Hackinggroup – Python Workshop – Part 1

A tale about dutch ducks with a fable for British comedy

Thomas Kastner Michael Rodler

20. Dezember 2010

Introduction – Wer san ma denn?

Michael Rodler

- aka f0rk, f0rki, f0rkmaster, Gabel, etc.
- Student SIB09
- 3 years coding python for fun
- 3 months coding python for profit

Thomas Kastner

- aka br3z3l, tom
- Student SIB08
- 4 years coding python for fun

Table of contents – Wos ma heit mochn

GO! – TU ES!

python

```
>>> import antigravity
```

Python Version Madness – Whyyyyy?

Python 1.x

- really really old
- sucks

Python 2.x

- new object system
- lots of legacy stuff in stdlib
- >= 2.5 on most (all) linux systems

Python 3.x

- not backwards compatible
- syntax cleanup
- stdlib cleanup

Every tutorial has this – Braucht ma oafoch

Wikipedia about “Hello World!”

A “Hello World” program is a computer program which prints out “Hello World!” on a display device. It is used in many introductory tutorials for teaching a programming language.

Every tutorial has this – Braucht ma oafoch

Wikipedia about “Hello World!”

A “Hello World” program is a computer program which prints out “Hello World!” on a display device. It is used in many introductory tutorials for teaching a programming language.

python

```
>>> print "Hello World!"
```

About variables & types

strong dynamic type system

- dynamic – any variable, any type
- strong – no magic type casting

About variables & types

strong dynamic type system

- dynamic – any variable, any type
- strong – no magic type casting

python

```
>>> x = 42
>>> x = "If it looks like a duck and quacks like a duck, it
    must be a duck."
>>> x = ["beer", "wine", "cheese"]
```

About variables & types

strong dynamic type system

- dynamic – any variable, any type
- strong – no magic type casting

python

```
>>> x = 42
>>> x = "If it looks like a duck and quacks like a duck, it
    must be a duck."
>>> x = ["beer", "wine", "cheese"]
```

python

```
>>> x = "duck" + 42
TypeError: cannot concatenate 'str' and 'int' objects

>>> x = "duck" + str(42)
```

python

```
>>> x = "Hello " + "I'm" + " a string"
>>> x.switchcase()
>>> x.lower()
>>> x.find('string')
>>> x.startswith("Hell")
>>> x.replace("I'm", "you're not")
>>> x.split(" ")
>>> ",".join(["a","b","c","d"])
```

python

```
>>> x = "Hello " + "I'm" + " a string"
>>> x.switchcase()
>>> x.lower()
>>> x.find('string')
>>> x.startswith("Hell")
>>> x.replace("I'm", "you're not")
>>> x.split(" ")
>>> ",".join(["a","b","c","d"])
```

python

```
>>> print "I'm your %s" % "bitch"
>>> print "Me is %d years old" % 12
>>> print "I have %s on %s for %s" % ("searched",
    '''wikiquote''', """this quotes""")
>>> print '%(language)s has %(#)03d quote types.' %
    {'language': "Python", "#": 2}
```

list, dict, tuple – Eh ois des söbe?

python

```
>>> party = ["cheese", "wine"]
>>> party.append("girls")
>>> party[0] = "beer"
>>> party += ["schnops", "punsch"]
>>> xmasparty = x[2:]
>>> print xmasparty
```

list, dict, tuple – Eh ois des söbe?

python

```
>>> party = ["cheese", "wine"]
>>> party.append("girls")
>>> party[0] = "beer"
>>> party += ["schnops", "punsch"]
>>> xmasparty = x[2:]
>>> print xmasparty
```

python

```
>>> x = {"awesome": "barney", 42: "the answer"}
>>> x["awesome"]
>>> x[42]
```

list, dict, tuple – Eh ois des söbe?

python

```
>>> party = ["cheese", "wine"]
>>> party.append("girls")
>>> party[0] = "beer"
>>> party += ["schnops", "punsch"]
>>> xmasparty = x[2:]
>>> print xmasparty
```

python

```
>>> x = {"awesome": "barney", 42: "the answer"}
>>> x["awesome"]
>>> x[42]
```

python

```
>>> x = (13, 37)
>>> a, b = x
>>> b, a = a, b
```

what **if**? – A if-Schleife

python

```
>>> if (True or False):
...     print "win"
... else:
...     print "fail"
...
```

python

```
>>> if (True or False):
...     print "win"
... else:
...     print "fail"
...
```

Truth value testing

Any object can be tested for truth value. The following values are considered *False*:

- None, False, 0
- any empty sequence, for example: “”, (), []
- any empty mapping, for example: {}

All other values are considered *true*

for as long as I live ...

python

```
>>> for word in ["python", "is", "awesome"]:
...     print word
```

python

```
>>> for word in "python is so fucking awesome".split():
...     print word

>>> for character in "python is so fucking awesome":
...     print character
```

for as long as I live ...

python

```
>>> for word in ["python", "is", "awesome"]:
...     print word
```

python

```
>>> for word in "python is so fucking awesome".split():
...     print word

>>> for character in "python is so fucking awesome":
...     print character
```

Iterating over what?

- returns next element, each round
- every python container type
- yo mama's objects

Curiosity killed the cat, but for a **while** I was a suspect.

python

```
>>> while not False:
...     print "print"
```

Curiosity killed the cat, but for a **while** I was a suspect.

python

```
>>> while not False:
...     print "print"
```

python

```
>>> while state != "legendary":
...     wait_for_it()
```

Curiosity killed the cat, but for a **while** I was a suspect.

python

```
>>> while not False:
...     print "print"
```

python

```
>>> while state != "legendary":
...     wait_for_it()
```

jumping

you can also **break** and **continue**.

About functions & methods

Where is the fucking difference?

About functions & methods

Where is the fucking difference? – There actually is none

About functions & methods

Where is the fucking difference? – There actually is none

python

```
>>> def doSomething(arg):
...     print "function args: " + str(arg)
...     return str(arg)
...
>>> x = doSomething("for the lulz")
```

About functions & methods

Where is the fucking difference? – There actually is none

python

```
>>> def doSomething(arg):
...     print "function args: " + str(arg)
...     return str(arg)
...
>>> x = doSomething("for the lulz")

>>> def func(arg0, arg1="default", *args, **kwargs):
...     print "arg0 =", arg0
...     print "arg1 =", arg1
...     print "args =", args
...     print "kwargs =", kwargs
...
>>> func(1, 2, 3, 4, 5, 6, john="doe", fu="bar")
>>> func(1, we="don't need", no="overloading!")
>>> func(0)
```

Exercises

- ✓ write a function (`find_mail`)
  - ✓ get text per argument
  - ✓ search for e-mail addresses (@ in word)
  - ✓ return list of e-mail addresses

RTFM – Hilfe zur Selbsthifle

- Python modules are generally well-documented
- Most Python programmers write at least minimal docstrings

RTFM – Hilfe zur Selbsthifle

- Python modules are generally well-documented
- Most Python programmers write at least minimal docstrings

python

```
>>> import os
>>> help(os)
>>> help(os.abort)
>>> dir(os)
```

RTFM – Hilfe zur Selbsthifle

- Python modules are generally well-documented
- Most Python programmers write at least minimal docstrings

python

```
>>> import os
>>> help(os)
>>> help(os.abort)
>>> dir(os)
```

*sh

```
[tom@workshop ~]$ pydoc os
[tom@workshop ~]$ pydoc os.path
```

RTFM – Hilfe zur Selbsthifle

- Python modules are generally well-documented
- Most Python programmers write at least minimal docstrings

python

```
>>> import os
>>> help(os)
>>> help(os.abort)
>>> dir(os)
```

*sh

```
[tom@workshop ~]$ pydoc os
[tom@workshop ~]$ pydoc os.path
```

python

```
>>> def function(a, b):
...     """Do X and return a list."""
...     pass
...
```

RTFM – Hilfe zur Selbsthifle

- `http://docs.python.org`
- `http://docs.python.org/library/re.html`

Zen of Python

python

```
>>> import this
```

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

python coding style – actually a religion

`http://www.python.org/dev/peps/pep-0008/`

Important

- 4 whitespaces indentation
- whitespaces around operators but not around brackets
- use docstrings!
- naming conventions
  - packages and modules – all lowercase
  - classes – CapWords/CamelCase
  - variables and functions – all lowercase with underscore as word seperator
- comparison with Singletons (e.g. None, True) use `is` keyword

"Programs must be written for people to read, and only incidentally for machines to execute." – Abelson & Sussman

"Programs must be written for people to read, and only incidentally for machines to execute." – Abelson & Sussman

not pythonic

```
colors = ['red', 'blue', 'green', 'yellow']
result = ''
for s in colors:
    result += s
```

"Programs must be written for people to read, and only incidentally for machines to execute." – Abelson & Sussman

not pythonic

```
colors = ['red', 'blue', 'green', 'yellow']
result = ''
for s in colors:
    result += s
```

pythonic

```
result = ''.join(colors)
result = ', '.join(colors)
```

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian W. Kernighan

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian W. Kernighan

not pythonic

```
for key in d.keys():
    print key

if d.has_key(key):
    do_something_with(d[key])
```

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." – Brian W. Kernighan

not pythonic

```
for key in d.keys():
    print key

if d.has_key(key):
    do_something_with(d[key])
```

pythonic

```
for key in d:
    print key

if key in d:
    do_something_with(d[key])
```

not pythonic

```
if x == True:
    do_something()

if len(items) != 0:
    do_something()

if items != []: # yuck!
    do_something()
```

not pythonic

```
if x == True:
    do_something()

if len(items) != 0:
    do_something()

if items != []: # yuck!
    do_something()
```

pythonic

```
if x:
    do_something()

if items:
    do_something()
```

pythonic

```
>>> items = "zero one two three".split()
>>> print items
```

pythonic

```
>>> items = "zero one two three".split()
>>> print items
```

not pythonic

```
index = 0
for item in items:
    print index, item
    index += 1

for i in range(len(items)):
    print i, items[i]
```

pythonic

```
>>> items = "zero one two three".split()
>>> print items
```

not pythonic

```
index = 0
for item in items:
    print index, item
    index += 1

for i in range(len(items)):
    print i, items[i]
```

pythonic

```
for (index, item) in enumerate(items):
    print index, item
```

bad idea

```
def bad_append(new_item, a_list=[]):
    a_list.append(new_item)
    return a_list
```

bad idea

```
def bad_append(new_item, a_list=[]):
    a_list.append(new_item)
    return a_list
```

good idea

```
def good_append(new_item, a_list=None):
    if a_list is None:
        a_list = []
    a_list.append(new_item)
    return a_list
```

everything is an object – ois, wirkli OIS

actually:

- no primitve types, pure OOP
- no literals, only references to singleton objects
- doesn't make much difference

python

```
>>> x = 21
>>> x += 21
>>> x = x.__add__(21)
```

We already know:

python

```
>>> import this
```

We already know:

python

```
>>> import this
```

python

```
>>> import os.path
>>> from random import randint
>>> from threading import *
```

the sys module

- python built-in library
- provides basic system information

python

```
>>> import sys
>>> dir(sys)
```

python

```
#!/usr/bin/python
import sys

def usage():
    print "Usage: %s -p" % sys.argv[0]
    sys.exit(1)

if sys.argv[1] == "-p" and len(sys.argv) == 3:
    do_stuff_with(sys.argv[2])
    sys.exit()
elif sys.argv[1] == "-h":
    usage()
else:
    usage()
```

the os module

- Miscellaneous operating system interfaces
- Mostly wrapper for C syscall functions

python

```
>>> import os
>>> os.getpid()
>>> os.chdir("/usr")
>>> os.getcwd()
>>> print os.linesep
```

the os.path module

- Common pathname manipulations
- Actually several different implementations
  - `posixpath` for UNIX-style paths
  - `ntpath` for Windows paths
  - `macpath` for old-style MacOS paths
  - `os2emxpath` for OS/2 EMX paths

python

```
>>> import os.path
>>> os.path.exists(".")
>>> os.path.getmtime("./somefile")
>>> os.path.abspath(os.path.join(os.getcwd(), "somedir",
    "somefile"))
```

We already know how to write to *stdout* with `print`. We can also use *stderr*.

python

```
>>> import sys
>>> print "I can't go, I've got this thing...."
>>> print >>sys.stderr, "a Penis."
```

Reading from *stdin* is quite easy.

python

```
>>> x = input("Please input x: ")
>>> y = raw_input("Please input y: ")
```

Reading from *stdin* is quite easy.

python

```
>>> x = input("Please input x: ")
>>> y = raw_input("Please input y: ")
```

Where's the difference?

- `raw_input` always returns a string

Reading from *stdin* is quite easy.

python

```
>>> x = input("Please input x: ")
>>> y = raw_input("Please input y: ")
```

Where's the difference?

- `raw_input` always returns a string
- `input` evaluates the input

Reading from *stdin* is quite easy.

python

```
>>> x = input("Please input x: ")
>>> y = raw_input("Please input y: ")
```

Where's the difference?

- `raw_input` always returns a string
- `input` evaluates the input $\rightarrow$ **DANGEROUS!**

Reading from *stdin* is quite easy.

python

```
>>> x = input("Please input x: ")
>>> y = raw_input("Please input y: ")
```

Where's the difference?

- `raw_input` always returns a string
- `input` evaluates the input $\rightarrow$ **DANGEROUS!**

python

```
>>> input("bad: ")
bad: __import__('os').getcwd()
```

But we can also write to files. To open a file you can use the built-in `open` function:

python

```
>>> help(open)
>>> f = open("bigbang.txt", "w+")
>>> f.readline()
'What am I supposed to do?'
>>> f.write("Well, have you considered telling her how you
    feel?\n")
>>> print >>f, "Leonard, I'm a physicist, not a hippie."
```

But we can also write to files. To open a file you can use the built-in `open` function:

python

```
>>> help(open)
>>> f = open("bigbang.txt", "w+")
>>> f.readline()
'What am I supposed to do?'
>>> f.write("Well, have you considered telling her how you
    feel?\n")
>>> print >>f, "Leonard, I'm a physicist, not a hippie."
```

File Modes

- `r`, `w`, `a` → read, write, append
- `+` → append to mode for r/w
- `b` → append to mode for binary data

New in version 2.5

python

```
>>> with open("mister.big", "r") as f:
...     content = f.read()
...     process(content)
```

New in version 2.5

python

```
>>> with open("mister.big", "r") as f:
...     content = f.read()
...     process(content)
```

What happens?

- opens file as it would normaly
- executes body
- closes file (even if errors occured!)

Exercises

- ✓ read a text file
- ✓ search text file for e-mail adresses
- ✓ write list of e-mail adresses to another file
- ✓ read file with e-mail addresses (as list)

Improvements – Wie mahs bessa mocht

Exercises

- ✓ read line by line
- ✓ rewrite `find_mail` to use stream objects
- ✓ use regex to search for valid e-mail addresses

Regular Expression – regex, oida!

python

```
>>> import re
>>> r =
    re.compile(r"[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}")
>>> valid = re.search("Net is 10.13.37.0") is not None
```

- `re.search` matches anywhere in the string string
- `re.match` matches only at the beginning of the

Things you could improve – Hausaufgaben

Exercises

- ✓ detect e-mails over multiple lines
- ✓ detect obfuscated e-mails
  - ✓ `user (at) example (dot) com`
  - ✓ `user at example dot com`

References

- the authors' epic python knowledge
- http://docs.python.org/
- Code like a Pythonista – http://python.net/~goodger/projects/pycon/2007/idiomatic/presentation.html
- http://diveintopython.org/