Reference implementation in C, portable to almost all platforms that support ANSI C
  - This class, "python" == "CPython"
- Jython
  - Java-based implementation, very similar to CPython
  - Major differences are memory management and OS-based libraries
  - Not covered in this class

Dev Process

- Guido van Rossum
  - BDFL: Benevolent Dictator For Life
- PEPs
  - Python Enhancement Proposals
  - Covers both process and features
  - Examples:
  - PEP 3 describes how to handle bug reports
  - PEP 278 is Universal Newline Support
  - PEP 283 is plan/schedule for Python 2.3

Making Decisions

- Informal
  - Apache-style voting (-1, -0, +0, +1), but Guido wants ***reasons*** more than votes
- BDFL Pronouncement
  - It's over
- Guido intuition
  - Guido (almost) always right
  - Explanations sometimes suboptimal

P/P: Philosophy

- Perl

  TMTOWTDI:

  There's More Than One Way To Do It

- Python

  There's Only One Way

Zen of Python

- Tim Peters

  Channeling Guido van Rossum

- `import this`

  Python 2.1.2 or higher

20 Pythonic Theses

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!

Why Whitespace?

- ABC
  - Research shows 3 or 4 spaces best
- Good code already indents
  - No brace wars

Guido Style

- PEP 8
  - Python Style Guide
- Readability
  - Follow rules when sensible
  - Use judgment for breaking rules

Style Summary

- 4-space indent
- Never mix tabs/spaces

  `python -t` or `python -tt`
- Punctuation

  one space around =, compare operators

  space after comma, colon, semi-colon

  no space for parens

Doc Tour

- Standard docs
  - Tutorial
  - Library Reference
  - Global Module Index
  - Language Reference
  - What's New
- Search

Core Language I

- Data
- Control structures
- Functions, Modules, Packages
- OOP
- Applications

Built-in Data Types

- `None`
- Numbers
- Sequences / Strings
- Dictionaries
- `True` / `False`

None

- P/P: `undef`

```
$ perl -e "print undef;"
$ python -c "print None"
None
$ python
>>> None
>>> print None
None
```

- Set explicitly
  - Except falling off end of function

Numbers

- Integer

  Platform C `int`, usually 32 bits

- Long

  P/P: Similar to `Math::BigInt`, but integrated into language

- Float

  Platform `double`

- Complex

  `0+1j` -- two floats

int/long Integration

- `1` is int
- `1L` is long
- Python 2.2+ autoconversion
  - `2 ** 100` => long
  - triggered on `OverflowError`
- int disappearing
  - Sometime between Python 2.4 and 3.0

Float Fun

- Python doesn't hide

```
>>> 1.1
1.1000000000000001
>>> print 1.1
1.1
>>> repr(1.1)
'1.1000000000000001'
>>> str(1.1)
'1.1'
```

Sequences

- Types
  - Strings
  - Tuples
  - Lists
- What's a sequence?
  - Array/vector; contiguous storage of items

Strings

- Delimiters

```
',",''',"""
r',r",r''',r"""
u',u",u''',u"""
ur',ur",ur''',ur"""
```

- No implicit interpolation

```
"%s %s" % (foo, bar)
```

  More later in the Text Processing section

String examples

```
"Double quotes don't muck with single quotes"
'<meta value="Single quotes good with tags">'

'''
This is
a multi-line
string
'''

>>> r"abc\"xyz" '\\'
'abc\\"xyz\\'
```

Lists

- Perl array
- Syntax:

```
>>> l = [1, 2, 3]
>>> l.append(2)
>>> l.pop()
2
>>> l.insert(0, 'abc')
>>> l
['abc', 1, 2, 3]
>>> l = [None] * 1000
>>> l = [8, 5, 13, 1]
>>> # Note: sort is in-place
... l.sort()
>>> l
[1, 5, 8, 13]
```

P/P: Lists

- No interpolation

```
>>> [1, "abc", [4, 5], (9, 8, 7)]
[1, 'abc', [4, 5], (9, 8, 7)]
```

Tuples

Immutable sequence of heterogeneous items

Constructor is comma, not parentheses:

```
>>> 1,'s'
(1, 's')
>>> 'abc',
('abc',)
```

But use parens for clarity

Empty tuple does require parens:

```
>>> a = ()
>>> a
()
>>> type(a)
<type 'tuple'>
>>> len(a)
0
```

Tuple Assignment

- Tuple LHS
- Must match number

```
>>> a,b = 3,5
>>> b
5
>>> a,b = b,a
>>> a,b = 7,8,9
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: unpack tuple of wrong size
```

Indexes

- Zero-based
- Negative

  Starts from end of sequence
- Out of bounds exception

```
>>> s = "abcdef"
>>> s[1]
'b'
>>> s[-2]
'e'
>>> s[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Slices

- Slice between elements

```
>>> s = "abcdef"
>>> s[0:2]
'ab'
>>> s[1:]
'bcdef'
>>> s[-2:]
'ef'
>>> s[:-3]
'abc'
>>> s[3:10]
'cdef'
>>> s[:3] + s[3:]
'abcdef'
```

Slice Assignment

```
>>> L = [0,1,2,3]
>>> L[1:3] = ['a','b','c','d']
>>> L
[0, 'a', 'b', 'c', 'd', 3]
>>> L[:2] = []
>>> L
['b', 'c', 'd', 3]
```

Dictionaries

- Dict is Perl hash (mostly)

  Also called "mapping" or "associative array"

- Syntax:

```
d = {
    'a': 1,
    1: [11, 12, 13]
    ('bar', 5): "xyz"
    }
```

- Lists cannot be key

Mutable vs. Immutable

- Immutable objects

  Strings, numbers, tuples, some class instances, and combinations

- Mutable objects

  Lists, dicts, most classes and class instances

- Why immutable?

  Dict keys, space efficiency

Mutable Subtleties

- Lists embedded in tuples

```
>>> a = (1, ['foo'], 's')
>>> a[1] = {'foo': 'bar'}
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> a[1].append('bar')
>>> a
(1, ['foo', 'bar'], 's')
```

  Can't be dict key

- Classes

  Cannot force class immutable; it's an implied contract in code and documentation

Tuples vs. Lists

- Which to use?

  Convention enforced by Guido:

  heterogeneous for tuples (light-weight struct) and homogeneous for lists

  Other than hetero/homo, use lists except when you need immutability or space efficiency -- list methods make your life easier

  (As with the rest of the Style Guide, use your judgment)

Data Constructors

- Constructors also converters
- Examples

```
>>> str(478)
'478'
>>> int("AC", 16)
172
>>> tuple(['abc'])
('abc',)
>>> tuple('abc')
('a', 'b', 'c')
>>> dict( (['a','d'], (5,'foo'), ['x',1]) )
{'a': 'd', 'x': 1, 5: 'foo'}
```

`True` / `False`

- Python 2.3

  Comparisons will return `True` or `False` instead of `1` or `0`

- Python 2.2.1

  `True` and `False` added as builtin singletons (like `None`) to make backporting from 2.3 easier, but no other changes made

Control Structures

- `if` / `while`

  P/P: Same as Perl, modulo short-circuit and no assignment in expressions

- `for`

  P/P: Equivalent to Perl's `foreach`, sort of

- `try`

  `try`/`except`

  `try`/`finally`

if

```
if <cond>:
    ...
elif <cond>:
    ...
else:
    ...
```

`if` Example

```
if line:
    if line.startswith('foo'):
        foo(line)
    elif line.startswith('bar'):
        bar(line)
    else:
        process(line)
else:
    cleanup()
```

while

```
while <cond>:
    ...
    [ continue / break ]
    ...
else:
    ...
```

- `else` skipped on `break`

while Example

```
done = False
while not done:
    result = getInfo()
    if result == END_TOKEN:
        done = True
```

for

```
for <var> in <iterable>:
    ...
    [ continue / break ]
    ...
else:
    ...
```

- `else` skipped on `break`
- P/P: similar to `foreach`
  - But can use iterators, which Perl doesn't have

for Examples

```
# print numbers from 0 through 9
for i in range(10):
    print i

L = ["foo", "bar", "", "xyz"]
for item in L:
    if not item:
        continue
    process(item)
    if item == "STOP":
        break
else:
    print "No breaks encountered"
```

`for` vs. strings

```
>>> for c in "abc":
...     print c
...
a
b
c
>>> for item in ('abc', ('def', 'xyz')):
...     print item
...
abc
('def', 'xyz')
```

Exceptions

- Structured throw of control
  - Dynamic scope, not lexical (unlike variables)
  - Unwinds call chain
- Errors
  - Exceptions mostly used for errors, but also used as a control structure (e.g. `for` loop terminates on `StopIteration` or `IndexError`)
- User-defined exceptions
  - Inherit from `Exception` or subclass of `Exception`

Exception Syntax

- try/except

```
try:
    ...
except [<exception>, [<variable>]]:
    ...
else:
    ...
```

- try/finally

```
try:
    ...
finally:
    ...
```

Using Exceptions

- List specific exception
- Get more info

  `sys.exc_info`

  `traceback`

Exceptions Demo

- `exceptions.py`

Functions and Modules

- Functions
- Modules
- Packages

Functions

- Syntax

```
def <name>([<parameter list>]:
    [global <var name>]
    ...
    [return [<expression>]]
```

- Calling func
  - Passed parameters must match param list
- `return` not required
  - Function falling off end returns `None`
  - Bare `return` also returns `None`

Default Arguments

```
def get_int(num_tries=3):
    for i in range(num_tries):
        print "Enter number: ",
        num = raw_input()
        try:
            num = int(num)
            return num
        except ValueError:
            pass
    raise ValueError, "No valid integer input"

x = get_int()
y = get_int(1)
```

- Avoid mutables

  Binding to parameter performed at compile time

Keyword Arguments

- Named parameter in call

```
def SQLexec(statement, connection=DEFAULT,
        commit=False)
    connection.send(statement, commit)

SQLexec(q)
SQLexec(q, commit=True)
SQLexec(q, commit=False,
    connection=getConnection())
SQLexec(q, getConnection(), True)
```

Variable Arguments

- `*args`

  Sequence of args

- `**kwargs`

  Dict of args

  "kwargs" == "keyword args"

Tuples and Functions

- Need parens to pass tuple:

  Compare

  ```
  f(1, 2)
  f( (1,2) )
  ```

Copy Semantics

- Immutables don't need copy
- Mutables must be copied
  - If you don't want them modified, that is

Copy Example

`list.sort()` is an in-place operation; here we create a function that returns the list

```
>>> def sort(L):
...     L.sort()
...     return L
...
>>> L = [11, 7, 12, 5]
>>> sort(L[:])
[5, 7, 11, 12]
>>> L
[11, 7, 12, 5]
>>> sort(L)
[5, 7, 11, 12]
>>> L
[5, 7, 11, 12]
```

Notice how failing to make copy modifies original list

Functions Demo

- `func.py`

Modules

- Script is module
- Executed only once
  - Only on first import

`import`

- Three forms

```
import foo
from foo import bar
from foo import *
```

- `import *`

  Pollutes namespace

  Makes debugging difficult

  Does not import names starting with "_"

  If `__all__` defined, uses that list

`import` Demo

- `grep.py`

Packages

- Package is directory

  `__init__.py`

- Example

```
foo/
    __init__.py
    bar.py
    baz.py
```

- Importing

```
import foo
foo.baz.x()

from foo import bar
bar.y()
```

OOP

- Syntax
- `__init__()`
- Inheritance
- Class vs. instance

`class` Syntax

```
class <name>[(<base class list>)]:
    ...
    [def __init__(self, [<param list>]):
        ...
        ]
    ...
    [def <name>(self, [<param list>]):
        ...
        ]
```

- Like module
  - Code at `class` scope gets executed once

`__init__()`

- Called automatically
  - After instance is created
- Not constructor

Inheritance

- Dynamic

  All inheritance in Python is resolved at run-time

- Multiple inheritance

  Mix-in classes provide convenience methods

is-a vs. has-a

- Inheritance vs. composition

  Use inheritance only when derived class is-a subset or extension of base class, or to inherit behavior (i.e. mix-in). Otherwise, use composition (make an instance of class you want to use an attribute). For example, an `Employee` class should contain a `person` attribute (not inherit from `Person`), but `SalariedEmployee` could inherit from `Employee`.

Typing vs. Inheritance

- Protocol

  Because Python only uses dynamic typing and late binding, classes do not need to inherit to fit static type definitions. Just define the appropriate methods and it works (except for a few operations that require builtin types).

Class vs. Instance

- Instance attribute
  - Access with `self`
- Class attribute
  - Access with class name

Private Data

- No private data

  Introspective object system makes it impossible to have truly private data. The closest Python comes is attributes with a leading double underscore (e.g. `__foo`), which get name-mangling -- but it's only intended for namespace separation between base class and derived class; a determined user can easily access the mangled name.

  Instead, Python relies on convention (attributes with a single leading underscore are considered private -- derived from `import *`) and documentation.

`class` demo

- `classes.py`

Applications

- Documenting apps
- Python files
- Distributing apps
- Testing
- Warnings

Documenting Apps

- Docstring

  String: first non-comment line of module, class, or function/method

  Can be multi-line string (and mostly should be)

  Docstring format: PEP 257

- Docstrings vs. comments

  Docstrings are accessible outside the source file; comments should only be used for information of interest to someone reading the source (i.e. implementation details)

Python Files

- `.py`

  Source files

- `.pyc`

  Compiled bytecode

- `.pyo`

  Optimized bytecode

- `.pyd`

  C Extensions

.pyo

- `python -O`

  Or `PYTHONOPTIMIZE` env var

  Discards some bytecodes, including `assert`

  ```
  if 0:
  ```

  ```
  if __debug__:
  ```

- `python -OO`

  Discards docstrings

Distributing Apps

- `Tools/freeze`
- Installer
- py2exe
  - Uses `distutils`
  - Windows-only

Testing

- Test first, then code
- `doctest`

  Simple to use, but susceptible to changes in output
- `unittest` (aka PyUnit)

  Class-based assertion testing

warnings

- PEP 230
  - Built on exceptions
- Filterable
- Python 2.3
  - Logging module planned

Core Language II

- Objects and Namespaces
- More Basics
- More OOP
- Advanced Features

Objects and Namespaces

- Objects
- Namespaces
- Scope

Everything an Object

- All data is object

  Functions, types, classes, class instances, iterators, generators, modules, and files all first-class objects

- Accessing objects

  Targets contain bindings to objects

  "Binding" used instead of "reference" because you can only deal with objects directly, never the references themselves (except in C API).

Targets

- Names
- Attributes

  Names in object space, accessed with dot notation
- Indexes / Keys
- Invisible targets

  E.g. `return`

  Important for understanding refcounts

Names

- What's a name?

  Bare word at builtin, module global, or function scope (more on scope shortly)

- "Name" instead of "variable"

  Names in Python don't contain data, only bindings; they're used like variables, mostly, but use "names" instead to remember the semantics

  Up to now I've used "variable" for convenience; from now on I'm using "name"

Each Object a Namespace

- Attributes

  Objects contain dict that maps key/value pairs to names and objects. For some objects (primarily built-in types, function locals, and some new-style classes), dict gets mapped to vector for speed.

Namespace Demo

- `namespace.py`

Binding Operations

- Creating names

  =, `for` (and list comprehensions), `def`, `import` (all forms), `class`, function/method calls

- Other targets

  `=` (sequences/maps), `return`, `yield`, `print`, function/method default parameters

Scope

- Two scope hierarchies:
  - Execution scope
  - Class inheritance scope
- Read-only
  - Non-local scopes are read-only unless made explicit

Execution Scope

- Searches three namespaces in order:
  - Function local
  - Module global
  - Builtins
- Shadowing
  - Locals shadow globals shadow builtins

Nested Scopes

- Functions inside functions
- Lexical scope
  - Not dynamic scope
  - Allows static analysis of source code
- Python 2.1

```
from __future__ import nested_scopes
```

Class Inheritance Scope

- Non-standard "scope" usage

  Some people disagree with me that class attribute/method inheritance should be discussed as a "scope", but it's a convenient way to explain the similarities with execution scope

- `self`

  Used in methods to distinguish between execution scope (implicit as in other functions) and the class inheritance scope (which is made explicit through `self`)

Scope Demo

- `scope.py`

Reference Counts

- Object deletion

  Objects are deleted when no bindings reference object

- Binding

  Each binding increases refcount

- Decrease refcount

  Rebinding

  Name going out of scope

  `del`

`del`

- Deletes bindings, not objects
- Can `del` parts of objects:

```
>>> L = [1,2,3,4,5]
>>> del L[1:3]
>>> L
[1, 4, 5]
>>> d = {'a': 1, 'b': 2, 'c': 5}
>>> del d['b']
>>> d
{'a': 1, 'c': 5}
```

Refcount Cascades

- Objects refer to other objects

  When an object gets deleted, all objects it refers to get decremented and may be deleted -- leading to a cascade of object deletions

- Large lists/dicts

  Deleting a large list or dict can take a long time as each object in the list/dict needs to be walked

GC

- Garbage collection

  In addition to refcount, not instead of

- Cycles

  Occur when two or more objects refer to each other; refcount can't go to zero. If no external targets refer to cycle, cycle is "garbage" and GC deletes it.

- `__del__()`

  Objects with `__del__()` can't be GC'd because of ordering problem

Refcount/GC Demo

- `refcount_gc.py`

More Basics

- Operators
- Handling Globals

Using ==

- `=` vs. `==`

  Can't use assignment in expression

- `==` vs. `is`

  Use `is` only for object identity comparisons or for builtin singletons (`None`, `True`, `False`)

Short Circuit

- Evaluates until truth known
- `and`

  Evaluates until first false expression
- `or`

  Evaluates until first true expression
- Example:

  `f() and g()`

  Calls `g()` only if `f()` true

True / False

- False

  `None`, `False`, `""`, `0`, `[]`, `()`

  `__nonzero__()` or `__len__()` return `0`

- True

  Everything else

- Truth testing

  If you just want the truth value of an object, always use the bare object in a conditional:

  ```
  if obj:
  ```

Mutability Again

- `=` vs. `+=` (and family)

  `=` always [re]binds

  mutable vs. immutable

  `+=` may rebind `self` to original object if mutable

Handling Globals

- `global`
- Class or class instance
- Module
  - Best for sharing globals across multiple modules
- Gang of Four
  - Modules are Singleton

Globals Demo

- `globals.py`

More OOP

- Class special methods
- New-style classes

Class Special Methods

- Double underscore

  Special methods have double underscore on both sides of their name (e.g. `__init__()`); do not use double-underscores for your own methods

  Leading double underscore causes name-mangling so derived classes can't access directly

Syntactic Sugar

- Syntax becomes method calls

  `a[i]` => `a.__getitem__(i)`

  `a[i] = x` => `a.__setitem__(i, x)`

  `a + b` => `a.__add__(b)`

  `a += b` => `a = a.__iadd__(b)`

  `a()` => `a.__call__()`

  `a(x, y, z)` => `a.__call__(x, y, z)`

  `float(a)` => `a.__float__()`

- Accessible magic

  Just set up the right methods for your classes

switch/case

- Dict-based function dispatch

```
def red(): pass
def green(): pass
def blue(): pass

dispatch = {
    'red': red,
    'green': green,
    'blue': blue
    }

dispatch[getColor()]()
```

New-style Classes

- PEPs 252, 253
- Classic can't subclass builtin types
  - New-style classes also referred to as "type/class unification"
- New-style classes have other benefits:
  - Properties
  - Static/class methods
  - Better multiple-inheritance lookup
  - Memory-saving `__slots__`
  - Easier metaclasses

Properties

- Classic classes

  Need to use `__getattr__()` and `__setattr__()`, both of which are clumsy (and usually overkill)

- New-style classes

  Property methods can be created for individual attributes

Static/class methods

- Static method
  Plain function in class namespace
- Class method
  Like instance method, except takes class argument

Multiple-Inheritance Lookup

- Shared ancestor

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
```

Classic classes visit A twice: DBACA

New-style classes visit A once: DBCA

`__slots__`

- Attributes
  - Normally stored in dict
  - `__slots__` creates indexed vector
  - Attribute names stored with class, not instance

Metaclasses

- Not touching

  I'm a Python expert, not a wizard

  Definition: metaclasses allow changing the way classes get instantiated (yes, classes, not class instances)

  Use case: creating classes dynamically for a simulation (but Python's dynamic introspection features usually don't require metaclasses for this)

New-style Class Demo

- `newstyle.py`

Advanced Features

- Iterators
- Generators
- List Comprehensions

Iterators

- PEP 234
- Efficiency

  Iterators don't need to create entire sequence
- Protocol

  `next()` method

  `raise StopIteration` when finished

Using Iterators

- `iter()`

  Create iterator by calling `iter()` on object or instantiating a class documented to be iterator

  `for` implicitly calls `iter()`

  `iter()` on iterator usually returns same iterator

- `iter(func, sentinel)`

  Use `iter()` on regular functions

  `func` is a function object, not function call

  Discouraged -- use generators if possible

Iterator Demo

- `iterator.py`

Generators

- PEP 255
- Function creates iterator

  Generator is a resumable function that maintains its local state between callbacks into the function (through the iterator interface)
- `yield`

  Keyword turns function into generator

  `from __future__ import generators` (not needed in Python 2.3+)

Generator Demo

- `generator.py`
- Notice
  - Generator code doesn't execute until first call to iterator's `next()`
  - Each call to generator creates new iterator

Generator Pipelines

- Pipelines save memory
- `genpipe.py`

  Imports `grep` and `uniq`

Iterators vs. Generators

- Generators simpler
  - No need to explicitly maintain state
- Iterators more controllable
  - Can manipulate internal state (e.g., `__del__` method on iterator object to handle cleanup)
  - Can be backported to older versions of Python

List Comprehensions

- Syntax

```
[<expr1> for <vars> in <iterable>
    for <vars2> in <iterable2> ...
    for <varsN> in <iterableN>
    if <condition>]
```

- Equivalent to

```
L = []
for <vars> in <iterable>:
    for <vars2> in <iterable2>:
    ...
        for <varsN> in <iterableN>:
            if <condition>:
                L.append(<expr1>)
return L
```

List Comp Examples

- Beware obfuscation

  Remember: "Readability counts"

```
>>> names = ["joe", "martha", "frank", "elizabeth"]
>>> [n.capitalize() for n in names]
['Joe', 'Martha', 'Frank', 'Elizabeth']
>>> [n for n in names if len(n)<6]
['joe', 'frank']
>>> # cross-product
... [(i,n) for i in range(2) for n in names]
[(0, 'joe'), (0, 'martha'), (0, 'frank'),
(0, 'elizabeth'), (1, 'joe'), (1, 'martha'),
(1, 'frank'), (1, 'elizabeth')]
```

Library

- Core
- Text Processing
- Essential 3rd Party
  - GUIs
  - Other

Core modules

- `__builtin__`
- `sys`
- `os`

`__builtin__`

- `__builtin__` vs `__builtins__`

  `__builtin__` is the actual module;
  `__builtins__` is the alias used for creating the namespace lookup

  Main reason is to avoid getting flooded with output when calling `vars()` in interactive mode.

os

- Somewhat Unix-centric
  - E.g. `rm` -> `os.remove()`
- Other platforms emulated
  - Read the docs

Text processing

- String manipulation
- Unicode

String Manipulation

- Interpolation
- String methods
- Regexes
- Text parsers

Interpolation

- Similar to C printf()

```
>>> '%s, %s!' % ('Hello', 'world')
'Hello, world!'
>>> '%10s' % "Hi!"
'       Hi!'
>>> '%.4f' % 1.5
'1.5000'
>>> d = {'foo': 1, 'bar': 5}
>>> 'Bar is %(bar)s' % d
'Bar is 5'
```

String Methods

- `s.method()` vs. `string.func(s)`

  String methods are favored over the `string` module unless backward compatibility is required

- Important methods

  `join()`, `split()`, `find()`, `replace()`, `startswith()`, `endswith()`

  `translate()` (but needs `string.maketrans()`) -- fastest way to delete characters

Building Strings

- Avoid '+'

  Especially in loops; Python immutable strings get constantly copied

- Use

  Interpolation

  `join()`

Regex

- `re` module

  P/P: regex syntax almost identical

  Use objects/methods to access functionality

  P/P: Python uses string methods instead of regexes for many common operations

- Python extensions

  `(?P`

  Named groups

`match()` vs. `search()`

- `match()`

  Matches only at start of string (or at optional position parameter); not quite equivalent to using "^"

- `search()`

  Standard regex search

Regex Demo

- `regex.py`

Text Parsers

- Regex vs. nested input

  Regexes don't handle nested constructs (e.g. XML) particularly well; text parsers are designed to handle this task

- Parsers

  mxTextTools

  SPARK

Unicode

- Encode/decode

  90% of the Unicode problems I've seen come from getting the encode and decode steps messed up

```
>>> unicode('Andr\x82')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII decoding error:
              ordinal not in range(128)
>>> unicode('Andr\x82', 'latin-1')
u'Andr\x82'
```

3rd Party Modules

- GUI
- Others

GUI Libraries

- Tkinter (default)
- wxPython
- PyGame (SDL)
- PyOpenGL
- PyQt

Other 3rd Party

- mxODBC
- PIL

  Python Imaging Library
- win32all
- NumPy
- PyChecker

Resources

- Main URL

`http://www.python.org/`

- Tutorial

`http://www.python.org/doc/current/tut/tut.html`

- PEPs

`http://www.python.org/peps/`

- Perl/Python conversion

`http://starship.python.net/crew/da/jak/cookbook.html`

Help

- `comp.lang.python` / `python-list`

  Bread-and-butter support

- `tutor@python.org`

  Good way to solidify your own knowledge by helping beginners

Reference Books

- *Programming Python*
- *Core Python Programming*
- *Python in a Nutshell*

  Not yet out
- *Python Essential Reference*

  If you get just one book, this is it

Other Books

- *Python Cookbook*

  Similar to *Perl Cookbook*

- *Python Programming on Win32*

Text Processing

- SPARK

http://pages.cpsc.ucalgary.ca/~aycock/spark/

- mxTextTools

http://www.lemburg.com/files/python/mxTextTools.html

Advanced Topics

- New-style classes

`http://www.python.org/2.2.1/descrintro.html`

- Unicode

`www.reportlab.com/i18n/python_unicode_tutorial.html`

- Regexes
  - `re` module

GUIs

- Tkinter

http://www.python.org/topics/tkinter/

- wxPython

http://www.wxpython.org/

- PyGame (SDL)

http://www.pygame.org/

- PyOpenGL

http://pyopengl.sourceforge.net/

- PyQt

http://www.riverbankcomputing.co.uk/pyqt/index.php

3rd Party Libs

- mxODBC

http://www.lemburg.com/files/python/mxODBC.html

- PIL

http://www.pythonware.com/products/pil/

- win32all

http://starship.python.net/crew/mhammond/

- NumPy

http://numpy.sourceforge.net/

- PyChecker

http://pychecker.sourceforge.net/

Dev resources

- Dev guide

`http://www.python.org/dev/`

- Version differences

`http://www.amk.ca/python/`

PEP 290

Jython

- Main

http://www.jython.org/

- Jython vs. CPython

http://jython.sourceforge.net/docs/differences.html