Package Managers

conda

- conda is a package manager in user space.
- tool to create isolated python installations
- it allows you to use multiple versions of python
- substitutes virtualenv (dead since 2016)
- commercial tool: anaconda
- 2 versions miniconda (free), anaconda (commercial)
- works on linux, MS-win, macOS
- packages are provided by channels (anaconda, conda-forge, bioconda, intel

package managers

python has a plentitude of package managers and package formats (contradicts zen of python), so don’t get confused

- easy_install (dead)
- pip (still alive)
- virtualenv (dead)
- conda (state of the art)
- wheel (official package format PEP427)
- egg (old package format)

conda

```
$ conda create –n my_env python=3.6
$ conda install –c conda-forge scipy=0.15.0
$ conda list
$ conda search numpy
$ conda update –all
$ conda info numpy
```

pip

- simple packages management tool for python
- comes preinstalled with python
- complementary to conda
- packages are called *.whl (wheel)
- easy_install is dead

```
$ pip install SomePackage              # latest version
$ pip install SomePackage==1.0.4       # specific version
$ pip install 'SomePackage>=1.0.4'     # minimum version
$ pip install --upgrade SomePackage    # upgrade
```


lrz
Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

Shells

ipython

the python interactive command line interface was not very comfortable, so ipython was born. It evolved later on to a Web-Interface (jupyter). You can enter even shell commands.

```
$ ipython
Python 3.6.2 |Continuum Analytics, Inc.| (default, Jul 20 2017, 13:51:32)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: pwd
Out[1]: '/home/hpc/pr28fa/a2815ah‘
In [2]: import os; os.getcwd()
Out[2]: '/home/hpc/pr28fa/a2815ah'
```

ipython

ipython is a hybrid between the python cli, a bash shell and macros. It recognizes shell commands (ls, pwd, cp, ..) and macros (magic commands) can be defined by %name or %%name.

```
In [2]: %timeit sum(range(1000))
20.8 µs ± 412 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
In [13]: %%timeit
    ...: x=sum(range(100))
    ...: y=x+1
    ...:
1.52 µs ± 5.34 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

ipython

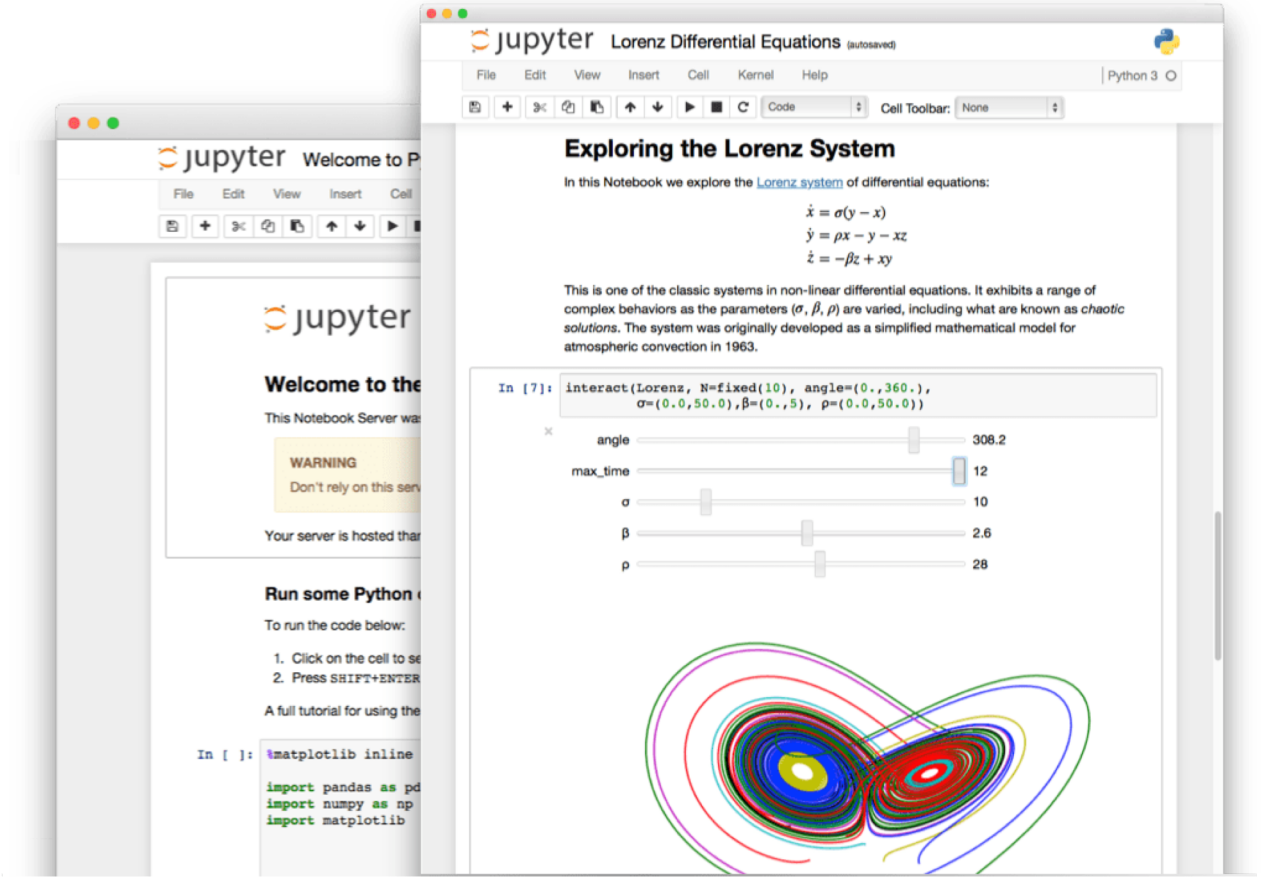
help information can be retrieved by ?command and more detailed information by ??command

```
In [17]: ?pprint
Docstring: Toggle pretty printing on/off.
File:      ~/.conda/envs/py36/lib/python3.6/site-
packages/IPython/core/magics/basic.py

In [16]: ??pprint
Source:
    @line_magic
    def pprint(self, parameter_s=''):
        """Toggle pretty printing on/off."""
        ptformatter = self.shell.display_formatter.formatters['text/plain']
        ptformatter.pprint = bool(1 - ptformatter.pprint)
        print('Pretty printing has been turned',....
```

jupyter

finally ipython evolved into a web-service where you can run any code through a browser interface and even plot.




lrz
Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

Functions

functions: keywords

```
def myfun(a, b=1, c=[1,2], *args):
    return a,b,c,args

>>> myfun(0)
(0,1,[1,2],())
>>> myfun(0,c=2)
(0,1,2,())
>>> myfun(0,1,2,3,4)
(0,1,2,(3,4))
```

functions: lambda functions

```
f1 = lambda x: x+1

def f2(x):
    return x+1

f = lambda *x:x
>>> f(”one”,2,[])
(”one”,2,[])
```

special functions

- function names with leading and trailing underscores are special in python ("magic methods")

```
>>> print(a)
```

is translated to:

```
>>> a.__print__()
```

and

```
>>> a+b
>>> a.__add__(b)
>>> f(x)
>>> f.__call__(x)
```

list comprehensions

- a list is defined by square brackets
- a list comprehension uses square brackets and for

```
>>> 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
```

Xonsh: python+bash

Python / Bash hybrid

```
>>> $(ls -al)
```

$() captures and returns the stdout of the command

You can reuse the result in a python expression

```
>>> [x for x in $(ls -al).split("\n")]
```

Or construct bash expressions from python:

```
>>> x="hello"
>>> y="world"
>>> echo @(x+" "+y)
```

xonsh

Construction of bash pipes:

```
>>> ls -1 | @(lambda a,s: s.read().upper())
```

Or create alias commands:

```
>>> aliases['g'] = 'git status -sb'
```

For more information see:

https://xon.sh/tutorial.html

Advanced Topics

Advanced topics

- try-except
- decorators
- with
- yield
- aspect oriented programming

try except

using try you can catch an exception that would normally stop the program

```
x=range(10)
y=[0]*10
for i in range(10):
  try:
    y[i]=1./x[i]
  except:
    y[i]=0.
```

@decorators

decorators are syntactic sugar for applying a function and overwriting it.

```
@mydecorator
def myfunc():
    pass
```

is the same as:

```
def myfunc():
    pass
myfunc = mydecorator(myfunc)
```

with statement motivation

The with statement allows for different contexts

```
with EXPR as VAR:
    BLOCK
```

roughly translates into this:

```
VAR = EXPR
VAR.__enter__()
try:
    BLOCK
finally:
    VAR.__exit__()
```

with statement examples

You need a context manager (has enter and exit methods)
Examples:

- opening and automatically closing a file

```
with open("/etc/passwd") as f:
    df=f.readlines()
```

- database transactions
- temporary option settings
- ThreadPoolExecutor
- log file on/off
- cd to a different folder and back
- set debug verbose level
- change the output format or output destination

```
with redirect_stdout(sys.stderr):
     help(pow)
```

generators

- range(10000) would generate a list of 10000 number although they would later on not be needed.
- generators to the rescue!!
- only generate what you really need
- new keyword: **yield** (instead of **return**)

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i
...
>>> a=createGenerator()
>>> next(a)
0
```

generator comprehensions

- like list comprehensions, but computed only when needed

```
>>> a=(i**4 for i in range(8))
>>> next(a)
0
>>> next(a)
1
>>> list(a)
[16, 81]
```

Aspect Oriented Programming in python

- AOP is about separating out *Aspects*
- You can switch contexts (like log-file on/off)

```
from contextlib import contextmanager
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

>>> with tag("h1"):
...     print("foo")

<h1>foo</h1>
```