Programming Python

STEP Infotek internal seminar
Some introductory matters

The Python libraries

Basic Python (types and statements)

A practical exercise

Object-oriented programming in Python

Seminar contents
Introduction

Getting to know Python
Classifying languages

• Programming languages
  – compiled, statically typed, fast, low–level, useable for big projects
  – C/C++, Java, Ada, Pascal, Eiffel, ...

• Scripting languages
  – interpreted, dynamically typed, slow, high–level, only for ’toy’ projects
  – tcl, Perl, *Basic, Ruby, ...
Classifying Python

• A scripting language
  – interpreted, dynamically typed, slow, high-level
• Useful for both small and large projects
  – CORBA ORB, RDBMS, web browser...
• Syntactically unusual, closest relative is probably ABC
Python features

- High-level data types
- Very extensive standard libraries; can access anything
- Everything is a first-class object
  - classes, functions, modules, ...
- Convenient, readable programming
- OOP without a straitjacket
- Features from functional languages
What does this mean?

- Incredibly rapid development
- Source code is nearly always easily readable
- Perfect for integration tasks
- Can be used for just about anything
- Programming becomes fun!
Where does it come from?

- Christmas 1989, Guido van Rossum creates ABC for Unix/C hackers
- Developed at CWI in Amsterdam in 1990/91, then posted on comp.sources
- Classical open source project
- Development (and Guido) move to CNRI in the US in 1995
- Development moves to BeOpen in 2000
Internals

• There are two implementations:
  – the original, written as a C program
  – JPython, a Java re–implementation

• It’s easy to integrate C code into Python as if it were Python code

• Java code can be used as is from JPython as if it were Python code
Portability and access

- Python runs anywhere
  - Win32, Unix, Mac, DOS, Amiga, BeOS, VMS, Palm, WinCE, VxWorks
- Integrates with platform–specific extensions on most platforms
- Source code is generally portable unless depends on specific libraries
Python versions

• 1.4 and earlier are totally obsolete now, though code will usually run
• 1.5 is also obsolete, but less so
• 1.6 was just an interim release
• 2.0 introduces many new things
  – Unicode support
  – new language features (GC++)
  – new modules
Hello, world!

• The Python port of the famous hello world program:
  
  ```python
  print "Hello, World!"
  ```

• Write this line into a file named hello.py

• `python hello.py` runs it
The interpreter

- An excellent programming tool!
- Lets you try things out on the fly
- *Very* useful for playing around with libraries and modules
- *Extremely* useful for debugging
- Use it whenever you wonder how something works!
Getting a grip on it

Python basics
Variables and values

- Variables created by assignment
  
  \( a = 2 \) creates the variable \( a \)

- Variables don’t have types

- Values have types

- `type(a)` will give `<type 'int'>`

- `del a` will delete the variable
Numbers

- Mostly work as you would expect
  - \((2 + 2) \times 6 \Rightarrow 24\)
  - \(2^{8} \Rightarrow 256\)
  - \(2.0 / 3.0 \Rightarrow 0.66666667\)
  - \(2 / 3 \Rightarrow 0!\)
  - \(3 \% 2 \Rightarrow 1\)
Strings

- "this is a Python string"
- 'so is this'
- `str(5 + 5) => '10'
- "Hi " + "there" => "Hi there"
- "say " + "a" * 3 => "say aaa"
- `len("python") => 6`
String templates

- \texttt{temp = "\%s is very \%s"}
- \texttt{temp \% ("Python", "cool") => "Python is very cool"}
- A very useful feature for producing text output of all sorts
- Has more fancy capabilities as well
Lists

- \( a = [1, 2, 3, 4] \)
- \( \text{len}(a) \Rightarrow 4 \)
- \( a[0] \Rightarrow 1 \)
- \( a[-1] \Rightarrow 4 \)
- \( a[1 : -1] \Rightarrow [2, 3] \)
- \( a[200] \Rightarrow \text{error!} \)
- \( \text{range}(5) \Rightarrow [0, 1, 2, 3, 4] \)
• `list.append(value)`, appends value
• `list.append(value)`, appends value
• `list.index(value)`, finds index of value
• `list.index(value)`, finds index of value
• `list.count(value)`, counts occurrences
• `list.count(value)`, counts occurrences
• `list.remove(value)`, removes value
• `list.remove(value)`, removes value
• `list.reverse()`, reverses list
• `list.reverse()`, reverses list
• `list.sort()`, sorts list
• `list.sort()`, sorts list

List methods
Dictionaries

- *Incredibly* useful feature!
- `lang = {"no" : "Norwegian",
                      "en" : "English"}
- `lang["en"] => "English"
- `lang["hu"] => "Hungarian"
- `lang.has_key("se") => 0
- `lang["se"] => error!
More dictionaries

- `lang.keys() => ["no", "en", "hu"]`
- `lang.values() => ["Norw..."]`
- `lang.items() => [("no", "N...
- `len(lang) => 3`
- `lang.get("se", "Unknown") => "Unknown"`
Truth values

- No explicit true and false
- Instead, any value can be evaluated
- 0, "", [], {} are all false
- and, or and not work as usual
- 0 and 1 usually used for false/true
- == is the comparison operator, with <, >, <=, >=
The while loop

print "Language? ",

l = raw_input()

while l != "stop":
    print lang.get(l, "???")
    print "Language? ",
    l = raw_input()
while l:
    print "Language? ",
    l = raw_input()
    if l == "stop":
        break
    print lang.get(l, "???")
The for loop

codes = lang.items()
codes.sort()

for (code, name) in codes:
    print "%5s %s" % (code, name)
codes = lang.items()
codes.sort()
for (code, name) in codes:
    if name == "sv":
        continue
    print "%5s %s" % (code, name)
Functions

def get_input():
    print "Language? ",
    return raw_input()

l = get_input()

while l:
    print lang.get(l, "???")
    l = get_input()
Parameters

def double(x):
    return 2 * x

def hello(who = "World"):  
    print "Hello, %s!" % who
An exercise!

- Write a function `verify(str)`
- This function returns true if the first character in `str` is 1, 2 or 3
- Save it in a file `dict.py`
- Test it interactively in the interpreter
- Try `filter(verify, ['110', '552', '240', '345', '007', '777'])`
Modules

- Any Python file can be a module
- `import foo` will locate the `foo` module and load it in
- Its contents will be available as `foo.bar`
- The PYTHONPATH is where modules are searched for
An essential module

- The string module
- `string.split(str, div = " ") => list`
- `string.join(list, div = " ") => str`
- `string.find(str, what) => index`
Another!

- Write another function in dict.py: `interpret_multiline(str)`
- This takes a string with several lines separated by `\r\n`
- It should return a list of the lines
- However, it should stop when a line contains only "."
Tuples

- Used for grouping data
- Perfect for coordinates \((x, y)\)
- Cannot be modified
- Can be accessed just like lists
- Essentially read–only lists
- Much used for grouping in data structures
Conversions

- str(anything)
- int(anything)
- float(anything)
- list(anything)
- tuple(anything)
Yet another!

• `split_and_strip` takes a string of the form `'name "descr..."'`

• It returns a tuple `(name, descr)` without the quotes

• Try `map(split_and_strip, list_of_strings_of_above_form)`
The Python libraries

The batteries
When programming...

- ...always keep the HTML library reference ready
- This *will* save you lots of work
Library categories

- System libraries
- String services
- Operating system interaction
- Internet libraries
- Python language services
- Various
  - multimedia, maths, crypto
System libraries

The built-in functions
Interpreter information
Garbage collector (2.0)
Save/load objects
Pickle database
Data pretty-printer

- builtins
  - sys
  - gc
  - pickle
  - shelf
  - pprint
String services

- string Our old friend
- re Regexps
- struct Work with binary data
- StringIO In–memory files
- codecs String converters (2.0)
OS interaction

- os    Files, paths, processes
- time  Date and time information
- curses Text screen handling
- getopt Deal with cmdline args
- socket Network access
- *dbm  Simple keyed databases
Internet libraries

• Protocols: http, ftp, pop, smtp...
• URLs: urllib, urlparse
• Web: htmllib, cgi, Cookie
• Formats: rfc822, MIME, base64...
• XML:
  – pyexpat (2.0)
  – SAX 2.0 (2.0)
File handling

- open(name, mode) opens a file
- file.read(chars = all)
- file.readline()
- file.readlines()
- file.write(data)
- file.close()
A file example

```python
out = open("hello.txt", "w")
out.write("Hello, world!\n")
out.close()

inf = open("hello.txt")
inf.readline() => 'Hello, world!\n'
inf.readline() => ''
inf.close()
```
sys

- argv: cmdline arguments (list)
- exit: halts the interpreter
- path: Python search path
- version: Python version string
- platform: platform identifier
- stdin, stderr, stdout
A sys example

import sys

print sys.argv
print sys.platform
print sys.version
print sys.path
An exercise!

- python iso.py <standard–number>
- This should print the name of the ISO standard with the given number
- A list of numbers are in input.txt
socket

- `socket(AF_INET, SOCK_STREAM)`
- `socket.connect((IP, port))`
- `socket.send(string)`
- `socket.recv(max_bytes)`
- `socket.close()`
- `gethostbyname(name) => IP`
A socket example

```python
from socket import *
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('195.225.9.157', 79))
sock.send("larsga\r\n")
print sock.recv(1024 * 16)
sock.close()
```
A urllib example

```python
from urllib import urlopen

url = "http://www.infotek.no"

inf = urlopen(url)

print inf.read(1024)

inf.close()
```
Important tools

• GUI  wxPython, PyGtk, tkInter
• Web  Zope, Medusa, PyApache
• Data  DBI + database modules
• Strings  SPARK
• Python  DistUtils, PyUnit
A practical exercise

Making a DICT client
DICT

• A network protocol for talking to dictionary servers (see dict.org)
• Many free implementations with free dictionaries exist
• Both stand–alone and web clients can be found
• Defined in RFC 2229
The concept

- The server has a set of databases (typically webster, jargon & wn)
- Client can ask for specific words in one of these or in all of them
- Client can also search for words
- Much more functionality is also available
The protocol

- Based on simple text commands being sent to the server
- Server responds in clearly defined formats
- Numeric codes are used in responses to indicate the rough meaning of a response
A sample session

<= 220 pc-larsga dictd 1.4.9
=> SHOW DATABASES
<= 110 4 databases present
<= web1913 "Webster’s Revi..."
<= wn "WordNet (r) 1.6"
<= foldoc "The Free On-line..."
<= jargon "Jargon File..."
<= .
<= 250 ok
=> QUIT
<= 221 bye
Some conventions

• Error codes beginning with 1, 2 or 3 signify success
• String lists are terminated by a single line containing only a .
• Lines are terminated by '\r\n'
Some tips

• Keep receiving until the response contains the terminator
• Use socket.gethostbyname to find the IP address of a machine
• Use the functions you wrote earlier to process the data
• Server is ’pc–larsga.infotek.no’
Suggested steps

- Connect to server (port 2628) and print the welcome message
- Send ’SHOW DATABASES’ and just print the result
- Turn the result into a list
- Try to implement ’DEFINE * word’
Object-oriented programming
Making a class

- Like Java, Python supports object-oriented programming
- Unlike Java, it also supports other styles of programming
- Python’s OOP support has some similarities with Java’s, but rather more with Perl’s
A simple class

class Person:

    def __init__(self, given, sur):
        self._given = given
        self._sur = sur

    def get_name(self):
        return self._given + " " + self._sur

    def get_sortname(self):
        return self._sur + " , " + self._given
Using the class

```python
from person import *

me = Person("Lars Marius", "Garshol")

me.get_name()
    => 'Lars Marius Garshol'

me.get_sortname()
    => 'Garshol, Lars Marius'
```
Inheritance

class HungarianPerson(Person):
    def get_name(self):
        return self._sur + " " + self._given

class FrenchPerson(Person):
    def get_name(self):
        return string.upper(self._sur) + " " + self._given
Noteworthy stuff

• Inheritance from more than one class allowed
• Names beginning with __ are protected by mangling
• Names beginning with _ are protected by convention
• Classes and methods are first-class citizens
Exceptions

- A very useful feature in handling errors
- C/C++ programs often make every single function return 0/1
- This is ugly and error–prone
- Exceptions provide a far better way
Why exceptions are good

- What to do with errors depends on the context
- Errors often occur in the innermost parts of the code
- These tend not to know about the context at all
- Solution: exceptions
- These can be ’thrown’ by a function back to its callers until someone catches it
An example

def bing():
    bong()
def bong():
    bang()
def bang():
    raise HolySmokeError("Whoops!")

try:
    bing()
    print "OK"
except HolySmokeError, e:
    print "ERROR:", e
Exceptions are objects

- In Python and Java, exceptions are objects
- This makes it possible to put error information in them
- This information can be used by catchers
def insert_person(person):
    try:
        db = get_connection()
        db.run(insert_sql % person)
    finally:
        db.release()
An exercise

• Write the DICT client as a class
• Define get_databases and get_definition as methods
• Define a DictProtocolError
• Make the client raise this when the server indicates an error
How to go on from here

- Read the appendix to my book
- Read the Python tutorial
- Read 'Learning Python'
- Read the library reference
- Write scripts!
- Attend my Python and XML seminar!