Kirk Byers – Python Course

Inhalt

[1 Week1 2](#_Toc443755580)

[1.1 Questions 2](#_Toc443755581)

[1.2 Code conventions 2](#_Toc443755582)

[1.3 Indentation matters! 3](#_Toc443755583)

[1.4 Comments 3](#_Toc443755584)

[1.5 Ho to pipe <stdin> into a python script 3](#_Toc443755585)

[1.6 Interpreter shell 4](#_Toc443755586)

[1.7 Strings 4](#_Toc443755587)

[1.7.1 String methods 6](#_Toc443755588)

[2 Week2 7](#_Toc443755589)

[2.1 Print and raw\_input of strings 7](#_Toc443755590)

[2.1.1 Use the format operator 7](#_Toc443755591)

[2.1.2 Printing with curly braces and format-method 7](#_Toc443755592)

[2.1.3 Biggest differences between python2 and python3 8](#_Toc443755593)

[2.1.4 raw\_input 8](#_Toc443755594)

[2.1.5 Additional info to strings 8](#_Toc443755595)

[2.2 Numbers 8](#_Toc443755596)

[2.2.1 useful math functions for network engineers 9](#_Toc443755597)

[2.2.2 Math library 10](#_Toc443755598)

[2.3 Lists 10](#_Toc443755599)

[2.3.1 Useful methods 11](#_Toc443755600)

[2.3.2 Copying lists 11](#_Toc443755601)

[2.4 Tuples 12](#_Toc443755602)

[2.5 Booleans 13](#_Toc443755603)

[3 Week3 13](#_Toc443755604)

[3.1 If Conditionals 13](#_Toc443755605)

[3.1.1 Meaning of false in python 13](#_Toc443755606)

[3.1.2 Operators 14](#_Toc443755607)

[3.1.3 If else statements 14](#_Toc443755608)

[3.1.4 nesting conditionals 14](#_Toc443755609)

[3.2 for loops 15](#_Toc443755610)

[3.2.1 Better way for both of these is the python enumerate-function 16](#_Toc443755611)

[3.2.2 continue 17](#_Toc443755612)

[3.2.3 break 17](#_Toc443755613)

[3.2.4 Nested for loops 18](#_Toc443755614)

[3.2.5 Pass as a placeholder 18](#_Toc443755615)

[3.3 Passing arguments into an script 18](#_Toc443755616)

[4 Week4 19](#_Toc443755617)

[4.1 While loops 19](#_Toc443755618)

[4.1.1 Ways to get out of a while loop 19](#_Toc443755619)

[4.2 Dictionaries 20](#_Toc443755620)

[4.2.1 Why do we need this? 20](#_Toc443755621)

[4.2.2 Examples 20](#_Toc443755622)

[4.2.3 Useful methods 21](#_Toc443755623)

[4.3 Exceptions 22](#_Toc443755624)

[4.3.1 Howto handle exceptions gracefully 23](#_Toc443755625)

[4.3.2 Howto handle multiple exception cases 24](#_Toc443755626)

[4.3.3 Store the exception object 24](#_Toc443755627)

[4.3.4 What you shouldn’t do 25](#_Toc443755628)

# Week1

## Questions

1. Why you need to learn programming?  
   Automation happens in a bigger level then in the past. Some amount of programming skills is going to be important.
2. What programming language should I learn?  
   Why python as a network engineer over something else? Large Community, being productive relatively quickly, large set of things you can do (netconf, ssh, snmp libraries… functional programming, asynchronous programming), default auf vielen Linux/MAC-OS: default on NX-OS default on Arista, network engineers and vendors are interested in python.
3. What is this language like? Characteristics?...  
   python people value clean code and convention. Easier to read other peoples code and my own code. For example perl is hard to read. Perl people show how cryptic they can make the code :), python tries to minimize the amount of syntax, BUT it has indentation!

## Code conventions

a. For variable names, function names, object names, and module names use lower case separated by underscore, for example:  
  
     my\_router   
     find\_set\_of\_devices  
     convert\_id\_string\_to\_list  
  
    b. For class names, capitalize the first letter of each word.  Do not use any underscores.  For example:  
  
﻿﻿﻿     ManyToManyField  
     ClientHistory  
     UserProfile﻿﻿﻿   
  
    c. For constants, use all upper case; use underscores for word separation.  
  
     PI = 3.14  
     EMAIL\_MODE  
     EMAIL\_FROM\_ADDRESS

## Indentation matters!

You should use 4 spaces as an indent.

>>>>> CODE <<<<<  
  
a = 10  
b = 20

if a == 10:  
    print 'First level of indentation'  
  
    if b == 20:  
        print 'Second level of indentation'  
        print 'Be consistent in your indentation'  
        print 'Use four spaces for each level'  
  
    print 'I am now out of the second if statement'  
  
print 'I am now out of the first if statement'

>>>>> END <<<<<

## Comments

Can be done with “#”.

## Ho to pipe <stdin> into a python script

>>>>> CODE (file test3.py) <<<<<

#!/usr/bin/env python  
  
import fileinput  
  
for line in fileinput.input():  
    print line.split(".")

>>>>> END <<<<<

Here is an example using this script where I echo an IP address into it and then the IP address is split into its octets.

$ echo '192.168.1.1'  | ./test3.py   
['192', '168', '1', '1\n']

## Interpreter shell

Test your code.

>>> a = 10

>>> b = 20

>>> c = 30

10,20,30 are in memory

A,b,c are references to the values in memory

* Shell useful for syntax tests
* Automatically evaluates expressions for you
  + 3 + 4 -> gives value back
* Expression => something that evaluates to a value
  + 3 + 4
  + 3 \* 4
  + A
* Statement => complete line of code
  + A = 10
  + B = 10
  + C = 30

Variables:

* a-zA-Z0-9\_
* no leading number
* case sensitive
* \_avar = 10 -> special meanings
* \_\_avar = 20 -> special meanings
* \_\_avar\_\_ = 20 -> special meanings

## Strings

Type(varname)

A = 10

A = ‘whatever’

No declarations needed!

* Dynamically typed programming language

Powerful, but you have to be careful!

Use “” or ‘’ ?? -> identical!

* Triple quotes:   
  ‘’’blablub  
  … I can have a ‘  
  … I can have a “  
  … I can have a newlines  
  … ‘’’

>>> a[1]

'l'

>>> a[0]

'b'

>>> a = 'whatever'

>>> a[-1]

'r'

>>> b = a

>>> b

'whatever'

>>> a

'whatever'

>>> a is b

True

* Point to the same value in memory

>>> a = 'newstring'

>>> b

'whatever'

* The new assignment of a creates a new object in memory and a new reference. B is sthill whatever

>>> c = 'c:\windows\new\test\dir'

>>> print c

c:\windows

ew est\dir

* \n and \t were interpreted by python
* You can escape them with \
* Or you can use a raw-string: c = r’blabla\n’

Strings in python2

* 8 bit representation
* That’s because there is a second type: Unicode

>>> c = u'whatever new'

>>> type(c)

<type 'unicode'>

>>> a

'newstring'

>>> b

'whatever'

>>> a + b

'newstringwhatever'

>>> a

'newstring'

>>> b

'whatever'

* Doesn’t modify the old strings

### String methods

**Strip: remove new line and white space**

>>> a = ' whatever \n'

>>> print a

whatever

>>> a.strip()

'whatever'

**Split: split the string**

>>> a = '192.168.1.10'

>>> a.split('.')

['192', '168', '1', '10']

* It forms a list with the values of the splitted string: [ ]

**Join: set it back to one string with a value between the strings**

>>> octets = a.split('.')

>>> ".".join(octets)

'192.168.1.10'

**Show all string methods available**

>>> dir(str)

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_formatter\_field\_name\_split', '\_formatter\_parser', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> b.upper()

'WHATEVER'

**Help on strings**

Help(str)

>>> b

'whatever'

>>> help(b.split)

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a', 'b', 'c', 'octets']

* Gives the defined variables back

**Check if substring is in a string**

>>> '192.168' in a

True

>>> '10.20' not in a

True

# Week2

## Print and raw\_input of strings

>>> ip\_addr = '192.168.1.1'

>>> print "My IP address is: " + ip\_addr

My IP address is: 192.168.1.1

>>> print "Router (" + router\_name + ") has an ip address of (" + ip\_addr + ")"

Router (sf-rtr-1) has an ip address of (192.168.1.1)

* That’s ugly!

### Use the format operator

>>> print "Router (%s) has an IP address of (%s)" % (router\_name, ip\_addr)

Router (sf-rtr-1) has an IP address of (192.168.1.1)

>>> print "Router (%s) has an IP address of" % router\_name

Router (sf-rtr-1) has an IP address of

Also string\_literals can be used and not variables

>>> a = 10

>>> b = 1000

>>> c = 707

>>> print "%s %s %s" % (a,b,c)

10 1000 707

>>> print "%15s %15s %15s" % (a,b,c)

10 1000 707

>>> print "%-15s %-15s %-15s" % (a,b,c)

10 1000 707

>>> print "%.2f" % a

3.14

>>> print "%.1f" % a

3.1

### Printing with curly braces and format-method

>>> print "Router ({0}) has an IP address of ({1})".format(router\_name, ip\_addr)

Router (sf-rtr-1) has an IP address of (192.168.1.1)

Probably the better method for the future. In python 2.7 also {} is possible without the position-numbers.

>>> print "Router ({}) has an IP address of ({})".format(router\_name, ip\_addr)

Router (sf-rtr-1) has an IP address of (192.168.1.1)

### Biggest differences between python2 and python3

In python3 they converted printing to be a function

print(“Test message”) for example…

raw\_input in python3 is renamed in “input”

use raw\_input only in python2

### raw\_input

prompt a user to input something.

{ class2 } » cat test4.py

ip\_addr = raw\_input("Please enter an ip address: ")

print "The ip address now is: %s" % ip\_addr

{ class2 } » python test4.py ~/scripts/ktbyers-course/class2

Please enter an ip address: 172.20.17.1

The ip address now is: 172.20.17.1

### Additional info to strings

>>> a = 'whatever'

>>> len(a)

8

>>> a = ''

>>> len(a)

0

## Numbers

We learned about integers before:

>>> a = 10

>>> b = 20

>>> c = 30

>>> type(a)

<type 'int'>

>>> a + b

30

>>> 10 / 3

3

>>> 11 / 3

3

* Python thinks this should be an integer divison

>>> 11 / 3.0

3.6666666666666665

* as soon as one number is a float python does a float divison

modulus like expected:

>>> 10 % 3

1

>>> 11 % 3

2

>>> 12 % 3

0

>>> a = 10000000000000000000000000000000000000000000000000000

>>> print a

10000000000000000000000000000000000000000000000000000

>>> a

10000000000000000000000000000000000000000000000000000L

>>> type(a)

<type 'long'>

* long integer because the space for integer is not so big to store this number.

>>> a = 2.0

>>> b = 7.091

>>> c = 1.73e7

>>> c

17300000.0

>>> d = 2.027e-3

>>> d

0.002027

* exponentional notation is possible

### useful math functions for network engineers

get binary numbers back:

>>> ip\_addr = '172.22.18.9'

>>> ip\_addr.split('.')

['172', '22', '18', '9']

>>> ip\_addr.split('.')[0]

'172'

>>> ip\_addr.split('.')[1]

'22'

BINARY

>>> bin(172)

'0b10101100'

>>> type(a)

<type 'str'>

* it is a string what we get back. A “representation”

>>> int(ip\_addr.split('.')[0])

172

>>> bin(int(ip\_addr.split('.')[0]))

'0b10101100'

Convert back:

>>> first\_octet\_bin

'0b10101100'

>>> int(first\_octet\_bin, 2)

172

* base “2” were used

HEX

>>> hex(172)

'0xac'

Convert back:

>>> first\_octet\_hex = hex(172)

>>> int(first\_octet\_hex, 16)

172

### Math library

>>> import math

>>> dir(math)

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

## Lists

>>> ip\_addr = ['8', '72', '190', '107']

>>> type(ip\_addr)

<type 'list'>

Lists don’t have to have the same data types inside:

>>> a = [107, 'whatever', 2.07, [] ]

>>> a

[107, 'whatever', 2.07, []]

>>> a[0]

107

>>> a[1]

'whatever'

>>> a[2]

2.07

>>> a[-1]

[]

>>> a[-2]

2.07

Lists are mutable:

>>> a[0] = 209

>>> a

[209, 'whatever', 2.07, []]

>>> a[1] = 'new string'

>>> a

[209, 'new string', 2.07, []]

>>> len(a)

4

>>> a

[209, 'new string', 2.07, []]

>>> b = [1,2,3]

>>> a + b

[209, 'new string', 2.07, [], 1, 2, 3]

>>> a

[209, 'new string', 2.07, []]

>>> b

[1, 2, 3]

>>> c = a + b

>>> c

[209, 'new string', 2.07, [], 1, 2, 3]

### Useful methods

>>> a.append(22)

>>> a

[209, 'new string', 2.07, [], 22]

>>> a.append('hellO')

>>> a

[209, 'new string', 2.07, [], 22, 'hellO']

>>> a.pop()

'hellO'

>>> a.pop(0)

209

>>> a

['new string', 2.07, [], 22]

>>> a.remove(22)

>>> a

['new string', 2.07, []]

* Removes the first list element with the value 22.

>>> del a[0]

>>> a

[2.07, []]

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(100, 110)

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

* Creates lists of number ranges

>>> a = range(100,110)

### Copying lists

>>> a

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> type(a)

<type 'list'>

>>> b = a

>>> a

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> b

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> a == b

True

>>> a is b

True

* a and b are the same object in memory!

>>> a[0] = 'new string'

>>> a

['new string', 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> b

['new string', 101, 102, 103, 104, 105, 106, 107, 108, 109]

* be careful about this!!

*So how do you make a copy of a list?*

Starting index 0 -> included

Ending index 5 -> excluded

>>> a[0:5]

[100, 101, 102, 103, 104]

>>> a[0:]

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> a[:]

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

>>> b = a[:]

>>> a == b

True

>>> a is b

False

* same values, but not the same objects in memory

## Tuples

Very similar to a list

>>> a = (1,2,3)

>>> a = (1,2,3,'some string')

>>> type(a)

<type 'tuple'>

* A TUPLE IS AN UNCHANGEABLE LIST!

>>> a[0] = 10

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

>>> a.append(100)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'tuple' object has no attribute 'append'

But why you want to use tuples?

Better for performance reasons. But in generally we should be fine with lists.

## Booleans

True or False.

* No quotes! Capital T/F.

>>> a = True

>>> b = False

>>> type(a)

<type 'bool'>

>>> a

True

>>> b

False

>>> a or b

True

>>> a and b

False

>>> not a

False

>>> not not a

True

>>> not not not a

False

# Week3

Review of what we did

Class1: introduction, strings

Class2: numbers, integers, floats, lists, tuples, Booleans

## If Conditionals

### Meaning of false in python

How does python treat for example

>>> a = ''

>>> if a:

Condition must be valuated true or false

**For strings it does**

‘’ -> false

Any other string ‘whatever’ -> true

**Numbers**

0 -> false

1. -> false

All others -> true

Blank **list** [] -> false

Blank **tuple** () -> false

Blank **dictionary** {} -> false

**None-Type**

>>> a = None

>>> type(a)

<type 'NoneType'>

* False

Python returns by default “None” if you don’t return something with your function.

### Operators

>>> 10 == 10

True

>>> 10 == 9

False

>>> a == 10

True

>>> a == 9

False

>>> a != 9

True

>>> 9 > 3

True

>>> 9 >= 3

True

>>> 17 <3

False

>>> 3 < 17

True

>>> 3 <= 17

True

### If else statements

if expression:

statement

statement

statement

elif expression:

statement

statement

statement

else:

statement

statement

statmenet

outside of conditional

### nesting conditionals

a = 10

b = 20

c = 30

if a == 10:

print "a is 10"

if b == 20:

print "b is 20"

ifelse.py:

a = 10

b = 20

c = 30

if a == 10:

print "a is 10"

if b == 20:

print "b is 20"

elif c == 30:

print 'c is 30'

else:

print 'whatever'

else:

print 'main else'

print 'outside of if'

{ class3 } » python ifelse.py

a is 10

b is 20

outside of if

try not to nest too much. -> restructure your code!

if (a == 10) and (b == 20) and (c == 30):

print 'a is 10, b is 20 and c is 30'

## for loops

>>> a = range(10)

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> for i in a:

... print i

...

0

1

2

3

4

5

6

7

8

9

>>> b = 'whatever'

>>> for i in b:

... print i

...

w

h

a

t

e

v

e

r

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> i = 0

>>> for element in a:

... a[i] = element\*\*2

... i += 1

...

>>> a

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> len(a)

10

>>> range(len(a))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> for i in range(len(a)):

... a[i] = a[i]\*\*2

...

>>> a

[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]

### Better way for both of these is the python enumerate-function

>>> a

[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561]

>>> for test in enumerate(a):

... print test

...

(0, 0)

(1, 1)

(2, 16)

(3, 81)

(4, 256)

(5, 625)

(6, 1296)

(7, 2401)

(8, 4096)

(9, 6561)

* Returns a tuple pair index/value
* i = index, element = value:

>>> for i,element in enumerate(a):

... a[i] = element\*\*2

...

>>> a

[0, 1, 256, 6561, 65536, 390625, 1679616, 5764801, 16777216, 43046721]

### continue

Test.py

a = range(10)

for i in a:

print i

if i == 6:

continue

print 'hello world'

{ class3 } » python fortest.py

0

hello world

1

hello world

2

hello world

3

hello world

4

hello world

5

hello world

6

7

hello world

8

hello world

9

hello world

### break

test.py

a = range(10)

for i in a:

print i

if i == 6:

break

print 'hello world'

{ class3 } » python fortest.py

0

hello world

1

hello world

2

hello world

3

hello world

4

hello world

5

hello world

6

### Nested for loops

Like always…

### Pass as a placeholder

for i in a:

pass

* Gives no syntax error. Good if you want to keep for loops in your code but you don’t need them at the moment.

## Passing arguments into an script

#!/usr/bin/env python

import sys

print sys.argv

{ class3 } » ./test6.py

['./test6.py']

{ class3 } » ./test6.py whatever

['./test6.py', 'whatever']

* Arguments are passed and saved as an element in a list

#!/usr/bin/env python

import sys

if len(sys.argv) == 2:

ip\_addr = sys.argv.pop()

print "The IP address is: %s" % ip\_addr

else:

print "You made an error"

{ class3 } » ./test6.py 172.23.1.1

The IP address is: 172.23.1.1

# Week4

## While loops

* For loops are good for iterating or I you have some number until you want to loop
* But what if you want to loop until something happens or forever?

>>> while (expression):

... statement

... statement

... statement

* There has to be something in you loop that causes the breakout of the loop
  + Change expression to false
  + Break statement

### Ways to get out of a while loop

>>> i = 0

>>> while i <= 10:

... print i

... print 'hello'

... i += 1

...

0

hello

1

hello

2

hello

3

hello

4

hello

5

hello

6

hello

7

hello

8

hello

9

hello

10

Hello

>>> i = 0

>>> while True:

... if i >= 11:

... break

... print i

... print 'hello'

... i += 1

...

0

hello

1

hello

2

hello

3

hello

4

hello

5

hello

6

hello

7

hello

8

hello

9

hello

10

Hello

## Dictionaries

A set of key:value pairs.

A set of mappings between key and the corresponding values (a hash in pearl)

### Why do we need this?

name = sf-rtr-1

ip\_addr = 1.1.1.1

serial\_number = FTX0000001

os\_version = 12.4.22T

...

12 attributes

With lists you can only remember the index for the value. Its much easier to remember a key.

Just like with DNS.

Another reason is performance. Its quicker to have a key and lookup the value.

### Examples

>>> a = {}

>>> type(a)

<type 'dict'>

>>>

>>>

>>> a = {

... 'name': 'sf-rtr-1',

... 'ip\_addr': '1.1.1.',

... 'serial\_number': 'FTX000001',

... 'os\_version': '12.4.15T',

... }

>>> a

{'serial\_number': 'FTX000001', 'ip\_addr': '1.1.1.', 'name': 'sf-rtr-1', 'os\_version': '12.4.15T'}

* The order when printing is deterministic. (always the same way). But you do not know whats the order after you created the dictionary.
* The keys have to be unique

### Useful methods

**assignment**

>>> a['name']

'sf-rtr-1'

>>> a['name'] = 'la-rtr-1'

>>> a['name']

'la-rtr-1'

**adding**

>>> a['model'] = '1941'

>>> a

{'serial\_number': 'FTX000001', 'model': '1941', 'ip\_addr': '1.1.1.', 'name': 'la-rtr-1', 'os\_version': '12.4.15T'}

**Delete**

>>> del a['model']

>>> a

{'serial\_number': 'FTX000001', 'ip\_addr': '1.1.1.', 'name': 'la-rtr-1', 'os\_version': '12.4.15T'}

**Get things**

>>> a.get('name')

'la-rtr-1'

* Better than a[‘name’] … because it gives no failure if you try to access a key that is not available

**Check if key is in dic**

>>> 'model' in a

True

>>> 'name' in a

True

**Iterate over dictionaries**

>>> a.keys()

['serial\_number', 'model', 'ip\_addr', 'name', 'os\_version']

>>> a.values()

['FTX000001', '1941', '1.1.1.', 'la-rtr-1', '12.4.15T']

>>> for key in a.keys():

... print key

...

serial\_number

model

ip\_addr

name

os\_version

>>> for value in a.values():

... print value

...

FTX000001

1941

1.1.1.

la-rtr-1

12.4.15T

**Items method**

>>> a.items()

[('serial\_number', 'FTX000001'), ('model', '1941'), ('ip\_addr', '1.1.1.'), ('name', 'la-rtr-1'), ('os\_version', '12.4.15T')]

>>> import pprint

>>> pprint.pprint(a.items())

[('serial\_number', 'FTX000001'),

('model', '1941'),

('ip\_addr', '1.1.1.'),

('name', 'la-rtr-1'),

('os\_version', '12.4.15T')]

* A list with couples (key/value)

>>> for k,v in a.items():

... print k

... print v

...

serial\_number

FTX000001

model

1941

ip\_addr

1.1.1.

name

la-rtr-1

os\_version

12.4.15T

## Exceptions

*#!/usr/bin/env python*

*a = {}*

*print 'hello'*

*print a['name']*

*print 'world'*

{ class4 } » ./test2.py ~/scripts/ktbyers-course/class4

hello

Traceback (most recent call last):

File "./test2.py", line 7, in <module>

print a['name']

KeyError: 'name'

* Program ends because of the key error

**Returncode:**

{ class4 } » ./test2.py ~/scripts/ktbyers-course/class4

hello

Traceback (most recent call last):

File "./test2.py", line 7, in <module>

print a['name']

KeyError: 'name'

{ class4 } » echo $? ~/scripts/ktbyers-course/class4 1

1

* We want errors to be handled gracefully.
* For example POST data that comes from a webform. Unexpected input generates an exception. If you don’t handle this the user gets back a 500 error.
* This has to be handled

### Howto handle exceptions gracefully

*#!/usr/bin/env python*

*a = {}*

*print 'hello'*

*try:*

*print 'string2'*

*print a['name']*

*print 'string3'*

*except KeyError:*

*print 'There was a key exception'*

*print 'string4'*

{ class4 } » ./test2.py ~/scripts/ktbyers-course/class4

hello

string2

There was a key exception

world

{ class4 } » echo $? ~/scripts/ktbyers-course/class4

0

### Howto handle multiple exception cases

The except only handles the exception if the type of the exception matches.

*#!/usr/bin/env python*

*a = {}*

*b = []*

*print 'hello'*

*a['name'] = 'whatever'*

*try:*

*print 'string2'*

*print a['name']*

*print b[0]*

*print 'string3'*

*except KeyError:*

*print 'There was a key exception'*

*except IndexError:*

*print 'There was an index exception'*

*print 'world'*

{ class4 } » ./test2.py ~/scripts/ktbyers-course/class4

hello

string2

whatever

There was an index exception

World

### Store the exception object

except KeyError as e:

print 'There was a key exception'

except IndexError, e:

print str(e)

print 'There was an index exception'

* “as e” is the new form and should be used!
* e points to the exception object in memory

{ class4 } » ./test2.py ~/scripts/ktbyers-course/class4

hello

string2

whatever

list index out of range

There was an index exception

World

* normally you capture the exception and store the information about it in a log…

### What you shouldn’t do

except Exception:

pass

except:

print 'whatever'

* Don’t match all exceptions with “except:”
  + “except:” catches also the keyboard exception (STRG+C)… what is bad.
  + You potentially hide programming errors.
  + A set of code calls a function and inside the function another function gets called. In the second function is an error. Python propagates the error upword and looks for try/excepts in the functions. If there is none it propagates it to the main program. So a “except:” there would handle all function errors….
  + That makes debugging difficult.
* “except Exception:” this generic type catches all exceptions. Excluded is the keyboard exception.
  + But the other issues would be still there!

# Week5

## Review of week 1-4.

1. **General**
   1. Python code structure
   2. Python interpreter shell
   3. Creating/Executing scripts
2. **Strings**
   1. Strip()
   2. Split() / .join()
   3. Substring in string
3. **Numbers**
   1. Numeric operators
   2. Int() and str() conversion
   3. Bin() and hex() functions
4. **Lists**
   1. Creating lists
   2. Accessing elements of a list
   3. Adding items using .append()
   4. Removing items using .pop()
   5. Iterating over lists
   6. List slices
5. **Tuples**
   1. Creating tuples
   2. Accessing elements of a tuple
   3. Iterating over tuples
6. **Booleans**
   1. Boolean operators
7. **Flow Control**
   1. If/Else
   2. For loops
      1. Using enumerate
      2. Using range
   3. While loops
      1. Break / continue
8. **Dictionaries**
   1. Creating dictionaries
   2. Adding elements
   3. Accessing/modifying dictionary elements
   4. Key in dict
   5. Get() method for accessing dictionaries
   6. Using .keys() and .items() for iteration
   7. Clarification on dictionary order
9. **Exception handling**
   1. Try/except structure

# Week6

## Functions, Part1

### Why use functions?

* Code reusability multiple times.
* Share code between applications

>>> def hello\_world():

... print "hello world"

>>> hello\_world()

hello world

>>> a = hello\_world()

hello world

>>> type(a)

<type 'NoneType'>

* There is always a return value! If you don’t specify one, than “none” gets returned.

### Docstrings

* Comments on what your function does

>>> def hello\_world():

... '''

... This is a docstring for hello world

... It takes no variables

...

... and returns True

... '''

... print 'hello world'

... return True

>>> help(hello\_world)

Help on function hello\_world in module \_\_main\_\_:

hello\_world()

This is a docstring for hello world

It takes no variables

and returns True

### Parameters

>>> def a\_sum(x, y):

... return x+y

...

>>> a\_sum(5, 10)

15

>>> a\_sum(20,10)

30

>>> a\_sum('str', 'ing')

'string'

* By default the variables/parameters are mapped by position.

>>> a\_sum(x=20,y=10)

30

>>> a\_sum(y=20,x=10)

30

* Order can be changed if explicitly given.

>>> def a\_sum(x,y,z):

... return x+y+z

...

>>> a\_sum(5,10,15)

30

>>> a\_sum(5,y=10,z=15)

30

* If you mix it the first given parameters has to be based on position. The named arguments must come last.

>>> def a\_sum(x,y,z=100):

... return x+y+z

...

>>> a\_sum(5,10,15)

30

>>> a\_sum(5,10)

115

* You can define default parameters in the function parameters.
* Useful for expanding functions if you don’t want to change your whole program.

## Namespaces

A core concept of python.

Namespace.py

x = 10

y = 20

z = 30

def simple\_func():

x = 100

y = 200

print x

print y

print z

simple\_func()

print x

print y

print z

{ class6 } » python namespace.py ~/scripts/ktbyers-course/class6

100

200

30

10

20

30

* There are different namespaces in this python script. The x and y in the function have a different namespace. The printed z inside the function has the same namespace, because in the local namespace inside the function there is no z. --> it looks at the module namespace next (after local in the function)… that’s the namespace in the actual file.
* When exiting out of a function the local variables get destroyed by garbage collection
* X and y are assigned inside the function, but this does NOT affect the global functions in the module namespace

Namespace.py

x = 10

y = 20

z = 30

def simple\_func():

def simple\_func2():

x = 1000

print x

print y

print z

x = 100

y = 200

print x

print y

print z

simple\_func2()

simple\_func()

print x

print y

print z

{ class6 } » python namespace.py ~/scripts/ktbyers-course/class6

100

200

30

1000

200

30

10

20

30

* First looks locally. Then any outer functions. (y = 200 !). Then any out functions or module namespace.

## Functions, Part2

### Names and Objects in Memory

>>> def f1(a\_list):

... a\_list.append("whatever")

...

>>> a = range(10)

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> f1(a)

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'whatever']

* List is modified by the function call
* Why does this work (namespaces)?
* DIFFERENCE BETWEEN NAME AND OBJECT IN MEMORY!
* F1 gets called and a as an argument passed
* A\_list points to the same object in memory as a at this moment
* That’s why it changes the list.

>>> def f2(a\_list):

... a\_list = []

... a\_list.append('something')

... print a\_list

...

>>> a = range(10)

>>> f2(a)

['something']

* A is unaffected now
* A\_list points to a in memory when given as a parameter
* BUT we make a\_list = [] --> that’s where a new object is created in memory!

BE CAREFUL WHEN YOU PASSING DICTS AND LISTS TO FUNCTIONS!! KNOW WHAT YOU ARE DOING!!

STRINGS AND INTEGERS ARE IMMUTABLE. THEY CAN NOT BE MODIFIED BY THE FUNCTIONS:

>>> def f3(a\_str):

... a\_str = 'whatever'

...

>>> f3('bla')

>>> a = 'hallo'

>>> f3(a)

>>> a

'hallo'

### Function name convention

* Lower case
* Underscore as separator

### Modules

A MODULE IS JUST A PYTHON FILE!

import

* Looks in the current directory
* If it cant find the module there it looks in the “python path”
* When you import a python module python executes all the code in the file!!

{ class6 } » cat test.py ~/scripts/ktbyers-course/class6

def a\_sum(x,y,z):

return x+y+z

print 10

{ class6 } » python ~/scripts/ktbyers-course/class6

Python 2.7.8 (default, Jul 28 2014, 01:34:03)

[GCC 4.8.3] on cygwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import test

10

* If we want to import without executing but execute something in the module we have to use if main…

if \_\_name\_\_ == '\_\_main\_\_':

# Get the inventory.

APICEMInventory()

### Using a module

After removing the print…

#### First way to import

{ class6 } » python ~/scripts/ktbyers-course/class6

Python 2.7.8 (default, Jul 28 2014, 01:34:03)

[GCC 4.8.3] on cygwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import test

>>>

>>> test.a\_sum(5,10,15)

30

#### Second way to import

{ class6 } » python ~/scripts/ktbyers-course/class6

Python 2.7.8 (default, Jul 28 2014, 01:34:03)

[GCC 4.8.3] on cygwin

Type "help", "copyright", "credits" or "license" for more information.

>>> from test import a\_sum

* The function is now imported into the local namespace:

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a\_sum']

>>> x = 10

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a\_sum', 'x']

>>> a\_sum(5,10,100)

115

Import in a python script:

{ class6 } » cat test2.py ~/scripts/ktbyers-course/class6

import test

a = test.a\_sum(5,10,15)

print a

{ class6 } » python test2.py ~/scripts/ktbyers-course/class6

30

# Week7

## Files

### Reading files

>>> f = open("r1\_cdp.txt", "r")

>>> f

<open file 'r1\_cdp.txt', mode 'r' at 0xfff0f2e0>

>>> a = f.readlines()

>>> a

['R1>show cdp neighbors detail \n', '-------------------------\n', 'Device ID: SW1\n', 'Entry address(es): \n', ' IP address: 10.1.1.22\n', 'Platform: cisco WS-C2950-24, Capabilities: Switch IGMP \n', 'Interface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11\n', 'Holdtime : 145 sec\n', '\n', 'Version :\n', 'Cisco Internetwork Operating System Software \n', 'IOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)\n', 'Copyright (c) 1986-2006 by cisco Systems, Inc.\n', 'Compiled Fri 28-Jul-06 15:16 by weiliu\n', '\n', 'advertisement version: 2\n', 'Protocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000\n', "VTP Management Domain: ''\n", 'Native VLAN: 1\n', 'Duplex: full\n']

* Each element of the list given back is a line of the file.
* We are now “at the end of the file”

>>> f.readlines()

[]

>>> f.seek(0)

>>> f.readlines()

['R1>show cdp neighbors detail \n', '-------------------------\n', 'Device ID: SW1\n', 'Entry address(es): \n', ' IP address: 10.1.1.22\n', 'Platform: cisco WS-C2950-24, Capabilities: Switch IGMP \n', 'Interface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11\n', 'Holdtime : 145 sec\n', '\n', 'Version :\n', 'Cisco Internetwork Operating System Software \n', 'IOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)\n', 'Copyright (c) 1986-2006 by cisco Systems, Inc.\n', 'Compiled Fri 28-Jul-06 15:16 by weiliu\n', '\n', 'advertisement version: 2\n', 'Protocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000\n', "VTP Management Domain: ''\n", 'Native VLAN: 1\n', 'Duplex: full\n']

* With seek(0) you go back to position 0 in the file

>>> f.seek(0)

>>> f.readline()

'R1>show cdp neighbors detail \n'

>>> f.readline()

'-------------------------\n'

>>> f.readline()

'Device ID: SW1\n'

>>> f.readline()

'Entry address(es): \n'

* Readline() returns line by line.

>>> f.seek(0)

>>> f.read()

"R1>show cdp neighbors detail \n-------------------------\nDevice ID: SW1\nEntry address(es): \n IP address: 10.1.1.22\nPlatform: cisco WS-C2950-24, Capabilities: Switch IGMP \nInterface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11\nHoldtime : 145 sec\n\nVersion :\nCisco Internetwork Operating System Software \nIOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)\nCopyright (c) 1986-2006 by cisco Systems, Inc.\nCompiled Fri 28-Jul-06 15:16 by weiliu\n\nadvertisement version: 2\nProtocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000\nVTP Management Domain: ''\nNative VLAN: 1\nDuplex: full\n"

* Read() returns the whole file content as one string.
* Good for regex matching for example

>>> f.seek(0)

>>> for line in f:

... print line

...

R1>show cdp neighbors detail

-------------------------

Device ID: SW1

Entry address(es):

IP address: 10.1.1.22

Platform: cisco WS-C2950-24, Capabilities: Switch IGMP

Interface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11

Holdtime : 145 sec

Version :

Cisco Internetwork Operating System Software

IOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)

Copyright (c) 1986-2006 by cisco Systems, Inc.

Compiled Fri 28-Jul-06 15:16 by weiliu

advertisement version: 2

Protocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000

VTP Management Domain: ''

Native VLAN: 1

Duplex: full

* Iterate over the file
* Double newline because of the file content \n and the \n of the print methods.

>>> f.seek(0)

>>> for line in f:

... print line.strip("\n")

...

R1>show cdp neighbors detail

-------------------------

Device ID: SW1

Entry address(es):

IP address: 10.1.1.22

Platform: cisco WS-C2950-24, Capabilities: Switch IGMP

Interface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11

Holdtime : 145 sec

Version :

Cisco Internetwork Operating System Software

IOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)

Copyright (c) 1986-2006 by cisco Systems, Inc.

Compiled Fri 28-Jul-06 15:16 by weiliu

advertisement version: 2

Protocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000

VTP Management Domain: ''

Native VLAN: 1

Duplex: full

* Now we strip the \n from the file. Now only the newline of the print method left.

### Writing files

!! be careful !!

* Windows stores newlines as \r\n
* Unix just \n

Python does an automatic convertion when reading in new lines from windows files.

That’s fine for reading in textbased files, but it’s a big problem for binary files.

If you are reading in or writing binary data do on windows:

>>> f = open("read\_file", "r**b**")

>>> f = open("write\_file", "w**b**")

* Tells python that this is a binary file and no convertion should be done!

>>> f = open("newfile", "w")

>>> f.write("test\n")

>>> f.close()

>>> exit()

* Would overwrite the file newfile and create a new one.

**Create a new file:**

>>> f = open("newfile", "w")

>>> f.write("This is a test\n")

>>> a = 'some other string\n'

>>> f.write(a)

>>> f.flush()

>>> f.close()

>>> exit()

{ class7 } » cat newfile ~/scripts/ktbyers-course/class7

This is a test

some other string

* Now the file gets buffered in the operating system level
* Use flush to write it out to the file. Or close() it.

**Append to a file**

Append opens the file and the pointer is already at the end of the file.

>>> f = open("newfile", "a")

>>> f

<open file 'newfile', mode 'a' at 0xfff0f2e0>

>>> f.write("some other text\n")

>>> f.write("some other text\n")

>>> f.write("some other text\n")

>>> f.close()

>>> exit()

{ class7 } » cat newfile ~/scripts/ktbyers-course/class7

This is a test

some other string

some other text

some other text

some other text

**Ensure the file gets closed (also when an exception occurs!)**

* There is another method instead of open and close.
* Use “with”:
* The close is automatically ensured

>>> with open("newfile") as f:

... f.readline()

... f.readline()

... f.readline()

...

'This is a test\n'

'some other string\n'

'some other text\n'

>>> f

<closed file 'newfile', mode 'r' at 0xfff0f2e0>

## Regular Exceptions

!!! you should use raw strings when working with regex in python. This is much simpler. You don’t have to worry about backslash escaping… !!!

!!! always comment regexs. They become very cryptic. !!!

>>> f = open("r1\_cdp.txt")

>>> a = f.readline()

>>> a

'Platform: cisco WS-C2950-24, Capabilities: Switch IGMP \n'

**>>> import re**

>>> dir(re)

['DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE', 'VERBOSE', 'X', '\_MAXCACHE', '\_\_all\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_version\_\_', '\_alphanum', '\_cache', '\_cache\_repl', '\_compile', '\_compile\_repl', '\_expand', '\_pattern\_type', '\_pickle', '\_subx', 'compile', 'copy\_reg', 'error', 'escape', 'findall', 'finditer', 'match', 'purge', 'search', 'split', 'sre\_compile', 'sre\_parse', 'sub', 'subn', 'sys', 'template']

>>> a\_var = re.search(r"Platform: ", a)

>>> a\_var

<\_sre.SRE\_Match object at 0xffc2f2c0>

>>> if a\_var:

... print "regex matches"

...

regex matches

>>> if re.search(r"Platform: ", a):

**Look at what matched:**

>>> a\_var.group()

'Platform: '

**Make it more complicate:**

. for any character

+ for any more character (does not include \n)

>>> a\_var = re.search(r"Platform: (.+)", a)

>>> a\_var.group()

'Platform: cisco WS-C2950-24, Capabilities: Switch IGMP '

>>> a\_var.group(1)

'cisco WS-C2950-24, Capabilities: Switch IGMP '

**The smallest match with a new blank:**

>>> a\_var = re.search(r"Platform: (.+?) ", a)

>>> a\_var.group(1)

'cisco'

>>> a\_var = re.search(r"Platform: (.+?) (.+?),", a)

>>> a\_var.group(1)

'cisco'

>>> a\_var.group(2)

'WS-C2950-24'

**One string:**

>>> f.seek(0)

>>> cdp\_data = f.read()

>>> cdp\_data

"R1>show cdp neighbors detail \n-------------------------\nDevice ID: SW1\nEntry address(es): \n IP address: 10.1.1.22\nPlatform: cisco WS-C2950-24, Capabilities: Switch IGMP \nInterface: FastEthernet1, Port ID (outgoing port): FastEthernet0/11\nHoldtime : 145 sec\n\nVersion :\nCisco Internetwork Operating System Software \nIOS (tm) C2950 Software (C2950-I6Q4L2-M), Version 12.1(22)EA8a, RELEASE SOFTWARE (fc1)\nCopyright (c) 1986-2006 by cisco Systems, Inc.\nCompiled Fri 28-Jul-06 15:16 by weiliu\n\nadvertisement version: 2\nProtocol Hello: OUI=0x00000C, Protocol ID=0x0112; payload len=27, value=00000000FFFFFFFF010221FF0000000000000019E845CE80FF0000\nVTP Management Domain: ''\nNative VLAN: 1\nDuplex: full\n"

Look for IP addresses:

>>> re.findall(r"IP address: .+", cdp\_data)

['IP address: 10.1.1.22']

>>> re.findall(r"IP address: (.+)", cdp\_data)

['10.1.1.22']

Look for Platform:

>>> re.findall(r"Platform: (.+?) (.+?),", cdp\_data)

[('cisco', 'WS-C2950-24')]

>>> re.findall(r"Platform: (.+?) (.+?), Capabilities: (.+)", cdp\_data)

[('cisco', 'WS-C2950-24', 'Switch IGMP ')]

>>> z = re.findall(r"Platform: (.+?) (.+?), Capabilities: (.+)", cdp\_data)

>>> for vendor,model,device\_type in z:

... print vendor

... print model

... print device\_type

...

cisco

WS-C2950-24

Switch IGMP

# Week8

## Modules

A python module is a python file.

Why? We want to be able to reuse code! My own code or code created by others.

>>> import re

>>> import pprint

* Here we are using modules others created
* Now lets create our own module

### Create a module

**First create the file:**

{ class8 } » cat test.py ~/scripts/ktbyers-course/class8

a = 100

print "Hello"

def some\_func():

print "Inside some\_func"

print "World"

### Import a module

**Way1:**

>> import test

Hello

World

* This brings the module in to my local namespace!
* You have to access the things with module name before.

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'test']

>>> test.some\_func()

Inside some\_func

>>> test.a

100

**Way2:**

>> from test import some\_func

* This brings in the function into my local namespace

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'some\_func']

>>> some\_func()

Inside some\_func

>>> from test import a

>>> a

100

**Way3:**

>>> import test as zzz

>>> zzz.a

100

* Import the module as a different name

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a', 'some\_func', 'zzz']

**What happens when you import a file?**

* Python goes over the file/module line by line.
* That’s why “Hello World” gets outputted when importing the module
* After the last line python goes back to your main program
* This happens only once (when you “import test” again it doesn’t print “Hello World”)
* The same things happens when you use Way2 (from test import…)

{ class8 } » cat test2.py ~/scripts/ktbyers-course/class8

import test

print "Inside test"

test.some\_func()

{ class8 } » python test2.py ~/scripts/ktbyers-course/class8

Hello

World

Inside test

Inside some\_func

### Modules create their own namespace

Just like functions, python modules create their own namespace.

For example:

{ class8 } » cat test.py ~/scripts/ktbyers-course/class8

a = 100

print "Hello"

def some\_func():

print "Inside some\_func"

print a

print a

print "World"

>>> import test

Hello

100

World

>>> test.some\_func()

Inside some\_func

100

>>> a = 77

>>> test.some\_func()

Inside some\_func

100

>>> test .a

100

>>> a

77

>>> test.a = 55

>>> test.some\_func()

Inside some\_func

55

### Separate executable code from importable code!

{ class8 } » python test.py ~/scripts/ktbyers-course/class8

**\_\_main\_\_**

{ class8 } » cat test.py ~/scripts/ktbyers-course/class8

print \_\_name\_\_

def some\_func():

print "Inside some\_func"

>>> import test

test

{ class8 } » cat test.py ~/scripts/ktbyers-course/class8 1

print \_\_name\_\_

def some\_func():

print "Inside some\_func"

if \_\_name\_\_ == '\_\_main\_\_':

print "hello"

**Execute it**

{ class8 } » python test.py ~/scripts/ktbyers-course/class8

\_\_main\_\_

hello

world

**Import it**

>>> import test

test

>>> dir(test)

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', 'some\_func']

## Packages

### How/where python searches for modules/packages

>>> import sys

>>> import pprint

>>> pprint.pprint(sys.path)

['',

'/usr/lib/python2.7/site-packages/ecdsa-0.13-py2.7.egg',

'/usr/lib/python2.7/site-packages/scp-0.10.2-py2.7.egg',

'/usr/lib/python2.7/site-packages/bnclient-0.3-py2.7.egg',

'/usr/lib/python2.7/site-packages/pycsco-0.3.4-py2.7.egg',

'/usr/lib/python2.7/site-packages/pyfg-0.47-py2.7.egg',

'/usr/lib/python2.7/site-packages/pyIOSXR-0.11-py2.7.egg',

'/usr/lib/python2.7/site-packages/junos\_eznc-1.3.1-py2.7.egg',

'/usr/lib/python2.7/site-packages/ncclient-0.4.7-py2.7.egg',

'/usr/lib/python2.7/site-packages/pyeapi-0.4.0-py2.7.egg',

'/usr/lib/python2.7/site-packages/gtextfsm-0.2.1-py2.7.egg',

'/usr/lib/python2.7/site-packages/xmltodict-0.9.2-py2.7.egg',

'/usr/lib/python2.7/site-packages/pexpect-4.0.1-py2.7.egg',

'/usr/lib/python2.7/site-packages/netaddr-0.7.18-py2.7.egg',

'/usr/lib/python2.7/site-packages/ptyprocess-0.5.1-py2.7.egg',

'/home/f0222557/ansible/lib',

'/home/f0222557/scripts/ktbyers-course/class8',

'/usr/lib/python27.zip',

'/usr/lib/python2.7',

'/usr/lib/python2.7/plat-cygwin',

'/usr/lib/python2.7/lib-tk',

'/usr/lib/python2.7/lib-old',

'/usr/lib/python2.7/lib-dynload',

'/usr/lib/python2.7/site-packages',

'/usr/lib/python2.7/site-packages']

**Modify by modyfing the PYTHONPATH variable:**

{ class8 } » env | grep PY ~/scripts/ktbyers-course/class8

PYTHONPATH=/home/f0222557/ansible/lib:

### What is a python package?

= A Directory, constructed in a special way.

{ class8 } » cd package1 ~/scripts/ktbyers-course/class8

{ package1 } » touch \_\_init\_\_.py

* Now it`s a python package!

>>> import package1

### What can I do with a package?

{ class8 } » cat package1/z\_test.py ~/scripts/ktbyers-course/class8

def some\_func():

print "Inside some\_func"

def some\_func2():

print "Func2"

if \_\_name\_\_ == '\_\_main\_\_':

print "hello"

print "world"

>>> import package1.z\_test

>>> package1.z\_test.some\_func()

Inside some\_func

>>> from package1.z\_test import some\_func

>>> some\_func()

Inside some\_func

>>> from package1 import z\_test

>>> z\_test.some\_func()

Inside some\_func

### What happens when you import a package?

\_\_init\_\_.py is executed!

{ class8 } » cat package1/\_\_init\_\_.py ~/scripts/ktbyers-course/class8

print "in \_\_init\_\_.py"

>>> import package1

in \_\_init\_\_.py

**What can you do with this?**

If you have f.e. 10 modules in the directory. You can say python to load them with the package.

In \_\_init\_\_.py for example: “from . import z\_test”

>>> import package1

in \_\_init\_\_.py

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'package1']

>>> dir(package1)

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_path\_\_', 'z\_test']

>>> dir(package1.z\_test)

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', 'some\_func', 'some\_func2']

* You can let \_\_init\_\_.py import a lot of other modules.
* For example you have a bunch of cdp functions and spread them in 3 different modules.
* You can then make a “cdp package” and load the 3 modules in the init file.
* Makes in easier to have an integrated set of code that gets load with one import command.

When importing you import a module or a library.

>>> import re

>>> import pprint

>>> import sys

>>> import paramiko

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'package1', 'paramiko', 'pprint', 're', 'sys']

In /usr/lib/python2.7

re.py is a module

pprint.py is a module

/usr/lib/python2.7/site-packages/paramiko

paramiko is a package

# Week9

Classes and Objects. Try not to overcomplicate this. A class is a “blueprint”. Functions in classes are called methods.

Each time you create an instance of a class you have an “object”.

## Classes

>>> class NetworkDevice(object):

... pass

NetworkDevice = Name of the class

Object = what we are **inheriting** from. If you don’t inherite from something you just specify the most common thing “object”.

### Create an instance of a class

>>> rtr3 = NetworkDevice()

>>> rtr3

<\_\_main\_\_.NetworkDevice object at 0xffec782c>

>>> rtr3.ip = '1.1.1.10'

>>> rtr3.username = 'ktb'

>>> rtr3.ip

'1.1.1.10'

### Konstruktor (\_\_init\_\_) & self

>>> class NetworkDevice(object):

... def \_\_init\_\_(self, ip, username, password):

... self.ip = ip

... self.username = username

... self.password = password

**\_\_init\_\_** is the constructor method. This method is automatically invoked when you create an instance of an object.

**self** we want to assign the parameters to the object. How do we go about doing that? What python does is when you call NetworkDevice AT FIRST an object in memory is created. Then the \_\_init\_\_ method is called. We need a way to refer to that object. The FIRST PARAMETER IN ALL methods is a reference to the object itself. The name should be choosen as “self”.

>>> rtr3 = NetworkDevice('10.1.1.1', 'bla', 'pw')

>>> rtr3

<\_\_main\_\_.NetworkDevice object at 0xffec79ec>

>>> rtr3.ip

'10.1.1.1'

**Class** = “blueprint for creating an object and for defining methods”

**Instance** = Object (in memory) of a Class

**Methods** = Functions in classes

**Parameters** = Names of the variables I am passing in

>>> class NetworkDevice(object):

... def \_\_init\_\_(self, ip, username, password):

... self.x = ip

... self.y = username

... self.z = password

>>> rtr4 = NetworkDevice('10.1.1.1', 'bla', 'pw')

>>> rtr4.x

'10.1.1.1'

### Methods

>>> class SomeClass(object):

... def \_\_init\_\_(self, x, y):

... self.x = x

... self.y = y

... def a\_sum(self):

... return self.x + self.y

... def a\_product(self):

... return self.x \* self.y

...

>>> a = SomeClass(3,7)

>>> a.x

3

>>> a.y

7

>>> a.a\_sum()

10

**Variables of the object can be changed.**

>>> a.z = 22

>>> a.z

22

>>> a.x = 7

>>> a.a\_sum()

14

>>> some\_str = '1.1.1.1'

>>> some\_str.split(".")

['1', '1', '1', '1']

* Behind the scenes this string is an object of an certain type
* One method the string has is “split()”

### Inheritence

First python looks in NewClass for an \_\_init\_\_ method. Then in its base class. Here this is “SomeClass”. The same it does for methods (here a\_sum()).

Order: Look for attribute in object, then inside the class, then in the base class.

Order: Look for method inside the class, then in the base class.

>>> class NewClass(SomeClass):

... pass

>>> b = NewClass(8,9)

>>> b.x

8

>>> b.y

9

>>> b.a\_sum()

17

**Other example (overwrite \_\_init\_\_):**

>>> class NewClass(SomeClass):

... def \_\_init\_\_(self, x, y, z):

... self.x = x

... self.z = z

>>> b = NewClass(8,9,10)

>>> b.x

8

>>> b.z

10

>>> b.y

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'NewClass' object has no attribute 'y'

**Next example (Call \_\_init\_\_ of base):**

>>> class NewClass(SomeClass):

... def \_\_init\_\_(self, x, y, z):

... self.z = z

... SomeClass.\_\_init\_\_(self, x, y)

…

>>> b = NewClass(8,9,10)

>>> b.x

8

>>> b.z

10

>>> b.y

9

>>> b.a\_sum()

17

**Next example (overwrite a method of base class):**

>>> class NewClass(SomeClass):

... def \_\_init\_\_(self, x, y, z):

... self.z = z

... SomeClass.\_\_init\_\_(self, x, y)

... def a\_sum(self):

... return self.x + self.y + self.z

...

>>> b = NewClass(8,9,10)

>>> b.x

8

>>> b.y

9

>>> b.z

10

>>> b.a\_sum()

27

### What can we do with this?

class NetworkDevice(object):

def \_\_init\_\_(self, ip, username, password):

self.ip = ip

self.username = username

self.password = password

def connect\_to\_device(self):

pass

def enter\_enable\_mode(self):

pass

def show\_version(self):

pass

**And Inheritance?**

For Connecting to network devices of different vendors:

Base class, subclass HP, subclass Cisco… overwrite things in the subclass. So changes in the HP-Class does not effect the Cisco class. A third class like Arista could easily be added.