Using Python at LRZ

Ferdinand.Jamitzky@LRZ.de
Python Intro
On each node there is a system python installed. Don't use it!

Use the module system:

$ module avail python
------------------------------- /lrz/sys/share/modules/files/tools -------------------
python/2.7_anaconda_nompi python/2.7_intel(default) python/3.5_intel

$ module load python

$ python
Python 2.7.13 (default, Jan 11 2017, 10:56:06) [GCC] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
LRZ uses the conda package manager for python libraries. In the default module only a minimal set of libraries is provided. You have to generate your own environment to get more.

$ module load python

$ conda create --n py36 python=3.6

$ source activate py36

$ conda install scipy=0.15

$ conda list
Beautiful is better than ugly
Explicit is better than implicit
Simple is better than complex
Complex is better than complicated
Readability counts

“There should be one (and only one) obvious way to do it“

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%“ (Donald Knuth)
Python in a nutshell
Python Syntax

- basic syntax
  - import, for, if, while, list comprehensions
- advanced syntax
- builtin data types
  - lists, tuples, arrays, sets
  - dicts
  - strings
if x==0:
    print "x is zero"
elif x>0 and x<1:
    print "x between 0 and 1"
else:
    print "x = ", x

“Python is executable pseudocode. Perl is executable line noise.” (– Old Klingon Proverb)
basic rules of the game

- indentation matters!
- file type matters (*.py)!
- directory hierarchy matters!
- comments are #
- lists start from 0

```
$ python
>>> import myfile
>>> import mymod
>>> myfile.myfunc()
hello
>>> mymod.myfunc()
world
```

$ ls
myfile.py
mymod/
mymod/__init__.py

myfile.py:
def myfunc():
    print("hello")

__init__.py:
def myfunc():
    print("world")
Python has the following number types:
- int, long, float, complex
- del var

Strings
- "this", 'this', """"this"""", '''this''', u'this', b'this' (python3)
- interpolation: "one plus %i = %s" % (1,"two")

Lists and tuples
- a=[1,2,3] is a list, b=(1,2,3) is a tuple (immutable)
- a+a, a[0:2], a[-1], a[0:]

Dictionaries
- a={ 'one': 1, 'two': "zwei"} is a dict, a['one']
import lib as name
from lib import n as n

if condition:
    elif condition:
        else:

for iterator in list:
    pass
    break
    continue

[expr for it in list if cond]

while condition:
    def function:
        """doc string""
        return value

class name:
    def __init__(self):
        def method(self):
Keywords (less than 10% of python code)

- raise name
- lambda var: expression
- try:
- except name:
- finally:
- with expression as var:
- async def fun -> ann:
  - assert condition
  - yield value
  - yield from generator
  - await expression
- global variable
- nonlocal variable
- @decorator
Python has the following number types:
- int, long, float, complex
- del var

```python
>>> x=0
>>> x=1234567890123456789012345
>>> x**2
1524157875323883675049533479957338669120562399025
```
basic types

```python
>>> x=1234567890123456789012345
>>> float(x)**12
1.2536598767934103e+289
>>> float(x**12)
1.2536598767934098e+289
>>> x**12
12536598767934098838515598795734462071977276343555841264391863470886000868462247628918940812290412402507934889820704250464446377864110414099084187826638368056804411536204404388409544441384289179095087047608175790842338441544887228788494128120919791295898721196764732642609051396426025390625
```
Imaginary and complex numbers are built in:

```python
>>> 1j**2               #imaginary unit
(-1+0j)

>>> (1+1j)**4            #4th root of -4
(-4+0j)

>>> 1j**1j              # i to the i
(0.20787957635076193+0j)

>>> import cmath

>>> cmath.log(-1)
3.141592653589793j      # pi
```
The import statement, which is used to import modules whose functions or variables can be used in the current program. There are four ways of using import:

```python
>>> import numpy
>>> from numpy import *
>>> import numpy as np
>>> from numpy import pi as Pie
```
python2 has byte strings, python3 has Unicode strings
- "this", 'this', """"this"""", '''this''', u'this', b'this'
- string interpolation (masks)
>>> "one plus %i = %s" % (1,"two")
- indexing strings: a="1234"
>>> print a[0] -> 1
>>> print a[0:] -> 1234
>>> print a[0:-1] -> 123
>>> print a[0::2] -> 13
>>> print a[:::-1] -> 4321
>>> print a[-1::-2] -> 42
- **split strings**
  >>> dd="a b c d"
  >>> dd.split()
  ['a', 'b', 'c', 'd']

- **join strings**
  >>> " ".join(['a', 'b', 'c', 'd'])

- **combine both**
  >>> " ".join([ "<"+x"/>" for x in dd.split()])
  '<a/> <b/> <c/> <d/>'
while

x=0.1
n=0

while x>0 and x<10:
    x*=2
    n+=1

if n>1000:
    break

run the loop until the "while" condition is false or the "if" condition is true.
for i in list:
    do_something_with(i)
print result(i)
if cond(i):
    break

loops over a list, prints the result and stops either when the list is consumed or the break condition is fulfilled
• text files
```
dd=open("data.txt").readlines()
```
• print lines
```
[x[:-1] for x in open("data.txt","r").readlines()]
```
• pretty print
```
from pprint import pprint
pprint(dd)
```
• binary files
```
xx=open("data.txt","rb").read()
xx.__class__
```
make script executable:
$ chmod u+x myscript.py

myscript.py:
#!/usr/bin/python
#!/usr/bin/env python2.7
import sys
print "The name of the script: ", sys.argv[0]
print "Number of arguments: ", len(sys.argv)
print "The arguments are: " , str(sys.argv)

in larger scripts use the `argparse` library
• **Lists** are what they seem - a list of values. Each one of them is numbered, starting from zero. You can remove values from the list, and add new values to the end. Example: Your many cats' names.

• **Tuples** are just like lists, but you can't change their values. The values that you give it first up, are the values that you are stuck with for the rest of the program.

• **Dictionaries** are similar to what their name suggests - a dictionary, or aka associative array or key-value store
Simple list:

```python
>>> x=[1,2,3]
>>> x.append("one")
>>> y=x
>>> y[0]=2
>>> x[0]
2
>>> x.append(x)
>>> x
[2, 2, 3, 'one', [...]]
```
tuples are immutable lists

```python
>>> a=(1,2,3)
>>> a[1]=3
-> error
```

reason for tuples: faster access
list comprehensions

- A list is defined by square brackets.
- A list comprehension uses square brackets and for.

```python
>>> x=[1,2,3,4,5]
>>> y=[ i for i in x]

>>> "<br>".join([s.split("\n") for s in open("file.txt").readlines()])

>>> import random.uniform as r
>>> np=1000000
>>> sum([(r(0,1)**2+r(0,1)**2 < 1) for i in range(np)])/np*4.
3.141244
```
dictionaries **aka** associative arrays **aka** key/value stores

```python
>>> a={'one':1, 'two':2.0, 'three':[3,3,3]}

dictionary comprehensions:
```n
```python
>>> {i:i**2 for i in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
```n
```python
>>> a.keys()
```n
```python
>>> a.values()
```
for loops with dicts

you can loop over a dict by:
```python
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
```

or
```python
>>> {'k' + ' ' + v for k, v in knights.items()}
>>> [k + ' ' + v for k, v in knights.items()]
```
arrays

arrays are lists with the same type of elements
there exists a special library for numeric arrays (numpy) which never made it into the official distribution.

ey they serve as an interface to c-code. If you need numerical arrays use the numpy library (see below)
sets are unordered lists. They provide all the methods from set theory like intersection and union. Elements are unique.

```python
g++ x=set((1,2,3,4,1,2,3,4))
g++ x
{1, 2, 3, 4}
g++ x & y
g++ x | y
g++ x-y
g++ x ^ y
```