Using Python at LRZ

Ferdinand.Jamitzky@LRZ.de
Python in a nutshell
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)
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)
python has a plenitude 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 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
```
Shells
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
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 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)
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',....
finally ipython evolved into a web-service where you can run any code through a browser interface and even plot.
Functions
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,[])
● 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

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

>>> 
```

```python
>>> "<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
```
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)
Construction of bash pipes:

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

Or create alias commands:

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

```python
x=range(10)
y=[0]*10
for i in range(10):
    try:
        y[i]=1./x[i]
    except:
        y[i]=0.
```
decorators are syntactic sugar for applying a function and overwriting it.

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

is the same as:

```python
def myfunc():
    pass
myfunc = mydecorator(myfunc)
```
The with statement allows for different contexts

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

roughly translates into this:

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

```python
>>> 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

```python
>>> a=(i**4 for i in range(8))
>>> next(a)
0
>>> next(a)
1
>>> list(a)
[16, 81]
```
AOP is about separating out *Aspects*

You can switch contexts (like log-file on/off)

```python
from contextlib import contextmanager

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

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

<h1>foo</h1>
```
Computing and Plotting Libraries
scipy

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis
- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x
array([1, 2, 3])
>>> y = np.arange(10)  # like Python's range, but returns an array
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = np.array([1, 2, 3, 6])
>>> b = np.linspace(0, 2, 4)
# create an array with four equally spaced points starting with 0 and ending with 2.
>>> c = a - b
>>> c
array([ 1.        ,  1.33333333,  1.66666667,  4.        ])
- DataFrame object for data manipulation with integrated indexing.
- Tools for reading and writing data between in-memory data structures and different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of data sets.
- Label-based slicing, fancy indexing, and subsetting of large data sets.
- Data structure column insertion and deletion.
- Group by engine allowing split-apply-combine operations on data sets.
- Data set merging and joining.
- Hierarchical axis indexing to work with high-dimensional data in a lower-dimensional data structure.
- Time series-functionality: Date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging.
```python
>>> import pandas as pd
>>> df = pd.read_csv("ign.csv")
>>> df.head()
>>> df.tail()
>>> df.shape
>>> df.loc[0:5,:]
>>> df.iloc[0:5,:]
>>> df.index
>>> df.loc[:5,['score','release_year']]
>>> df["score"].mean()
>>> df.corr()
>>> r1=df['scores'] > 7
>>> df[r1]
>>> df['score'].plot(kind="hist")
```


```python
import matplotlib.pyplot as plt
import numpy as np

# Data for plotting

tagle = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

# Note that using plt.subplots below is equivalent to using
# fig = plt.figure() and then ax = fig.add_subplot(111)

fig, ax = plt.subplots()

ax.plot(t, s)

ax.set(xlabel='time (s)', ylabel='voltage (mV)',
      title='About as simple as it gets, folks')

ax.grid()

fig.savefig("test.png")

plt.show()
```
Simple spectral analysis

An illustration of the Discrete Fourier Transform using windowing, to reveal the frequency content of a sound signal.

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \quad k = 0, \ldots, N - 1 \]

We begin by loading a datafile using SciPy's audio file support:

```python
In [1]: from scipy.io import wavfile
   : rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's built-in `specgram` routine:

```python
In [2]: %matplotlib inline
   : from matplotlib import pyplot as plt
   : fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
   : ax1.plot(x); ax1.set_title('Raw audio signal')
   : ax2.specgram(x); ax2.set_title('Spectrogram');
```
Data Analysis
**Familiar**: Provides parallelized NumPy array and Pandas DataFrame objects

**Flexible**: Provides a task scheduling interface for more custom workloads and integration with other projects.

**Native**: Enables distributed computing in Pure Python with access to the PyData stack.

**Fast**: Operates with low overhead, low latency, and minimal serialization necessary for fast numerical algorithms

**Scales up**: Runs resiliently on clusters with 1000s of cores

**Scales down**: Trivial to set up and run on a laptop in a single process, even on a smartphone running android

**Responsive**: Designed with interactive computing in mind it provides rapid feedback and diagnostics to aid humans
dask.array

- dask arrays are composed of numpy arrays.
- the subarrays can live in the same process or in another process on a different node.
- dask has a scheduler which distributes the work on a whole cluster if needed.

```python
>>> import dask.array as da
>>> a=da.random.uniform(size=1000, chunks=100)
```
dask.dataframe

- like dask.arrays uses numpy arrays, dask.dataframe uses pandas
- dask.dataframes can be distributed over a cluster of nodes and operations on them are scheduled by the dask scheduler

```python
>>> import dask.dataframe as dd
>>> df=dd.read_csv('2014-*\.csv')
```
```python
>>> a = da.random.uniform(size=1000, chunks=100)
>>> b = a.sum()
>>> c = a.mean() * a.size
>>> d = b - c
>>> d.compute()
```

the computation starts at the last command. If you have a dask cluster then all computations can be distributed to the cluster.
Data Streams
Apache Spark™ is a fast and general engine for large-scale data processing.

text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
  .map(lambda word: (word, 1))
  .reduceByKey(lambda a, b: a+b)
Apache Spark is a fast and general engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing

- written in java
- built on top of Hadoop cluster technology
- language bindings for python, R and scala
- plugs seamlessly into the python ecosystem (scipy, matplotlib, jupyter)
```python
import pandas as pd
import numpy as np

data = pd.read_csv('data/train.csv')
```
Spark: internals

Client/server Architecture

RDD (Resilient Distributed Dataset)
read-only, partitioned collection of records
Spark - programming

- map
- filter
- flatMap
- mapPartitions
- union / intersection
- distinct
- groupByKey, reduceByKey, aggregateByKey
- sortByKey
- join
Machine Learning Packages
Theano:

- numerical computation library for Python
- computations are expressed using a Numpy-esque syntax
- compiled to run efficiently
- CPU or GPU architectures
- Dead since 2017, but still in use
- TensorFlow
- open-source software library
- dataflow programming across a range of tasks
- symbolic math library
- used for machine learning applications
- neural networks
- research and production at Google
- very active
- steep learning curve
# load TensorFlow
>>> import tensorflow as tf
# Initialize two vectors
>>> x = tf.constant([1,2,3,4])
>>> y = tf.constant([5,6,7,8])
# Multiply
z = tf.multiply(x, y)
# Initialize Session and run
>>> with tf.Session() as sess:
    . . . out = sess.run(z)
    . . . print(out)
6
# load TensorFlow
>>> import tensorflow as tf

# Initialize two vectors
>>> x = tf.constant([1,2,3,4])
>>> y = tf.constant([5,6,7,8])

# Multiply
z = tf.multiply(x, y)

# Initialize Session and run
>>> with tf.Session() as sess:
. . . out = sess.run(z)
. . . print(out)
6
Keras is a high-level neural networks API
Running on top of TensorFlow, CNTK, or Theano
Developed with a focus on enabling fast experimentation
Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility)
Supports both convolutional networks and recurrent networks, as well as combinations of the two
Runs seamlessly on CPU and GPU
>>> from keras.models import Sequential
>>> model = Sequential()
>>> from keras.layers import Dense
>>> model.add(Dense(units=64, activation='relu',
                  input_dim=100))
>>> model.add(Dense(units=10, activation='softmax'))
>>> model.compile(loss='categorical_crossentropy',
                 optimizer='sgd', metrics=['accuracy'])
>>> model.fit(x_train, y_train, epochs=5,
            batch_size=32)
>>> classes = model.predict(x_test, batch_size=128)
# resnet50 pretrained application in keras

```python
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np

model = ResNet50(weights='imagenet')
img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
preds = model.predict(x)
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u'tusker', 0.1122357), (u'n02504458', u'African_elephant', 0.061040461)]
```
Parallel and distributed programming
Why?
- You have many independent tasks (easy)
  or
- You want to accelerate single complex task (hard)

Recipe:
Turn the single complex task into many independent simple tasks, but how?
Why?
● You have many independent tasks (easy)
or
● You want to accelerate single complex task (hard)

Recipe:
Turn the single complex task into many independent simple tasks, but how?
LRZ from the system perspective

Cluster

Switch

Fat Tree

Pruned Tree

Island

Node

Accelerator: GPU, FPGA
Core
Sockets
Node
Parallel and Distributed Programming

- multiprocessing
- dask.distributed
- Mpi4py
- Scoop
- Ipython parallel

See also:
https://chryswoods.com/parallel_python/README.html
The standard Python interpreter (called CPython) does not support the use of threads well. The CPython Python interpreter uses a “Global Interpreter Lock” to ensure that only a single line of a Python script can be interpreted at a time, thereby preventing memory corruption caused by multiple threads trying to read, write or delete memory in parallel. Because of the GIL, parallel Python is normally based on running multiple forks of the Python interpreter, each with their own copy of the script and their own GIL.
Embarrassingly parallel

- many independent processes (10 - 100,000)
- no communication between processes
- individual tasklist for each process
- private memory for each process
- results are stored in a large storage medium
Take as example the following script

```bash
#!/bin/bash
source /etc/profile.d/modules.sh
module load python
source activate py36
cd ~/mydir
python myscript.py
```

You can run it by:

```
$ ./myscript.sh
```
Embarrassingly parallel (step-by-step)

Run the script in an interactive slurm shell:

```bash
$ salloc -pmpp2_inter -n1 myscript.sh
```

Change the last line in the script:

```bash
#!/bin/bash
source /etc/profile.d/modules.sh
module load python
source activate py36
cd ~/mydir
srun python myscript.py
```
Run the script in an interactive slurm shell again:

```bash
$ s alloc -pmpp2_inter -n4 myscript.sh
```

You will get 4 times the output of the same run.

```
SLURM_PROCID is set to 0,1,2,3,...
```

Use this variable to select your workload.

```
$ s alloc -pmpp2_inter -n2 srun python -c "import os; os.environ['SLURM_PROCID']"
0
1
```
Run the script as slurm batch job:

```
sbatch -pmpp2_inter -n4 myscript.sh
```

You can put the options inside the slurm file:

```
#!/bin/bash
#SBATCH -pmpp2_inter
#SBATCH -n4
source /etc/profile.d/modules.sh
module load python
cd ~/mydir
srun python myscript.py
```
For serial loads use the serial queue and add options for the runtime:

```
#!/bin/bash
#SBATCH --clusters=serial
#SBATCH --time=01:00:00 # 1hour
source /etc/profile.d/modules.sh
module load python
cd ~/mydir
srun python myscript.py
```

$ sbatch myscript.slurm
If you want to send a large number of jobs then use Job Arrays.

```bash
$ sbatch --array=0-31 myscript.slurm
```

The variable `SLURM_ARRAY_TASK_ID` is set to the array index value. Get it in python via:

```python
>>> os.environ['SLURM_ARRAY_TASK_ID']
```

The maximum size of array job is 1000
SLURM commands

• List my jobs:
  
  $ squeue -Mserial -u <uid>

• Cancel my job
  
  $ scancel <jobid>

• Submit batch job
  
  $ sbatch myscript.slurm

• Run interactive shell
  
  $ salloc -n1 srun --pty bash -i
Ipython and ipcluster

The **ipcluster** command provides a simple way of starting a controller and engines in the following situations:

- When the controller and engines are all run on localhost. This is useful for testing or running on a multicore computer.
- When engines are started using the **mpiexec** command that comes with most MPI implementations.
- When engines are started using the SLURM batch system.
Starting ipcluster:

```
$ ipcluster start -n 4
```

Then start ipython and connect to the cluster:

```
$ ipython
In [1]: from ipyparallel import Client
In [2]: c = Client()
   ...: c.ids
   ...: c[:].apply_sync(lambda: "Hello, world!")
Out[2]: ['Hello, world!', 'Hello, world!', 'Hello, world!', 'Hello, world!']
```
Create a parallel profile:

```
ipython profile create --parallel --profile=slurm
```

cd into ~/.ipython/profile_slurm/ and add the following:

**ipcontroller_config.py:**
```
c.HubFactory.ip = u'*'
c.HubFactory.registration_timeout = 600
```

**ipengine_config.py:**
```
c.IPEngineApp.wait_for_url_file = 300
c.EngineFactory.timeout = 300
```
- ipcluster_config.py:
  ```python
c.IPClusterStart.controller_launcher_class = 'SlurmControllerLauncher'
c.IPClusterEngines.engine_launcher_class = 'SlurmEngineSetLauncher'
c.SlurmEngineSetLauncher.batch_template = """#!/bin/sh
#SBATCH --ntasks={n}
#SBATCH --clusters=serial
#SBATCH --time=01:00:00
#SBATCH --job-name=ipy-engine-
srun ipengine --profile-dir="{profile_dir}" --cluster-id=""
"""
```
Shared Memory (your laptop)

• a few threads working closely together (10-100)
• shared memory
• single tasklist (program)
• cache coherent non-uniform memory architecture aka ccNUMA
• results are kept in shared memory
Multiprocessing allows your script running multiple copies in parallel, with (normally) one copy per processor core on your computer.

One is known as the master copy, and is the one that is used to control all of worker copies.

It is not recommended to run a multiprocessing python script interactively, e.g. via ipython or ipython notebook.

It forces you to write it in a particular way. All imports should be at the top of the script, followed by all function and class definitions.
# all imports should be at the top of your script
import multiprocessing, sys, os
# all function and class definitions must be next

def sum(x, y):
    return x+y

if __name__ == '__main__':
    # You must now protect the code being run by
    # the master copy of the script by placing it

    a = [1, 2, 3, 4, 5]
b = [6, 7, 8, 9, 10]

    # Now write your parallel code... etc. etc.
from multiprocessing import Pool, current_process

def square(x):
    print("Worker %s calculating square of %d" % (current_process().pid, x))
    return x*x

if __name__ == "__main__":
    nprocs = 2

    # print the number of cores
    print("Number of workers equals %d" % nprocs)

    # create a pool of workers
    pool = Pool(processes=nprocs)

    # create an array of 10 integers, from 1 to 10
    a = range(1,11)

    result = pool.map( square, a )
    total = reduce( lambda x,y: x+y, result )

    print("The sum of the square of the first 10 integers is %d" % total)
Multiprocessing futures

- Use futures and a context manager:

```python
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=1) as ex:
    future = ex.submit(pow, 323, 1235)
    print(future.result())
```
Scoop is a developing third-party Python module that supports running parallel Python scripts across clouds, distributed compute clusters, HPC machines etc.

- **conda install scoop** if you are using anaconda python
- **pip install scoop** if you have installed pip
- **easy_install scoop** in all other cases (i.e. if the other two commands don’t work)
from scoop import futures

def product(x, y):
    return x*y

def sum(x, y):
    return x+y

if __name__ == '__main__':
    a = range(1, 101)
    b = range(101, 201)

    results = futures.map(product, a, b)
    total = reduce(sum, results)

    print("Sum of the products equals %d" % total)
Running scoop

- Run this script using the command

```
$ python -m scoop mapreduce.py
```

- You need to use -m scoop so that Scoop has time to set up the distributed cluster before running your script.

```
$ python -m scoop --hostfile hostfile script.py
```
Caveats

Scoop provides a very similar interface as multiprocessing, with the same caveats, requirements and restrictions. For example:

- You must ensure that all use of Scoop is protected within an `if __name__ == '__main__'`

- You must import all modules and declare all functions at the top of your script, before the `if __name__ == '__main__'`

- Scoop does not yet support anonymous (lambda) functions, again because of Python’s poor support for pickling those functions. Hopefully this will change soon.
Message Passing

- many independent processes (10 - 100.000)
- one tasklist for all (program)
- everyone can talk to each other (in principle)
- private memory
- needs communication strategy in order to scale out
- very often: nearest neighbor communication
- beware of deadlocks!
$ mpiexec -n 4 python myapp.py

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
Worker queue

- many independent processes (10 - 100,000)
- central task scheduler (database)
- private memory for each process
- results are sent back to task scheduler
- rescheduling of failed tasks possible
- Start a scheduler which organizes the computing tasks
  
  ```
  $ dask-scheduler
  ```

- dask workers
  
  ```
  $ dask-worker localhost:8786
  $ dask-ssh host.domain
  $ mpirun --np 4 dask-mpi
  $ dask-ec2
  $ dask-kubernetes
  $ dask-drmaa
  ```
● Start a client

```python
>>> from distributed import Client
>>> client = Client('localhost:8786')
```

now all dask operations will be distributed to the scheduler which distributes them to the cluster
```python
>>> a = da.random.uniform(size=1000, chunks=100)
>>> b = a.sum()
>>> c = a.mean() * a.size
>>> d = b - c
>>> d.compute()
```

The computation starts at the last command. If you have a dask cluster then all computations can be distributed to the cluster.
- install qpython
- open pip console
- install dask
- install toolz
- install ipython
Native Programming
Python numerical libraries
• fast numerical expression evaluator for NumPy
• multi-threaded, MKL enabled
• best on matrices that are too large to fit in L1 CPU cache

```python
>>> import numpy as np
global numexpr
>>> import numexpr as ne
>>> a = np.arange(10)
>>> b = np.arange(0, 20, 2)
>>> ne.evaluate("2*a+3*b")
array([ 0,  8, 16, 24, 32, 40, 48, 56, 64, 72])
```
- superset of the Python programming language
- designed to give C-like performance
- code is mostly written in Python
- compiled language that generates CPython extension modules
- extension modules can then be loaded and used by regular Python code using the import statement
- Cython files have a .pyx extension
hello.pyx:
def say_hello():
    print "Hello World!"

launch.py:
import hello
hello.say_hello()
cython in ipython/jupyter notebooks

```python
In [1]: %load_ext Cython

In [2]: %%cython
   ....: def f(n):
   ....:     a = 0
   ....:     for i in range(n):
   ....:         a += i
   ....:     return a
   ....:
   ....: cpdef g(int n):
   ....:     cdef int a = 0, i
   ....:     for i in range(n):
   ....:         a += i
   ....:     return a
   ....:

In [3]: %timeit f(1000000)
42.7 ms ± 783 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [4]: %timeit g(1000000)
74 µs ± 16.6 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```
FORTRAN to Python wrapper:

- to call Fortran 77/90/95 external subroutines and Fortran 90/95 module subroutines as well as C functions;
- to access Fortran 77 COMMON blocks and Fortran 90/95 module data, including allocatable arrays
- interfacing is done via numpy arrays
Run `f2py` and generate a python module

```
$ f2py -c fib1.f -m fib1
```

```c
C FILE: FIB1.F
SUBROUTINE FIB(A,N)
  C
  C    CALCULATE FIRST N FIBONACCI NUMBERS
  C
  INTEGER N
  REAL*8 A(N)
  DO I=1,N
    IF (I.EQ.1) THEN
      A(I) = 0.0D0
    ELSEIF (I.EQ.2) THEN
      A(I) = 1.0D0
    ELSE
      A(I) = A(I-1) + A(I-2)
    ENDIF
  ENDDO
END
C END FILE FIB1.F
```
GPU Programming

- Compute parts of your program on the GPU
- GPUs are programmed using CUDA or OpenCL
- Tools for python:
  - pyCUDA
  - numba
pyCUDA

- numpy like library for GPUs using CUDA (Nvidia only)
- arrays living on the GPU
- generate your own CUDA kernels
- arrays and textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and Events
- device queries
- Allow interactive use
- Integrate tightly with numpy
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpu

a = np.random.randn(4, 4).astype(float32)
a_gpu = gpu.to_gpu(a)

a_gpu2 = 2 * a_gpu  #<- this is running on the GPU

a_doubled = a_gpu2.get()

print a_doubled
print a_gpu
gpuarray class

pycuda.gpuarray:

feels like numpy.

• `gpuarray.to_gpu(np.array)`
• `np.array = gpuarray.get()`
• `+`, `-`, `*`, `/`, `fill`, `sin`, `exp`, `rand`, `basic indexing`, `norm`, `inner product`
• Mixed types (`int32 + float32 = float64`)
• print `gpuarray` for debugging.
• Allows access to raw bits
• Use as kernel arguments, textures, etc.
Avoiding extra store-fetch cycles for elementwise math:

```python
def a():
    a_g = curand((50,))
    b_g = curand((50,))

    def kernel(a, b):
        return ElementwiseKernel(  
            "float a, float x, float b, float y, float z",  
            "z[i] = a*x[i] + b*y[i]"
        )

    c_g = gpuarray.empty_like(a_g)
    kernel(5, a_g, 6, b_g, c_g)

    assert la.norm((c_g - (5*a_g + 6*b_g)).get()) < 1e-5
```
Example: A scalar product calculation

```python
from pycuda.reduction import ReductionKernel
dot = ReductionKernel(dtype_out=numpy.float32, neutral="0",
reduce_expr="a+b", map_expr="x[i]*y[i]",
arguments="const float *x, const float *y")

from pycuda.curandom import rand as curand
x = curand((1000*1000), dtype=numpy.float32)
y = curand((1000*1000), dtype=numpy.float32)

x_dot_y = dot(x,y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())
```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule(""
__global__ void multiply_them(float *dest, float *a, float *b)
{
    const int i = threadIdx.x;
    dest[i] = a[i] * b[i];
}
""")
multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1))
print dest-a*b
- just-in-time compiler for GPUs and CPUs
- LLVM compiler infrastructure
- integration with numpy

```python
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)
print(sum2d(a))
```
The End:
XKCD

I learned it last night! Everything is so simple! Hello world is just print "Hello, world!"

I dunno... Dynamic typing? Whitespace?

Come join us! Programming is fun again! It's a whole new world up here!

But how are you flying?

I just typed import antigravity

That's it?

... I also sampled everything in the medicine cabinet for comparison.

But I think this is the Python.
Python Intro
Python Syntax

- basic syntax
  - import, for, if, while, list comprehensions
- advanced syntax
- built-in 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):
raise name

try:
except name:
finally:

with expression as var:

global variable
nonlocal variable

lambda var: expression

@decorator

async def fun -> ann:
    assert condition
    yield value
    yield from generator
    await expression
- 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
125365987679340988385155987957344620719772763435558412643918634708860008684622476289189408
12290412402507934889820704250464446463778641104140990841878266383680568044115362044043884095
44441384289179095087047608175790842338441544887228788494128120919791295898721196764732642609051396426025390625
```
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
  \[
  \text{dd} = \text{open(”data.txt”).readlines()}
  \]
- print lines
  \[
  [x[:-1] \text{ for } x \text{ in open(”data.txt”,”r”).readlines()}]
  \]
- pretty print
  \[
  \text{from pprint import pprint}
  \]
  \[
  \text{pprint(dd)}
  \]
- binary files
  \[
  \text{xx} = \text{open(”data.txt”,”rb”).read()}
  \]
  \[
  \text{xx.__class__}
  \]
interaction with the shell

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:
>>> x=[1,2,3]
>>> x.append("one")
>>> y=x
>>> y[0]=2
>>> x[0]
2
>>> x.append(x)
>>> x
[2, 2, 3, 'one', [...]]
lists and tuples

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()
```
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.

They serve as an interface to C-code. If you need numerical arrays, use the numpy library (see below).
sets

sets are unordered lists. They provide all the methods from set theory like intersection and union. Elements are unique.

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