GETTING STARTED WITH JUNOS & AUTOMATION

Umberto Manferdini

WHAT'S PYTHON

- Python a programming/scripting language
- It is pre-compiled!
  - No compilation when running your script
  - Not optimized as other languages like C

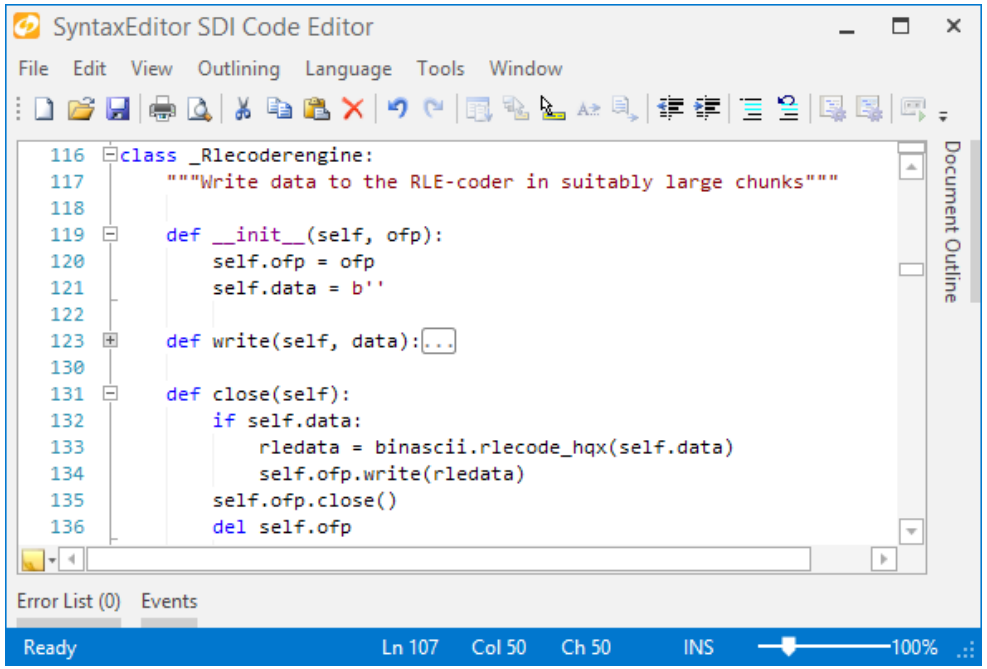
- Easy to learn and use!
- Many pre-built modules with functions and data structures
- Possibility to add your own modules

PYTHON IS ANOTHER PROGRAMMING LANGUAGE

Python is a programming language.

Like every language we have:

- Variables
- Conditional statements
- Data structures
- Input/output



WHY DOES PYTHON MATTERS?

PYTHON STORE

Python is like a smartphone

- It has its own store
- To download new modules
- Graphic, scientific, networking
- Python store is called PIP

PIP PACKAGE MANAGER

- pip stand for "Pip Installs Packages" or "Pip Installs Python".
- pip is a package management system used to find, install and manage Python packages.
- Many packages can be found in the Python Package Index (PyPI).
  - This is a repository for Python.
  - There are currently 70000 packages.
  - https://pypi.python.org/pypi.
- You can use pip to find packages in Python Package Index (PyPI) and to install them.

BUILDING BLOCKS: MODULES & FUNCTIONS

Module:

- A file with Python code. A python file.
- The file name is the module name with the suffix .py appended (module.py).
- A module can define functions, classes, variables ...

Package: several python modules all together in a directory, accompanied with a file named __init__.py. The file __init__.py can be empty.

Function:

- A function returns a value. Call a function passing arguments.
- There are many built-in functions. You can also create your own functions.
- A function is defined once and can be called multiple times.

BUILDING BLOCKS: CLASS AND METHODS

Class:

- Classes define objects.
- Call a class passing arguments. The returned value is an object. So each instance of a class is an object.
- A class defines functions available for this object (in a class, these functions are called methods)

Method:

- A method is a function defined in a class.
- To call a method, we first need to create an instance of the class. Once you have an instance of a class, you can call a method for this object.

HELLO WORLD

hello.py

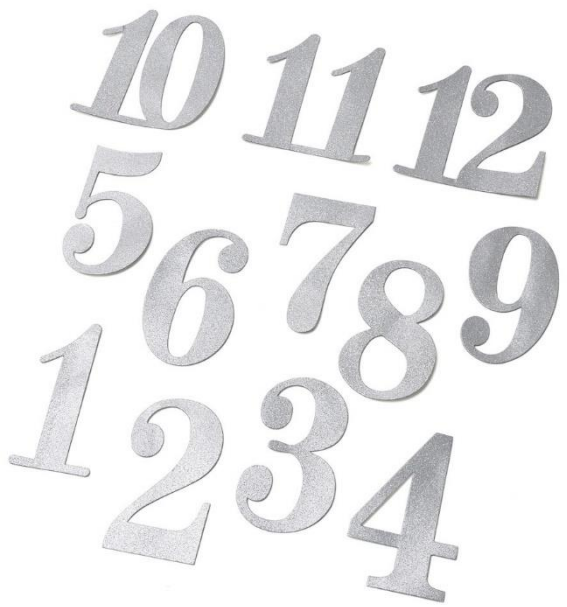
```
print("Hello, World!")
```

GETTING PYTHON FOR UBUNTU

```
root@ubuntu:~# python
The program 'python' can be found in the following packages:
 * python-minimal
 * python3
Try: apt install <selected package>

root@ubuntu:~# apt-get install python
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib
python-minimal python2.7 python2.7-minimal
Suggested packages:
  python-doc python-tk python2.7-doc binutils
The following NEW packages will be installed:
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib python
python-minimal python2.7 python2.7-minimal
0 upgraded, 7 newly installed, 0 to remove and 263 not upgraded.
Need to get 3,877 kB of archives.
After this operation, 16.6 MB of additional disk space will be
used.
Do you want to continue? [Y/n] y
...

root@ubuntu:~# python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

INTEGERS

```
>>> a = 5
>>> b = 3

>>> print (a)
5
>>> print (a+b)
8

>>> a
5
>>> a+b
8

>>> c = a + b
>>> print c
8

>>> d=10
>>> type(d/a)
<type 'int'>
>>> type(d/a*1.0)
<type 'float'>
```



- Integers are numbers
- You can print variables
- You can add variables
- Division between integers gives an integer
- Division involving a float number gives a float

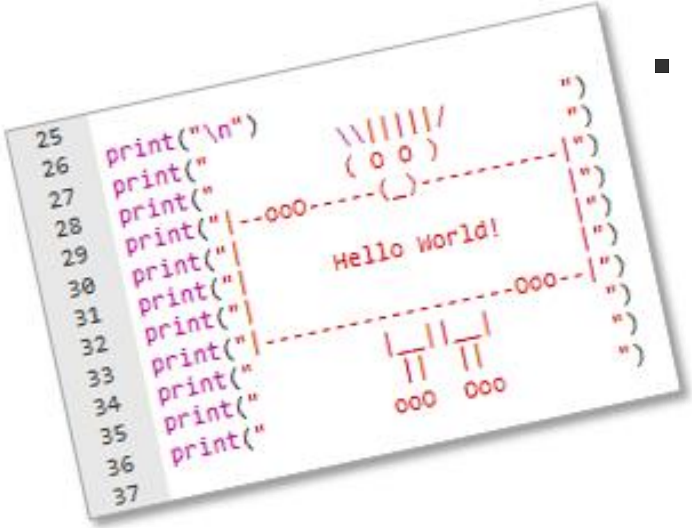
PLAYING WITH INTEGERS

```
>>> a = 5
>>> b = 3

>>> a + b
8
>>> a - b
2
>>> a * b
15
>>> a / b
1

>>> a / b * 1.0
1.0
>>> float(a / b)
1.0
>>> a / float(b)
1.6666666666666667

>>> a ** b
125
>>> a % b
2
```

- Add, subtract, multiply, divide integers
- Same can be done with floats
- “a ** b” means “a to the power of b”
- We also have the module operation

STRINGS

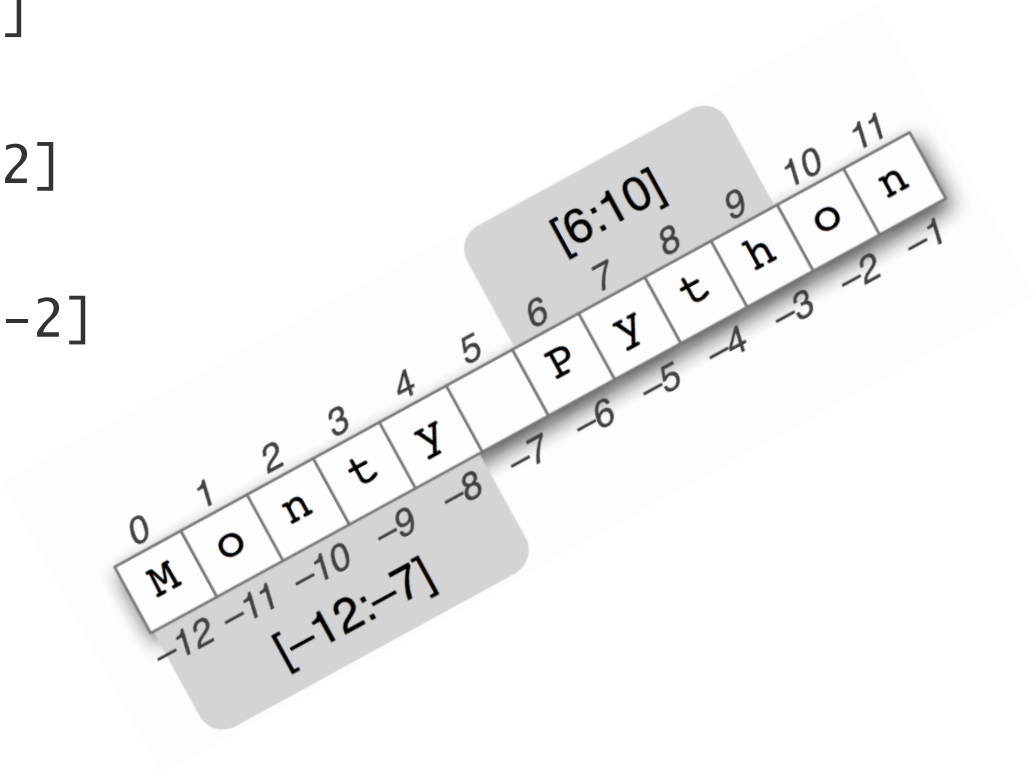
```
>>> a = "string"
>>> a
'string'
>>> print a
string

>>> b = "string\nstring2"
>>> b
'string\nstring2'
>>> print b
string
string2

>>> a = "hello"
>>> b = " "
>>> c = "world"
>>> print a+b+c
hello world
```



- String is a sequence of characters
- String is also an array
- Strings can be concatenated
- Strings can contain special characters like “\n”, newline

MANIPULATING STRINGS

```
>>> a = "table“

>>> a[1]
'a‘

>>> a[:2]
'ta‘

>>> a[:-2]
'tab‘

>>> a[1:-2]
'ab'
```



- String is an array
- Can access a specific character
- Can extract substrings

CONVERTING NUMBERS

```
>>> a = 128

>>> str(a)
'128‘

>>> hex(a)
'0x80‘

>>> bin(a)
'0b10000000'
```

- Integers can be converted into strings
- Integers can be converted into hexadecimal
- Integers can be converted into binary

CONCATENATING STRINGS AND NUMBERS

```
>>> a = 3
>>> b = "number “

>>> type(a)
<type 'int'>
>>> type(b)
<type 'str'>

>>> c = b + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> c = b + str(a)

>>> c
'number 3'
>>> type(c)
<type 'str'>
```

- Strings with strings!
- Cannot concatenate strings and numbers
- Need to convert numbers to strings first!

BOOLEAN AND COMPARISONS

```
>>> a = 192
>>> b = 10

>>> a==b
False

>>> a!=b
True

>>> a>b
True

>>> a<=b
False
```

- Booleans operators available
- Check if two variables are identical or not
- Or if one is bigger than the other

CHECKING SUBSTRINGS

```
>>> a = "oceania“

>>> type(a)
<type 'str'>

>>> 'oce' in a
True

>>> 'qwz' in a
False

>>> 'nia' not in a
False
```

- Check if a substring is inside a string
- Check if a substring is not inside a string

STRINGS AND CASE

```
>>> a = "lower"
>>> a.upper()
'LOWER'

>>> b = "UPPER"
>>> b.lower()
'upper'
```

- String case can be changed
- To lowercase
- Or to uppercase

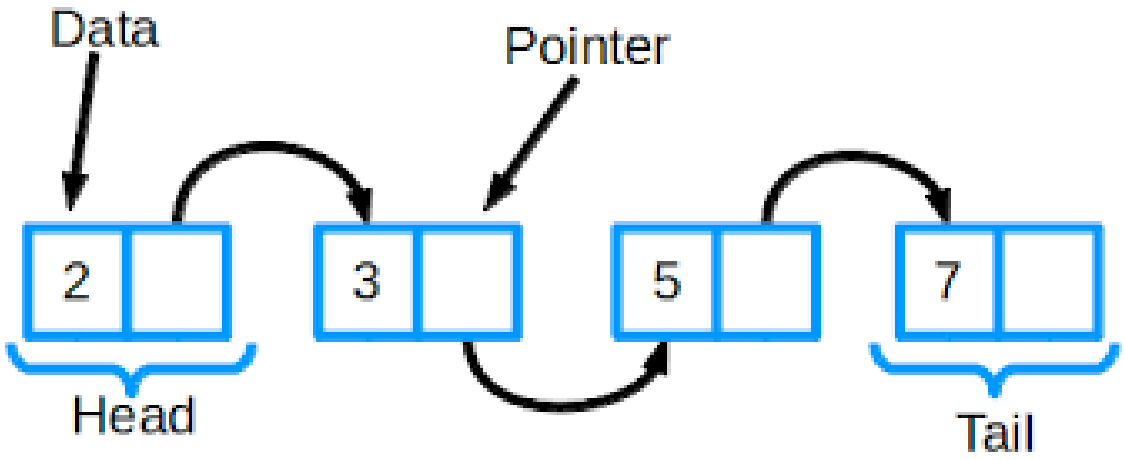
COMMENTING PYTHON

```
>>> # this is a comment

>>> domain="jnp.net" #other comment
>>> domain
'juniper.net‘

>>> print domain
juniper.net
>>>
```

- Use “#” to add comments
- Comments are not executed
- They are there to make code more readable

LISTS

- A collection of items
- Items are ordered
- Items separated by commas and enclosed within square brackets ([])
- A list is iterable: a “for loop” iterates over its items



CREATING A LIST

```
>>> a = []

>>> type(a)
<type 'list'>

>>> a.append(30)
>>> a.append("s")

>>> a
[30, 's']
```

- Create an empty list
- Add new elements
- Elements are added at the end of the list
- List can include a mix of elements (strings, integers, float, other lists)

MIXING OBJECTS WITHIN A LIST

```
>>> a
[30, 's', 3.0]

>>> type(a[0])
<type 'int'>
>>> type(a[1])
<type 'str'>
>>> type(a[2])
<type 'float'>

>>> a[:-1]
[30, 's']

>>> range(0,5)
[0, 1, 2, 3, 4]
>>> type(range(0,1))
<type 'list'>
```

- List can contain multiple variable types
- You can extract sublists
- There is a built-in function called “range” which creates a list of integers

LOOKING FOR ELEMENTS WITHIN A LIST

```
>>> a
[30, 's', 3.0]

>>> 30 in a
True

>>> s in a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 's' is not defined

>>> "s" in a
True
>>> 3 in a
True
>>> 3.0 in a
True

>>> a.index('s')
1
```

- You can verify whether an element is within a list or not

PLAYING WITH LISTS

```
>>> a
['a', 'b', 'b', 'c', 'd']

>>> a.append("e")
>>> a
['a', 'b', 'b', 'c', 'd', 'e']

>>> a.insert(2,"f")
>>> a
['a', 'b', 'f', 'b', 'c', 'd', 'e']

>>> a.count("b")
2

>>> a.sort()
>>> a
['a', 'b', 'b', 'c', 'd', 'e', 'f']

>>> a.reverse()
>>> a
['f', 'e', 'd', 'c', 'b', 'b', 'a']
```

- Lists have interesting built-in methods
- Append elements at the bottom
- Insert elements in a specific position
- Count the number of occurrences of a value
- Sort list in ascending order
- Sort list in descending order

SPLITTING AND JOINING STRINGS

```
>>> s = "this is a string“

>>> l = s.split(" ")
>>> l
['this', 'is', 'a', 'string']

>>> a=" ".join(l)
>>> a
'this is a string'
```

- String can be split
- Split is performed against a specified character
- Result is a list

DICTIONARIES

```
>>> a = {}
>>> type(a)
<type 'dict'>
>>> a['to']=21
>>> a['ge']=13
>>> a['na']=2
>>> a['rm']=5
>>> a
{'to': 21, 'ge': 13, 'rm': 5, 'na': 2}

>>> a.keys()
['to', 'ge', 'rm', 'na']
>>> a.values()
[21, 13, 5, 2]
>>> a['to']
21

>>> a['mi']=18
>>> del(a['na'])
>>> a
{'to': 21, 'ge': 13, 'mi': 18, 'rm': 5}
```

- Dictionary is a collection of key:value pairs
- Keys must be unique
- Values can be repeated
- You can extract a list of keys
- You can extract the list of values
- New key:value pair can be added
- Elements can be removed

FOR CYCLES

```
>>> l
['mario', 'luca', 'paolo']
>>> for x in l:
...     print "Name is " + x
...
Name is mario
Name is luca
Name is paolo


>>> l = range(1,5)
>>> for x in l:
...     print "set interface xe-0/0/%s disable" %x
...
set interfaces xe-0/0/1 disable
set interfaces xe-0/0/2 disable
set interfaces xe-0/0/3 disable
set interfaces xe-0/0/4 disable
```

- For cycles are used to perform an iteration
- You can iterate over a list
- For can be used to print all the elements within a list
- For is useful to speed up time consuming and repetitive tasks

BROWSING DICTIONARIES

```
>>> a
{'to': 21, 'ge': 13, 'mi': 18, 'rm': 5}

>>> for x in a:
...     print x
...
to
ge
mi
Rm

>>> for x in a:
...     print a[x]
...
21
13
18
5
```

- Use for to print dictionaries
- You can print keys
- You can print values
- Alternatively, we told we have special functions to obtain keys/values lists

WHILE LOOPS

```
>>> x = 0

>>> while x < 5:
...     print x
...     x+=1
...

0
1
2
3
4
```

- While is similar to for cycles
- While body is performed as long as the condition is valid
- Be sure the condition will be false at a certain point
- The risk is to fall into an infinite loop!

IF STATEMENT

```
>>> light="green“

>>> if light == "red":
...     print "STOP!"
... elif light == "yellow":
...     print "SLOW DOWN!"
... else:
...     print "GO!!!"
...

GO!!!
```

- If/then/else is used to perform a task based on particular conditions

MIXING FOR AND IF

```
>>> l
['mx80-rm', 'mx240-rm', 'mx80-mi',
'mx960-mi', 'ex2200-rm']

>>> for x in l:
...     if "rm" in x:
...         print x
...

mx80-rm
mx240-rm
ex2200-rm
```

- Of course you can mix different building blocks
- A for can have an if statement within its body

LOOKING FOR A DEVICE

```
>>> my_devices_list
['172.30.108.11', '172.30.108.133', '172.30.108.133', '172.30.108.14’]

>>> '172.30.108.14' in my_devices_list
True

>>> if '172.30.108.14' in my_devices_list:
       print "172.30.108.14 was found in my_devices_list"
     else:
       print "172.30.108.14 was not found in my_devices_list "


172.30.108.14 was found in my_devices_list
```

FUNCTIONS

```
root@ubuntu:~# cat first.py
#! /usr/bin/python
import sys

def square(x):
    return int(x)**2
def cube(x):
    return int(x)**3

def main():
    n=sys.argv[1]
    s=square(n)
    c=cube(n)
    print s
    print c

if __name__ == "__main__":
    main()

root@ubuntu:~# python first.py 3
9
27
```

- We can define custom functions
- They can receive input parameters and compute an output value
- We can also pass command line arguments when running a python script
- In order to read and use command line argument we need to import the sys module

CREATING A CLASS

```
root@ubuntu:~# cat employee.py
#! /usr/bin/python

class employee:

    def __init__(self, n, a, s):
        self.n=n
        self.a=a
        self.s=s

    def get(self):
        print "Name: " + str(self.n) + ",
age: " + str(self.a) + ", salary: " +
str(self.s) + "$."

    def money(self, r):
        self.s+=r

    def birthday(self):
        self.a+=1
```

- Class is an object
- A class has its own variables
- Use “self” to reference object variables
- Class can have its own methods

USING PYTHON CLASSES

```
root@ubuntu:~# cat obj.py
#! /usr/bin/python
from employee import employee

def main():
    paul = employee('paul',23,30000)
    laura = employee('laura',21,40000)

    laura.get()
    paul.get()

    laura.birthday()
    laura.money(5000)
    paul.money(3000)

    laura.get()
    paul.get()

if __name__ == "__main__":
    main()
```

- We can create multiple objects referencing the same class
- They are independent objects
- We can call methods to modify class info or print class info

RUNNING PYTHON WITH CLASSES

```
root@ubuntu:~# python obj.py
Name: laura, age: 21, salary: 40000$.
Name: paul, age: 23, salary: 30000$.
Name: laura, age: 22, salary: 45000$.
Name: paul, age: 23, salary: 33000$.
```

- Script creates 2 employees
- We see employee status before and after updates
- Updates are method calls

READING AN ENTIRE FILE

```
>>> f=open("devices_ex.txt","r")

>>> f
<open file 'devices_ex.txt', mode 'r' at 0x7f39d96c8540>
>>> type(f)
<type 'file'>

>>> f.name
'devices_ex.txt'
>>> f.closed
False
>>> f.mode
'r‘

>>> s=f.read()
>>> type(s)
<type 'str'>
>>> s
'mx80\nex2200\nsrx320\nqfx5100\n'
>>> print s
mx80
ex2200
srx320
qfx5100

>>> f.close()
>>> f.closed
True
```

- To open files use the “r” option, read
- You can check active file mode
- You can check whether the file is open or not
- You can read the whole file using function “read()”
- The whole file is copied into a string
- Each line terminates with a “\n”

READING A FILE LINE BY LINE

```
>>> f=open("devices_ex.txt","r")
>>> s=f.readline()
>>> s
'mx80\n'
>>> s=f.readline()
>>> s
'ex2200\n'
>>> s=f.readline()
>>> s
'srx320\n'
>>> s=f.readline()
>>> s
'qfx5100\n'
>>> s=f.readline()
>>> s
''
```

- Alternatively you can have python reading the file line by line
- Readline() function never gives you an error as long as the file is available
- After we reached the end of the file, calling again the readline() function will result in an empty string!

A POSSIBLE WAY TO READ A FILE

```
>>> f=open("devices_ex.txt","r")
>>> line=f.readline()
>>> while line:
...     print line.strip()
...     line = f.readline()
...
mx80
ex2200
srx320
qfx5100
```

- This is a possible general schema to read an entire file
- We use a while block
- While condition is valid as long as readline gives a non empty string
- Condition will become flase when line will be empty; this happens when file is terminated!

WRITING A FILE

```
root@ubuntu:~# python
>>> f=open("w.txt","w")
>>> f.write("AAA\n")
>>> f.close()
>>> exit()
root@ubuntu:~# cat w.txt
AAA

root@ubuntu:~# python
>>> f=open("w.txt","w")
>>> f.write("BBB\n")
>>> f.close()
>>> exit()
root@ubuntu:~# cat w.txt
BBB

root@ubuntu:~# python
>>> f=open("w.txt","a")
>>> f.write("CCC\n")
>>> f.close()
>>> exit()
root@ubuntu:~# cat w.txt
BBB
CCC
```

- We can write to files as well
- If file does not exist, python creates it automatically
- When mode “w” is used, if file already existed, it is completely overwritten
- When mode “a” is used, if file already existed, new content is appended at the bottom of the file

SPLITTING LINES

```
>>> f=open("devices_ex.txt","r")
>>> s=f.read()
>>> l=s.splitlines()
>>> l
['mx80', 'ex2200', 'srx320', 'qfx5100']
>>> for x in l:
...     print x
...
mx80
ex2200
srx320
qfx5100
```

- Use splitlines to break a string into a list based on the “\n” character

TWO POSSIBLE APPROACHES

```
>>> f=open("devices_ex.txt","r")
>>> l=f.read().splitlines()
>>> l
['mx80', 'ex2200', 'srx320', 'qfx5100']

>>> f=open("devices_ex.txt","r")
>>> l=f.readlines()
>>> l
['mx80\n', 'ex2200\n', 'srx320\n', 'qfx5100\n']
>>> for x in range(0,len(l)):
...     l[x]=l[x][:-1]
...
>>> l
['mx80', 'ex2200', 'srx320', 'qfx5100']
```

- These are 2 alternative ways to accomplish the same task
- This teaches us how, with python, not to rely just on “standard” functions but to make us of “custom” functions that are built to reduce/hide the complexity

COMMAND LINES ARGUMENTS

```
root@ubuntu:~/python# cat argv.py
#! /usr/bin/python

import sys

i=0

for x in sys.argv:
    print "Argument " + str(i) + " : " + str(x)
    i+=1

root@ubuntu:~/python# python argv.py cane 4.0 2
Argument 0 : argv.py
Argument 1 : cane
Argument 2 : 4.0
Argument 3 : 2
```

- Command line arguments are variables passed when invoking the script
- Importing the sys module is required
- First argument is always the script name
- Arguments are stored into a list called sys.argv

ENRICHING PYTHON

Python allows you to import modules to reuse code.

- Good programmers write good code; great programmers reuse/steal code ☺
- Importing a module is done without using the .py extension

Anyone can create modules for private uses or to share with community

Some very nice Python modules for network engineers:

- netaddr: a Python library for representing and manipulating network addresses
- re: regular expressions
- requests: rest api manipulation
- jinja2: generate documents based on templates
- Yaml: “users to programs” communication (to define variables)
- PyEZ: Python library to interact with Junos devices

WHERE TO FIND PYTHON PACKAGES

- Python looks for its modules and packages in $PYTHONPATH.

```
>>> pprint(sys.path)
['',
 '/usr/lib/python2.7',
 '/usr/lib/python2.7/plat-x86_64-linux-gnu',
 '/usr/lib/python2.7/lib-tk',
 '/usr/lib/python2.7/lib-old',
 '/usr/lib/python2.7/lib-dynload',
 '/usr/local/lib/python2.7/dist-packages',
 '/usr/lib/python2.7/dist-packages']
```

GETTING A NEW PACKAGE

```
root@ubuntu:~# apt-get install python-pip
...

root@ubuntu:~# python
>>> import netaddr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named netaddr
>>> exit()

root@ubuntu:~# pip install netaddr
Collecting netaddr
  Downloading
https://files.pythonhosted.org/packages/ba/97/ce14451a9fd7bdb5a397abf99b24a1a6bb7a1a440b019bebd2e9a0dbec74/netaddr-
0.7.19-py2.py3-none-any.whl (1.6MB)
    100% |████████████████████████████████| 1.6MB 238kB/s
Installing collected packages: netaddr
Successfully installed netaddr-0.7.19

root@ubuntu:~# python
>>> import netaddr
>>>
```

CREATING AN IP ADDRESS

```
>>> from netaddr import IPAddress
>>> ip=IPAddress('192.168.1.2')
>>> type(ip)
<class 'netaddr.ip.IPAddress'>
>>> print ip
192.168.1.2

>>> ip.version
4
>>> ip.is_private()
True
>>> ip.is_unicast()
True
>>> ip.is_multicast()
False

>>> ip.bits()
'11000000.10101000.00000001.00000010‘

>>> ip
IPAddress('192.168.1.2')
>>> ip+1
IPAddress('192.168.1.3')
>>> ip+255
IPAddress('192.168.2.1')
```

- Netaddr library has an object to create IP addresses
- The object has several functions
  - Version
  - Unicast or multicast
  - Private address
- We may “explode” the IP 4 bytes
- We can perform operations, for example to know the next 10th IP address

CREATING A NETWORK

```
>>> from netaddr import IPNetwork
>>> net = IPNetwork('192.168.1.0/24')

>>> print net
192.168.1.0/24
>>> net[0]
IPAddress('192.168.1.0')
>>> net[-1]
IPAddress('192.168.1.255')

>>> net.netmask
IPAddress('255.255.255.0')
>>> net.hostmask
IPAddress('0.0.0.255')
>>> net.network
IPAddress('192.168.1.0')
>>> net.broadcast
IPAddress('192.168.1.255')
>>> net.prefixlen
24
>>> net.is_private()
True

>>> net.next()
IPNetwork('192.168.2.0/24')
>>> net.previous()
IPNetwork('192.168.0.0/24')
```

- Similarly, we have an object for Networks
- We can obtain
  - Broadcast address
  - Netmask and prefix length
  - Hostmask
- We can compute next or previous subnets

PRINTING NETWORK MEMBERS

```
>>> net = IPNetwork('192.168.1.0/30')

>>> l=[]
>>> for x in net:
...     l.append(x)
...

>>> l
[IPAddress('192.168.1.0'),
IPAddress('192.168.1.1'),
IPAddress('192.168.1.2'),
IPAddress('192.168.1.3')]
```

- We can also print all the addresses belonging to a network

WHAT'S YAML?

YAML stands for "Yaml Ain't Markup Language"

Yaml is human-readable language.

- Less markup than XML.
- A superset of JSON.

Used for "users to programs" communication

- For users to read/change data.
- Used to communicate with program.
- Designed to translate to structures which are common to various languages (cross language: Python, Perl, Ruby, etc).
- Used to define variables value.

YAML BASICS

- Yaml files use a .yaml or .yml extension
- Yaml documents begin with three dashes `---`
- Comments begin with `#`
- Strings are unquoted
- Indentation with one or more spaces
  - never with tabulations
- Lists: one member per line.
  - Hyphen + space for each item.
- Keys are separated from values by a colon + space.

YAML LIST

```
root@ubuntu:~# cat list.yaml
#this is a yaml list
---
    - 192.168.100.2
    - 192.168.100.29
    - 192.168.100.54
```

- This is a sample YAML file
- This represents a list

GETTING PYYAML

```
root@ubuntu:~# pip install pyyaml
Collecting pyyaml
  Downloading
https://files.pythonhosted.org/packages/4a/85/db5a2df477072b2902b0eb892feb37d88ac635d36245a72a6a69b23b383a/PyYAML-
3.12.tar.gz (253kB)
    100% |████████████████████████████████| 256kB 1.5MB/s
Building wheels for collected packages: pyyaml
  Running setup.py bdist_wheel for pyyaml ... done
  Stored in directory: /root/.cache/pip/wheels/03/05/65/bdc14f2c6e09e82ae3e0f13d021e1b6b2481437ea2f207df3f
Successfully built pyyaml
Installing collected packages: pyyaml
Successfully installed pyyaml-3.12
You are using pip version 8.1.1, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

LOADING A YAML LIST INTO PYTHON

```
>>> import yaml

>>> f=open("list.yaml","r")
>>> s=f.read()

>>> v=yaml.load(s)

>>> v
['192.168.100.2', '192.168.100.29',
'192.168.100.54']
>>> type(v)
<type 'list'>
```

- We can load a YAML list into python
- In python it appears as a list
- Then we can use it as any other list
- We need to import the YAML module

YAML DICTIONARY

```
root@ubuntu:~# cat dict.yaml
#this is a dictionary in yaml
---
interfaces:
    - ge-0/0/1
    - ge-0/0/3
vlan:
    - eng
    - it
    - guest
```

- YAML can also model dictionary
- We list keys
- and for each key, a list of values

LOADING A YAML DICTIONARY INTO PYTHON

```
>>> import yaml

>>> f=open("dict.yaml","r")
>>> s=f.read()

>>> v=yaml.load(s)
>>> type(v)
<type 'dict'>

>>> v.keys()
['interfaces', 'vlan']
>>> v

{'interfaces': ['ge-0/0/1', 'ge-0/0/3'],
'vlan': ['eng', 'it', 'guest']}
>>> v['interfaces'][1]
'ge-0/0/3'
```

- We can use python to import YAML dictionaries into python dictionaries
- Once it is loaded, you can use the new object like any other dictionary

JINJA2

- Jinja2 is a Python package used to generate documents based on templates.
- There are other templating engines for Python: jinja2 is simple, rich, stable and widely used.
- Jinja2 files use a .j2 file extension
- Variables are marked in the template
  - use a {{ variable-name }} syntax.
- Supports some control structures (if and for).
  - use a {% ... %} syntax.
- We will use Jinja2 to handle junos templates

GETTING JINJA2

```
root@ubuntu:~# pip install jinja2
Collecting jinja2
  Downloading
https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca76f32659220a731/Jinja2-
2.10-py2.py3-none-any.whl (126kB)
    100% |████████████████████████████████| 133kB 624kB/s
Collecting MarkupSafe>=0.23 (from jinja2)
  Downloading
https://files.pythonhosted.org/packages/4d/de/32d741db316d8fdb7680822dd37001ef7a448255de9699ab4bfcbdf4172b/MarkupSaf
e-1.0.tar.gz
Building wheels for collected packages: MarkupSafe
  Running setup.py bdist_wheel for MarkupSafe ... done
  Stored in directory: /root/.cache/pip/wheels/33/56/20/ebe49a5c612fffe1c5a632146b16596f9e64676768661e4e46
Successfully built MarkupSafe
Installing collected packages: MarkupSafe, jinja2
Successfully installed MarkupSafe-1.0 jinja2-2.10
You are using pip version 8.1.1, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

RENDERING JINJA2

```
>>> from jinja2 import Template

>>> gc = Template("set interfaces {{interface}}
disable")
>>> type(gc)
<class 'jinja2.environment.Template'>

>>> s=gc.render(interface="ge-0/0/1")
>>> s
u'set interfaces ge-0/0/1 disable'

>>> s=str(gc.render(interface="ge-0/0/1"))
>>> s
'set interfaces ge-0/0/1 disable‘

>>> gc = Template("set interfaces {{interface}} unit
{{unit}}")
>>> s=str(gc.render(interface="ge-0/0/1",unit=20))
>>> s
'set interfaces ge-0/0/1 unit 20'
```

- First we need to create a Template object
- Template object contain Jinja2 strings
- We use the render function to replace jinja2 generalized values with actual values

JINJA2 TO GENERALIZE CONFIGURATION

```
root@ubuntu:~# cat setex.txt
set interfaces {{iface}} unit {{u}} family inet
address {{ip}}
set inerfacess {{iface}} mtu {{size}}

root@ubuntu:~# python
>>> from jinja2 import Template

>>> f=open("setex.txt","r")
>>> s=f.read()
>>> gc = Template(s)

>>> print gc.render(iface="ge-0/0/2", u=15,
size=1476)
set interfaces ge-0/0/2 unit 15 family inet address
set inerfacess ge-0/0/2 mtu 1476

>>> print gc.render(iface="xe-3/0/2", u=11,
size=1500)
set interfaces xe-3/0/2 unit 11 family inet address
set inerfacess xe-3/0/2 mtu 1500
```

- We may call the render function multiple times with different values
- This is the base concept to automate the creation of multiple similar statements
- For example, configuring multiple interfaces

ADVANCED JINJA2 SYNTAX

```
root@ubuntu:~# cat setlist.txt
{%- for interface in interfaces_list %}
set interfaces {{ interface }} unit 0 family ethernet-switching port-mode access vlan members {{
vlan_name }}
{%- endfor %}

root@ubuntu:~# python>>> from jinja2 import Template
>>> f=open("setlist.txt","r")
>>> s=f.read()
>>> t=Template(s)
>>> print t.render(interfaces_list=['ge-0/0/1','xe-0/0/0'], vlan_name='v10')

set interfaces ge-0/0/1 unit 0 family ethernet-switching port-mode access vlan members v10
set interfaces xe-0/0/0 unit 0 family ethernet-switching port-mode access vlan members v10
```

- Jinja2 files can contain for cycles (or if statements)
- Render function can include ists

AUTOMATING VPN PROVISION: YAML & JINJA2

```
root@ubuntu:~# cat vpn.yaml
instances:
    - name: VPN1
      type: vrf
      rd: 192.168.100.1:21
      rt: target:64555:601
    - name: VPN2
      type: vrf
      rd: 192.168.100.1:22
      rt: target:64555:602

root@ubuntu:~# cat vpn.j2
{%- for x in instances %}
set routing-instances {{ x.name }} instance-type {{ x.type }}
set routing-instances {{ x.name }} route-distinguisher {{ x.rd }}
set routing-instances {{ x.name }} vrf-target {{ x.rt }}
{%- endfor %}
```

AUTOMATING VPN PROVISION: PYTHON

```
>>> import yaml
>>> from jinja2 import Template

>>> f=open("vpn.yaml","r")
>>> s=f.read()
>>> v=yaml.load(s)
>>> f.close()

>>> f=open("vpn.j2","r")
>>> s=f.read()
>>> t=Template(s)
>>> conf=t.render(v)

>>> print conf
set routing-instances VPN1 instance-type vrf
set routing-instances VPN1 route-distinguisher 192.168.100.1:21
set routing-instances VPN1 vrf-target target:64555:601
set routing-instances VPN2 instance-type vrf
set routing-instances VPN2 route-distinguisher 192.168.100.1:22
set routing-instances VPN2 vrf-target target:64555:602
```

everywhere