Laboratorio di Basi di Dati e Programmazione Web

Python tutorial

Stefano Zacchiroli
zack@pps.jussieu.fr

Nota

- Come slide di riferimento per il nostro Python tutorial useremo “*Introduction to Python”*, di Guido Van Rossum, 2002
  - http://www.python.org/doc/essays/ppt/lwnyc2002/intro22.pdf
- Tali slide sono inframmezzate da add-ons:
  - Nuove feature del linguaggio (up to Python 2.5 / 2.6)
  - Tool di supporto
  - “Zack's favorites”
  - Esercizi

ZOPE CORPORATION

Introduction to Python

LinuxWorld - New York City - January 2002

Guido van Rossum

Director of PythonLabs at Zope Corporation

guido@python.org
guido@zope.com

Why Python?

- Have your cake and eat it, too:
  Productivity **and** readable code
- VHLLs will gain on system languages
  (John Ousterhout)
- "Life's better without braces"
  (Bruce Eckel)

Tutorial Outline

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules & packages
- exceptions
- files & standard library
- what's new in Python 2.0 and beyond

ZOPE

Try It Out!

- If you brought a laptop into the classroom, feel free to play along
- Download Python from www.python.org
- Any version will do for this class
  - By and large they are all mutually compatible
  - Recommended version: 2.1.1 or 2.2
  - Oldest version still in widespread use: 1.5.2
  - Avoid 1.6/1.6.1 if you can
  - When using 2.0 or 2.1, upgrade to 2.0.1 / 2.1.1
  - 2.1.2 is coming soon!
- Use IDLE if you can

Interactive “Shell”

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: IDLE (GUI),
  python (command line)
- Type statements or expressions at prompt:

```
>>> print "Hello, world"
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

ZOPE

`ipython` – improved top-level

- L'interprete `python` è un buon punto di partenza per autoapprendimento e rapid prototyping interattivo
- `ipython` lo è ancora di più
  - Multi-line editing e multi-line history
  - TAB-completion (metodi, funzioni, variabili, attributi, …, filesystem!)
  - Help interattivo a portata di '?' post-fisso
    - `foo.bar  # accesso a foo.bar`
    - `foo.bar?  # help interattivo su foo.bar`
  - Auto-indentazione
  - Integrato con alcuni “major mode” di Python (e.g., matplotlib)
- Reference
  - Pacchetto `ipython` nella maggior parte delle distro

Numbers

- The usual suspects
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
    - In Python 2.2 and beyond, 2**100 does the same thing
  - 1j**2 -> (-1+0j)

ZOPE

Strings

- "hello"+"world" "helloworld" # concatenation
- "hello"*3 "hellohellohello" # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ell" # slicing
- len("hello") 5 # size
- "hello" < "jello" 1 # comparison
- "e" in "hello" 1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"

ZOPE

Lists

- Flexible arrays, *not* Lisp-like linked lists
  - a = [99, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
  - a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
  - a[0] = 98
  - a[1:2] = ["bottles", "of", "beer"]

    -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
  - del a[-1] # -> [98, "bottles", "of", "beer"]

More List Operations

```
>>> a = range(5)            # [0,1,2,3,4]
>>> a.append(5)             # [0,1,2,3,4,5]
>>> a.pop()                 # [0,1,2,3,4]
5
>>> a.insert(0, 42)         # [42,0,1,2,3,4]
>>> a.pop(0)                # [0,1,2,3,4]
5.5
>>> a.reverse()             # [4,3,2,1,0]
>>> a.sort()                # [0,1,2,3,4]
```

ZOPE

Dictionaries

- Hash tables, "associative arrays"
  - d = {"duck": "eend", "water": "water"}
- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - del d["water"] # {"duck": "eend", "back": "rug"}
  - d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - d["duck"] = "duik" # {"duck": "duik", "back": "rug"}

ZOPE

More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

ZOPE

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Tuples

- key = (lastname, firstname)
- point = x, y, z # parentheses optional
- x, y, z = point # unpack
- lastname = key[0]
- singleton = (1,) # trailing comma!!!
- empty = () # parentheses!
- tuples vs. lists; tuples immutable

ZOPE

Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

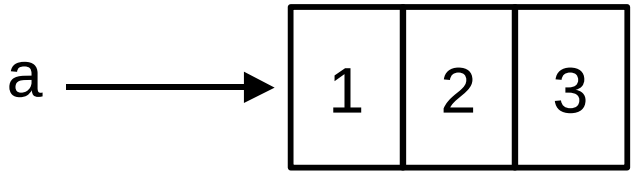
Reference Semantics

- Assignment manipulates references
  - x = y **does not make a copy** of y
  - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

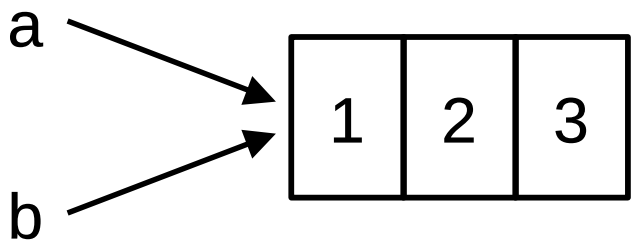
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Changing a Shared List

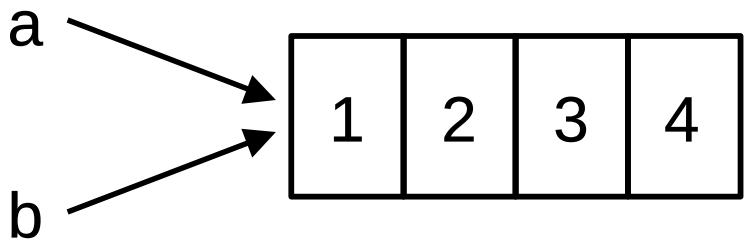
```
a = [1, 2, 3]
```




```
b = a
```



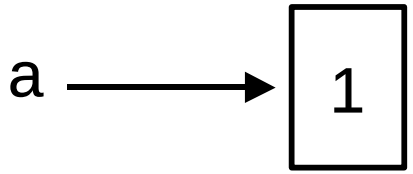

```
a.append(4)
```



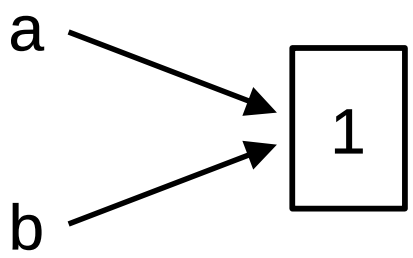
ZOPE

Changing an Integer

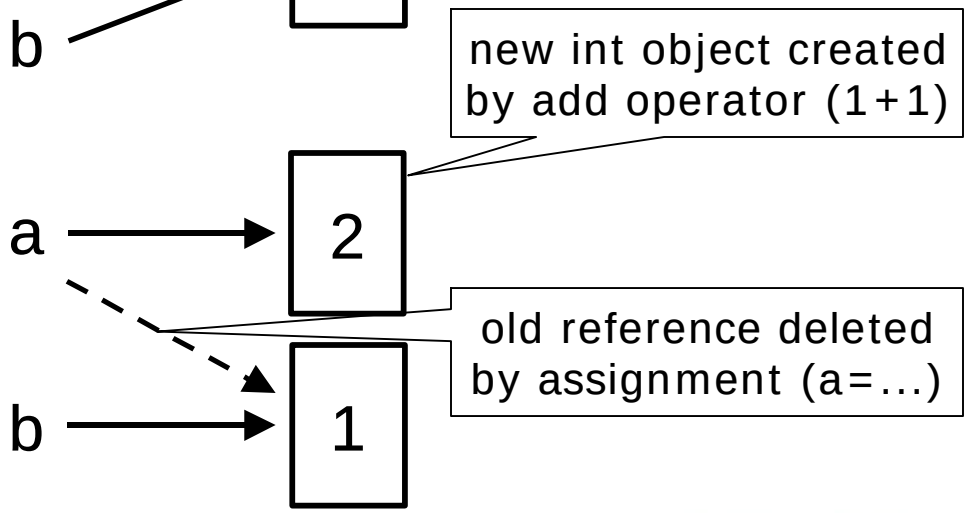
```
a = 1
```




```
b = a
```




```
a = a+1
```



ZOPE

Control Structures

if *condition*:
    statements
[elif *condition*:
    statements] ...
else:
    statements

while *condition*:
    statements

for *var* in *sequence*:
    statements

break

continue

ZOPE

Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

In C:

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {

    printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

ZOPE

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation""" # optional doc string
    statements

return                  # from procedure
return expression       # from function
```


ZOPE

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```


ZOPE

Auto-apprendimento

- Ogni “oggetto” che dispone di docstring la esporta come attributo `__doc__`
- La funzione `help()`, invocabile su ogni oggetto, attiva interattivamente un pager sulla docstring (uso interattivo)
  - Alternativa legacy al “?” postfisso di ipython
- La funzione dir(), invocabile su ogni oggetto, ritorna la lista dei nomi di metodi e attributi disponibili per quell'oggetto
  - Alternativa legacy alla TAB-completion di ipython

Classes

class *name*:

    "*documentation*"

    statements

-or-

class *name*(*base1*, *base2*, ...):

    ...

Most, *statements* are method definitions:

    def *name*(self, *arg1*, *arg2*, ...):

        ...

May also be *class variable* assignments

ZOPE

Example Class

```
class Stack:
    "A well-known data structure..."
    def __init__(self):             # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)        # the sky is the limit
    def pop(self):
        x = self.items[-1]          # what happens if it's
    empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0         # Boolean result
```


ZOPE

Using Classes

- To create an instance, simply call the class object:

```
x = Stack()        # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()          # -> 1
x.push(1)                          # [1]
x.empty()          # -> 0
x.push("hello")                    # [1, "hello"]
x.pop()            # -> "hello"    # [1]
```

- To inspect instance variables, use dot notation:

```
x.items            # -> [1]
```

ZOPE

Metodi e variabili privati

- Python non offre supporto per metodi e variabili *privati*
- Come surrogato, Python offre *name mangling* basato sul nome della classe corrente
  - Identificatori tipo __questo sono soggetti al mangling
    - Almeno due leading “_”, al max un trailing “_”
  - Il mangling riscrive *sintatticamente* “__questo” in “_classname__questo” dove “_classname” è il nome della classe corrente
  - Il mangling è attivo solo all'interno di class definition
- Non è resistente contro “attacker” determinati
- Reference: http://docs.python.org/tutorial/classes.html#private-variable

Subclassing

```
class FancyStack(Stack):
    "stack with added ability to inspect inferior stack items"

    def peek(self, n):
        "peek(0) returns top; peek(-1) returns item below that; etc."
        size = len(self.items)
        assert 0 <= n < size                    # test precondition
        return self.items[size-1-n]
```


ZOPE

Subclassing (2)

```
class LimitedStack(FancyStack):
    "fancy stack with limit on stack size"

    def __init__(self, limit):
        self.limit = limit
        FancyStack.__init__(self)               # base class
     constructor

    def push(self, x):
        assert len(self.items) < self.limit
        FancyStack.push(self, x)                # "super" method call
```


ZOPE

super()

- Nel caso di ereditarietà *singola*, non è (più) necessario ripetere il nome della superclasse per invocarne metodi: si può utilizzare super()

```
class MySubClass(MySuperClass):
      def __init__(self):
            super(MyClass, self).__init__()
      def meth(self, arg):
            super(MyClass, self).meth(arg)
```

- Gotcha: affinché la ricerca di metodi nella gerarchia delle classi funzioni come in altri linguaggi OO (e.g., Java), è necessario ereditare (anche indirettamente) dalla classe "`object`"
  - Riferimento "new style classes": http://www.python.org/doc/newstyle/

Class / Instance Variables

```
class Connection:
    verbose = 0                         # class variable
    def __init__(self, host):
        self.host = host                # instance variable
    def debug(self, v):
        self.verbose = v                # make instance variable!
    def connect(self):
        if self.verbose:                # class or instance variable?
            print "connecting to", self.host
```


ZOPE

Instance Variable Rules

- On use via instance (self.x), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (self.x = ...):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

ZOPE

Function decorator

- Nel gergo Python, un *decorator* è una funzione (di ordine superiore) che prende in input una funzione e ne ritorna una versione modificata
  - Tipicamente, un decorator invoca la funzione argomento
  - È applicabile anche a metodi (dato che sono funzioni)

```
def twice(f):                    # decorator
    def new_f(x):
        return f(f(x))
    return new_f                 # or, shorter: return lambda x: f(f(x))
def add7(x):
    return x +7
add14 = twice(add7)              # note: no “def”, function as values
```

Function decorator (cont.)

- Python offre una sintassi compatta per ridefinire funzioni o metodi utilizzando decoratori

```
@twice      # applico il decoratore twice alla funzione che segue
def add14(x):
    return x + 7
```

  - Equivalente a

```
def add14(x):
    return x + 7
add14 = twice(add14)
```

- Usi *notevoli* dei decoratori:
  - Type checking, logging, aspect-oriented programming
  - Metodi statici e metodi di classe ...

Metodi statici e di classe

- Static methods
  - Funzioni definite all'interno di classi
    - Invocabili senza istanza e senza classe
    - Equivalenti ai metodi statici di Java
  - @staticmethod

```
class C:
    @staticmethod
    def f(args,...):
        ...
```

- Class methods
  - Funzioni che dipendono da una classe (e.g. class variable)
    - Usano ereditarietà
    - Invocabili senza istanza
  - @classmethod

```
class C:
    @classmethod
    def f(cls,args,...):
        ...
```

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables
- Importing modules:
  - import re; print re.match("[a-z]+", s)
  - from re import match; print match("[a-z]+", s)
- Import with rename:
  - import re as regex
  - from re import match as m
  - Before Python 2.0:
    - import re; regex = re; del re

ZOPE

Packages

- Collection of modules in directory
- Must have __init__.py file
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo() # new

Catching Exceptions

```
def foo(x):
    return 1/x

def bar(x):
    try:
        print foo(x)
    except ZeroDivisionError, message:
        print "Can't divide by zero:", message

bar(0)
```


ZOPE

Try-finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()          # always executed
print "OK" # executed on success only
```


ZOPE

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range"
- try:
  something
  except: # catch everything
  print "Oops"
  raise # reraise

More on Exceptions

- User-defined exceptions
  - subclass Exception or any other standard exception
- Old Python: exceptions can be strings
  - WATCH OUT: compared by object identity, not ==
- Last caught exception info:
  - sys.exc_info() == (exc_type, exc_value, exc_traceback)
- Last uncaught exception (traceback printed):
  - sys.last_type, sys.last_value, sys.last_traceback
- Printing exceptions: traceback module

Exception hierarchy

- BaseException
- +-- SystemExit
- +-- KeyboardInterrupt
- +-- GeneratorExit
- +-- Exception
- +-- StopIteration
- +-- StandardError
- | +-- BufferError
- | +-- ArithmeticError
- | | +-- FloatingPointError
- | | +-- OverflowError
- | | +-- ZeroDivisionError
- | +-- AssertionError
- | +-- AttributeError
- | +-- EnvironmentError
- | | +-- IOError
- | | +-- OSError
- | | +-- WindowsError (Windows)
- | | +-- VMSError (VMS)
- | +-- EOFError
- | +-- ImportError
- | +-- LookupError
- | | +-- IndexError
- | | +-- KeyError
- | +-- MemoryError
- | +-- NameError
- | | +-- UnboundLocalError
- | +-- ReferenceError
- | +-- RuntimeError
- | | +-- NotImplementedError
- | +-- SyntaxError
- | | +-- IndentationError
- | | +-- TabError
- | +-- SystemError
- | +-- TypeError
- | +-- ValueError
- | +-- UnicodeError
- | +-- UnicodeDecodeError
- | +-- UnicodeEncodeError
- | +-- UnicodeTranslateError

- Le eccezioni in Python sono istanze di classi
- “except” segue la gerarchia della loro ereditarietà
  - “except” su una super-classe verrà seguito anche per tutte le eccezioni che sono istanze di sue sottoclassi

File Objects

- f = open(*filename*[, *mode*[, *buffersize*]])
  - mode can be "r", "w", "a" (like C stdio); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - buffersize: 0=unbuffered; 1=line-buffered; buffered
- methods:
  - read([*nbytes*]), readline(), readlines()
  - write(*string*), writelines(*list*)
  - seek(*pos*[, *how*]), tell()
  - flush(), close()
  - fileno()

Iteratori

- Python dispone di una nozione di iteratori per tutti gli oggetti (built-in o user defined) “iterabili”
- Perché un oggetto sia iterabile, deve esportare:
  - `__iter__()`: deve restituire un iteratore *fresh*
- Perché un oggetto sia un iteratore, deve esportare:
  - `next()`: restituisce il prossimo oggetto o solleva l'eccezione `StopIteration`
- Tutti gli oggetti iterabili sono compatibili con il costrutto `for` (e.g.: liste, file, tuple, dizionari, ...)

```
for i in [1,2,3]: …
for i in (1,2,3): …
for line in open('/a/b'): …
```

```
for k in dict.iterkeys(): …
for (k,v) in dict.iteritems(): …
```

List comprehension

- Sintassi compatta per esprimere liste
  - Che rappresentano un concetto chiaro
  - Richiedente computazione su singoli elementi per essere espresso
  - Paradigma comune nei linguaggi funzionali (e.g., Haskell)

```
[ i for i in range(4) ]       # caso degenere
[ i*2 for i in range(10) ]
[ (i,j) for i in range(5) for j in range(5) ]
[ 2*x for x in range(101) if x**2 > 3 ]
```

Standard Library

- Core:
  - os, sys, string, getopt, StringIO, struct, pickle, ...
- Regular expressions:
  - re module; Perl-5 style patterns and matching rules
- Internet:
  - socket, rfc822, httplib, htmllib, ftplib, smtplib, ...
- Miscellaneous:
  - pdb (debugger), profile+pstats
  - Tkinter (Tcl/Tk interface), audio, *dbm, ...

ZOPE

Standard library

- Highlights:
  - `re` – regular expressions
  - `string`
  - `datetime`
  - `sets` – insiemi
  - `random` – pseudo random (number) generation
  - `os.path` – filesystem path manipulation
  - `os` – operating system interface
  - `logging`
  - `subprocess`
  - E ancora: XML, networking (all levels), CGI, …
- Reference: http://docs.python.org/library/
- Su Debian: pacchetto `python-doc`

Python 2.0: What's New

- Augmented assignment: x += y
- List comprehensions:
  [s.strip() for s in f.readlines()]
- Extended print: print >>sys.stderr, "Hello!"
- Extended import: import foo as bar
- Unicode strings: u"\u1234"
- New re implementation (faster, Unicode)
- Collection of cyclic garbage
- XML, distutils

ZOPE

Python 2.1: What's New

- From __future__ import nested_scopes
  - def make_adder(*n*):
    def adder(x): return x+*n*
    return adder
  - add2 = make_adder(2)
  - add2(10) == 12
- Rich comparisons
  - Overload <, <=, ==, !=, >=, > separately
- Warnings framework
  - Prepare for the future

Python 2.2: What's New

- Iterators and Generators
  - from __future__ import generators
    def inorder(tree):
        if tree:
            for x in inorder(tree.left): *yield* x
            yield tree.label
            for x in inorder(tree.right): *yield* x
- Type/class unification
  - class mydict(dict): ...
- Fix division operator so 1/2 == 0.5; 1//2 == 0
  - Requires __future__ statement in Python 2.x
  - Change will be permanent in Python 3.0

ZOPE

Python 3.0: what's new

- Prima release non backward compatibile
- http://docs.python.org/3.0/whatsnew/3.0.html
- Highlights
  - print()
  - views – viste su oggetti con semantica set-based
  - ½ = 0.5
  - text vs data string; string Unicode by default
  - (a, *rest, b) = range(5)
  - Dictionary comprehension
  - New-style classes by default
  - `with` e context management
  - Class decorators

URLs

- http://www.python.org
  - official site
- http://starship.python.net
  - Community
- http://www.python.org/psa/bookstore/
  - (alias for http://www.amk.ca/bookstore/)
  - Python Bookstore

Further Reading

- Learning Python: Lutz, Ascher (O'Reilly '98)
- Python Essential Reference: Beazley (New Riders '99)
- Programming Python, 2nd Ed.: Lutz (O'Reilly '01)
- Core Python Programming: Chun (Prentice-Hall '00)
- The Quick Python Book: Harms, McDonald (Manning '99)
- The Standard Python Library: Lundh (O'Reilly '01)
- Python and Tkinter Programming: Grayson (Manning '00)
- Python Programming on Win32:
  Hammond, Robinson (O'Reilly '00)
- Learn to Program Using Python: Gauld (Addison-W. '00)
- And many more titles...

ZOPE

TIME FOR QUESTIONS

Esercizi

- Word trick: trovate, con uno script Python, tutte le parole inglesi che contengano almeno tre doppie consecutive
  - Usate una word list installata sulla macchina
  - (not so) E.g.: “committee” andrebbe bene se non per la “i”
- Implementare l'utility `join(1)` in Python
- Implementare 2 componenti di un importer IMDb → wiki
  1. Dato un URL HTTP di una pagina IMDB, genera una pagina wiki (sintassi a vs scelta) con le informazioni salienti: titolo, interpreti, regista, anno, …
  2. Data una pagina wiki nella sintassi di cui sopra, ne effettua la resa in HTML

Esercizi (cont.)

- Implementate un top-level per query interattive a dpkg
  - Al boot legge `/var/lib/dpkg/status` (database di dpkg)
  - Poi si comporta come shell: attende un nome di pacchetto
  - Una volta ricevuto stampa i campi corrispondenti al pacchetto
- Implementare un webchecker
  - Riceve come input un URL di pagina HTML
  - Scarica la pagina corrispondente usando wget
  - Parsa i suoi hyperlink e va in ricorsione su tutti quelli dello stesso dominio
  - Riporta tutti i broken link (identificabili ispezionando il return code di wget)