Python primer

Ferdinand.Jamitzky@LRZ.de

Python Intro

Principle of least surprise

More generally, the principle means that a component of a system should behave in a way that most users will expect it to behave; the behavior should not astonish or surprise users.

Login to Jupyter

Start up a browser and enter the following URL:

http://138.246.232.54:8000

Then use the following credentials:

User: user1 ...user99

Executable Pseudo-Code

```
for x in range(10):
    y=2*x
    if x==0:
        print("x is zero")
    elif x>5 and x<10:
        print("x is between 5 and 10")
    else:
        print(f"twice {x} = {y}")
```

“Python is executable pseudocode. Perl is executable line noise.” (– Old Klingon Proverb)

Zen of python (20.2.1991-?)

- 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 as seen from the orbit

Python

Basic Language and Syntax

Python 2

Python 3

Cython

External Libraries

Dask

NumPy

SciPy

Pandas

Builtin Libraries

Strings, IO

Web, XML

os, zip

Extensions

Jupyter

Tensor Flow

Spark

Blender

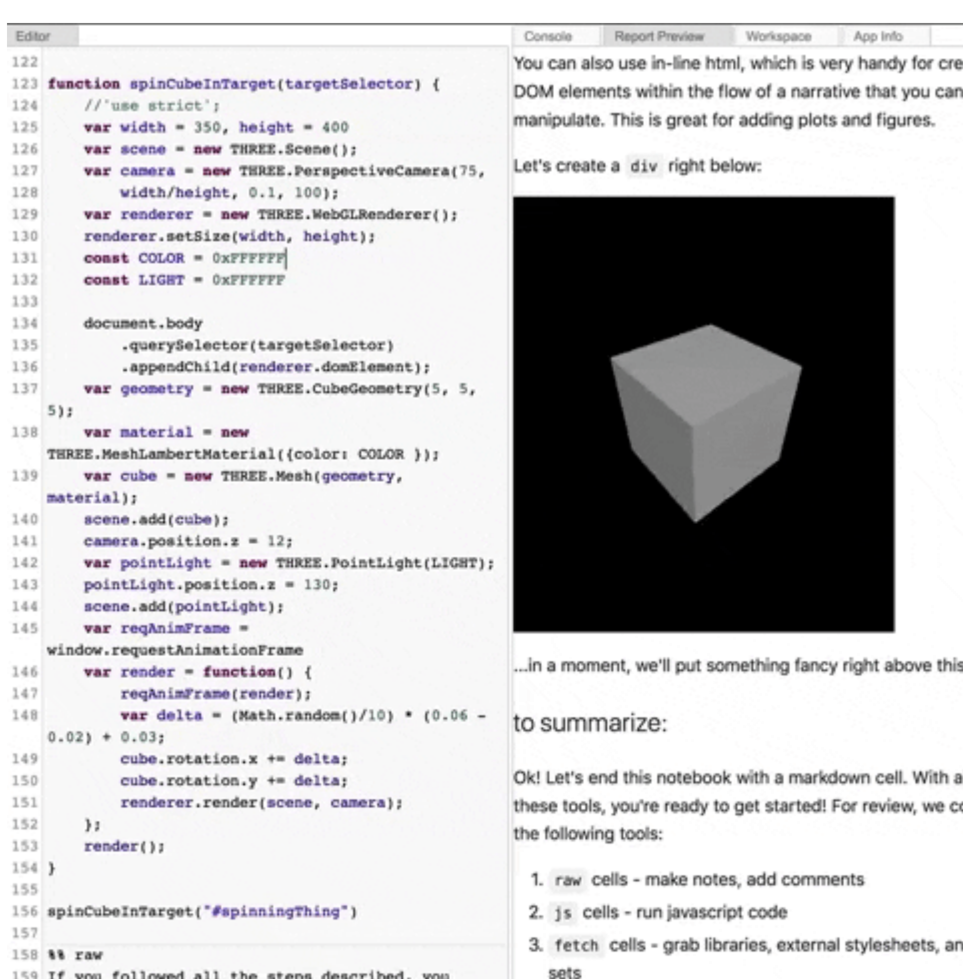
Python Syntax

- basic syntax
  - import, for, if, while, list comprehensions
- advanced syntax
- builtin data types
  - lists, tuples, arrays, sets
  - dicts
  - strings

How to try out python

- python in the browser:

https://alpha.iodide.io/



https://www.jetbrains.com/pycharm/

```
djtp_first_steps > polls > tests.py                                   Polls
tests.py ×
20          """
21          response = self.client.get(reverse('polls:index'))
22          self.assertEqual(response.status_code, 200)
23          self.assertContains(response, "No polls are available.")
24          self.assertQuerysetEqual(response.context['latest_question_list'], [])
25          self.test
26          (m) test_index_view_with_a_future_question(self)          QuestionViewTests
27      def te (m) test_index_view_with_a_past_question(self)        QuestionViewTests
28          """ (m) test_index_view_with_future_question_and_past_question   QuestionVi…
29          Qu (m) test_index_view_with_no_questions(self)           QuestionViewTests
30          in (m) test_index_view_with_two_past_questions(self)     QuestionViewTests
31          """ (f) _testMethodDoc                                    TestCase
32          cr (f) _testMethodName                                    TestCase
33          re (m) countTestCases(self)                               TestCase
34          se (m) defaultTestResult(self)                            TestCase
35          ^↓ and ^↑ will move caret down and up in the editor >>    π
36
37          )
38
39      def test_index_view_with_a_future_question(self):
40          """
41          Questions with a pub_date in the future should not be displayed on
42          the index page.
43          """
44          create_question(question_text="Future question.", days=30)
45          response = self.client.get(reverse('polls:index'))
46          self.assertContains(response, "No polls are available.",
47                              status_code=200)
48          self.assertQuerysetEqual(response.context['latest_question_list'], [])
49
50      def test_index_view_with_future_question_and_past_question(self):
51          """
52          Even if both past and future questions exist, only past questions
53          should be displayed.
54          """
55          create_question(question_text="Past question.", days=-30)
56          create_question(question_text="Future question.", days=30)
57          response = self.client.get(reverse('polls:index'))
58          self.assertQuerysetEqual(
59              response.context['latest_question_list'],
60              ['<Question: Past question.>']
61          )
62
63      def test_index_view_with_two_past_questions(self):
64          """
Statement seems to have no effect. Unresolved attribute reference 'test' for class 'QuestionViewTests'.   25:18  LF÷  UTF-8÷  Git: master÷
```

jupyter

A web-service where you can run any code through a browser interface.

basic rules of the game

- indentation matters!
- # denotes comments
- lists start from 0

- file type matters (*.py)!
- directory hierarchy matters!

```
for x in range(10):
        y = 2*x+1
# here comes the output:
        print(f"y= {y}")
print("finished loop")
```

basic rules of the game

Modules can be defined either by filename or by folder name.

```
$ python
# module by filename
>>> import myfile
# module by folder name
>>> import mymod

# call:
>>> myfile.myfunc()
hello
>>> mymod.myfunc()
world
```

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

myfile.py:

```
def myfunc():
    print(“hello”)
```

mymod/__init__.py:

```
def myfunc():
    print(“world”)
```

Variable names

- Variable names can consist of:
  - Alphabetic (also Greek or Umlauts)
  - Numbers
  - Underscore _
- Variables have to start with Alphabetic or Underscore

e.g. this is valid:

```
_sumOfAll_μラ0
```

- Try to stick to ASCII for readability, but YMMV

```python
Länge = [10,20,30,40,50]
#Berechne Mittelwert
ΣLänge = 0
for L in Länge:
    ΣLänge = ΣLänge + L
μ = ΣLänge / len(Länge)
print(f"Mittelwert = {μ}")
#this is valid:
ラーメン = "delicious"
π = 3.14159
jalapeño = "a hot pepper"
```

types, lists, tuples and dicts

- Python has the following number types:
  - `int, long, float, complex`
- Strings
  - `”this”, ’this’, ”””this”””, ’’’this’’’, u’this’, b’this’`
- Lists and tuples
  - `a=[1,2,3]` is a list,
  - `b=(1,2,3)` is a tuple (immutable)
- Dictionaries aka Associative Arrays
  - `a={ ’one’: 1, ’two’: ”zwei”}` is a dict

Keywords (more than 90% of python code)

```
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

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
```

Syntax

import

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:

```
>>> import numpy
>>> from numpy import *
>>> import numpy as np
>>> from numpy import pi as Pie
```

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

```
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

file i/o

- 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__
```

lrz

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

Data Structures

basic types

- Python has the following number types:
  - `int, long, float, complex`
  - `del var`

```
>>> x=0
>>> x=12345678901234567890123 45
>>> x**2
15241578753238836750495334799573386691205623 9
9025
```

basic types

```
>>> x=12345678901234567890123456789012345
>>> float(x)**12
1.253659876793410 3e+289
>>> float(x**12)
1.2536598767934098e+289
>>> x**12
1253659876793409883851559879573446207197 72763
```

basic types

Imaginary and complex numbers are built in:

```
>>> 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
```

Strings

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
```

strings

- 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/>'
```

lists, tuples, dictionaries

- **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

lists

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

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

reason for tuples: faster access

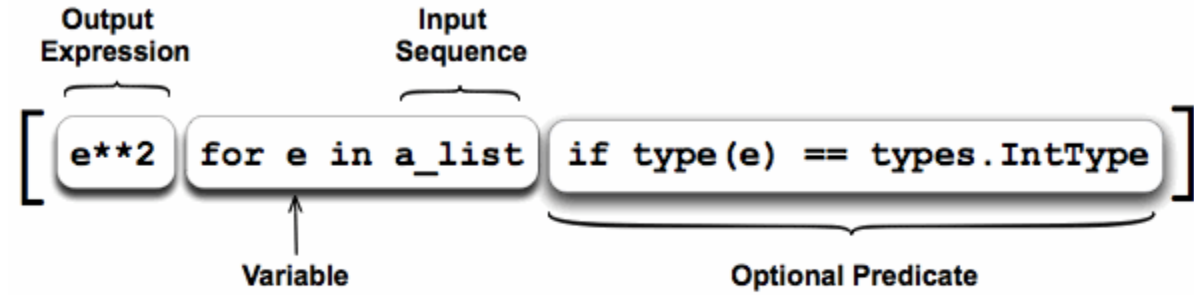
list comprehensions

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

```
>>> 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
```



dicts

dictionaries **aka** associative arrays **aka** key/value stores

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

dictionary comprehensions:

```
>>> {i:i**2 for i in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
>>> a.keys()
>>> a.values()
```

for loops with dicts

you can loop over a dict by:

```
>>> knights = {'gallahad': 'the pure',
'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
```

or

```
>>> {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.

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

python2 vs python3

- why python3?
  - you need Unicode?
  - you want to use generators (yield) extensively
- why python2?
  - many lists are iterators in py3 (range, filter, zip, map,...)
  - many old packages do not (yet) have a python3 version

- use 2to3 converter (or 3to2 for backwards)

```
$ 2to3 myprog.py
```

Functions

functions

- keywords
- doc strings
- specials:
  - lambda
  - yield, yield from
  - annotations
  - async, await, ...

lrz

functions: keywords

```
def myfun(a, b=1, c=[1,2], *args):
    “decription goes here”
    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))
```

lrz

functions: lambda functions

```
f1 = lambda x: x+1

def f2(x):
    return x+1

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

lrz

Putting it all together

Compute prime numbers up to 30

```
def isprime(n):
    return n not in \
      [x*y for x in range(n) for y in range(n)]

print([n for n in range(2,30) if isprime(n)])
```

Classes

Object oriented programming in a nutshell

- Everything in python is an Object (numbers too)
- Objects: instances of classes
- Classes: blueprints for objects
- Methods: functions attached to objects
- Classes can inherit "blueprints" from other classes

```
>>> a=[]
>>> type(a)
>>> print a.__class__
>>> print dir(a)
>>> a.__doc__
```

class definition

```
class point2d:
  def __init__(self,x=0,y=0):
    self.x=x
    self.y=y
  def move(self,dx=0,dy=0):
    self.x+=dx
    self.y+=dy
    return self
  def __str__(self)
    return f"Point at {self.x}, {self.y}”
```

class usage

```
>>> p0=point2d()
>>> p1=point2d(x=1)
>>> p2=point2d(3,4)

>>> p0.move(1,2)
>>> p3 = p1.move(dx=2).move(dy=3)
>>> print(p3)
```

lrz

magic methods

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

```
>>> str(a)
```

is translated to:

```
>>> a.__str__()
```

and

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

Advanced Topics

Advanced topics

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

lrz

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)
```

lrz

decorators aka macros

```
@mymacro
def ff(y):
  return y*2

def mymacro(f):
  return lambda *x: ”Hey! ”+str(f(*x))

def mymacro(somefunc)
  def tempfunc(*x):
    return ”Hey! ”+str(somefunc(*x))
  return tempfunc
```

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)
```

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__()
```

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():
...     yield “one“
...     yield 2
...     yield [3,4]
...
>>> a=createGenerator()
>>> next(a)
“one“
```

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]
```

Output Expression
Input Sequence

```
[ e**2 for e in a_list if type(e) == types.IntType ]
```

Variable
Optional Predicate

Aspect Oriented Programming in python

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

```
print("foo")
with tag("h1"):
    print("foo")

foo
<h1>foo</h1>

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

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, virtualenv (dead)
- pip (alive, default package manager for python)
- conda (state of the art)
- Data formats:
  - wheel (official package format PEP427)
  - egg (old package format)

lrz

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

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
```

Python module at LRZ

- 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.
>>>
```

Generate your own python LRZ environment

- LRZ uses the conda package manager for python libraries. In the default module only a minimla 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
```

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'
```

lrz

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

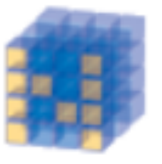
ipython exercise

Explain what the following commands return

```
>>> !ls
>>> files=!ls -al
>>> files.sort(5,num=True)
>>> files.grep(“a”,field=2)
>>> %cd
>>> %timeit
```

Computing and Plotting Libraries

scipy



NumPy
Base N-dimensional array package

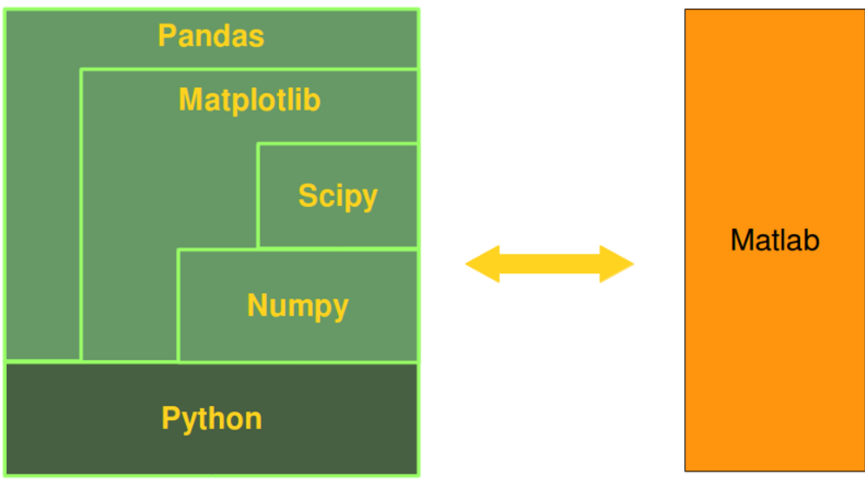
SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis



Numerical Computations

numpy

- 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
- for comparison to other array languages (Numpy vs MATLAB, R, IDL) see:

http://mathesaurus.sourceforge.net/

Numpy in a nutshell

- NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called axes.

```
>>> A=[[ 1., 0., 0.],[ 0., 1., 2.]]
>>> A.ndim
>>> A.shape
>>> A.size
>>> A.dtype
>>> A.itemsize
```

Array Creation

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

Array Creation

```
>>> np.zeros((3,4))
>>> np.ones((3,4), dtype=np.int16)
>>> np.empty((2,3))
>>> np.arange(10,30,5)
>>> np.arange(0,2,0.3)
>>> np.linspace(0,2,9)
>>> b = np.arange(12).reshape(4,3)
```

Basic Operations

- Vector Operations on Arrays:
  - elementwise add, substract, multiply, divide, power
  - special functions: sin, cos, ...
  - elementwise comparison
  - Matrix Product A@B
  - in place operations A+=3
  - A.sum(), A.cumsum(), A.min(), A.max()

Universal Functions

- these functions operate elementwise on an array, producing an array as output

all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Indexing, Slicing, Iterating

- indexing and slicing like for python lists

```
>>> a[2:5]
>>> a[ : :-1]
>>> b[1:3, : ]
>>> b[-1]
```

Stacking Arrays

```
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Copy and views

- Simple assignments make **no** copy of array objects or of their data.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same object
True
>>> d = a.copy()     # a new array object with new data is created
>>> d is a
False
```

Random Numbers

Numpy has a plentitude of random number distributions

uniform:

```
>>> A = np.random.random(2,3))
>>> A = np.random.uniform(size=10)
```

others are:

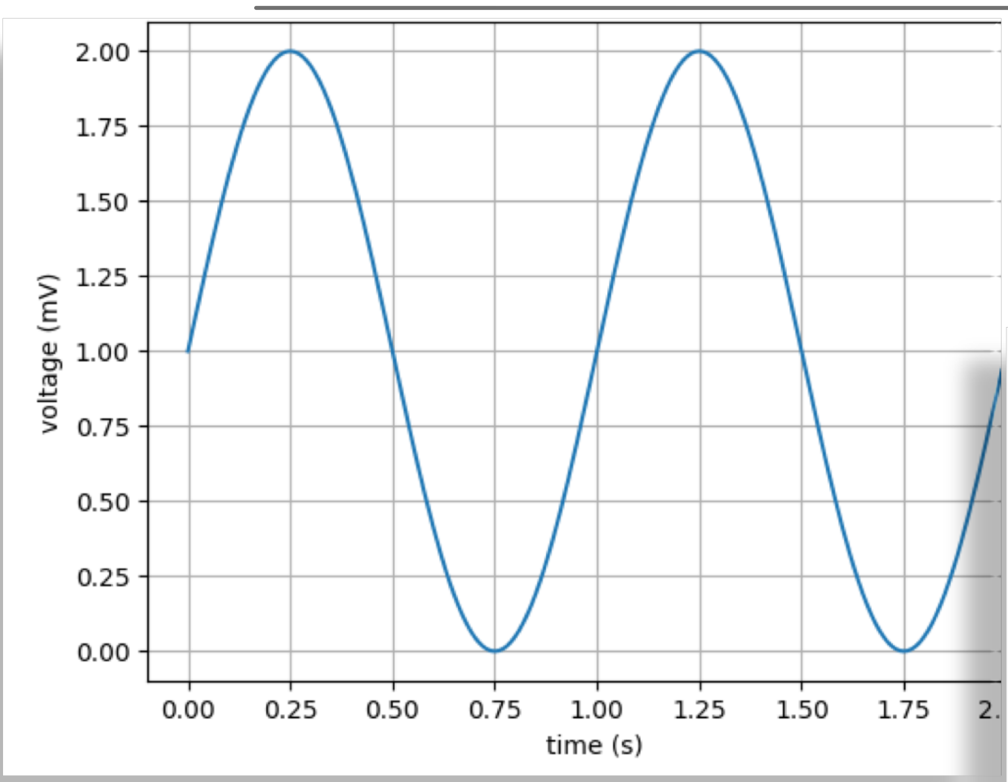
beta, binomial, chisquare, dirichlet, exponential, F, gamma, geometric, gumbel, hypergeometric, laplace, logistic, lognormal, logseries, multinormal, normal, pareto, poisson, power, Rayleigh, Cauchy, standard_t, triangular, uniform, vonmises, wald, weibul, zipf

numpy exercise

Explain the output of the following commands:

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x
>>> y = np.arange(10)
>>> y
>>> a = np.array([1, 2, 3, 6])
>>> b = np.linspace(0, 2, 4)
>>> c = a - b
>>> c
>>> a**2
```

matplotlib




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

# Data for plotting
t = 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()
```

matplotlib

matplotlib

pcolormesh with levels

contourf with levels

Scores by group and gender

Volume and percent change

jupyter+scipy+matplotlib+latex

IP[y]: Notebook spectrogram Last Checkpoint: a few seconds ago (autosaved) IPython (Python 3)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

Simple spectral analysis

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

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

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

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 builtin `specgram` routine:

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');
```

Raw audio signal

8000, 6000, 4000, 2000, 0, −2000, −4000, −6000, −8000, −10000

0, 10000, 20000, 30000, 40000, 50000

Spectrogram

1.0, 0.8, 0.6, 0.4, 0.2, 0.0

0, 5000, 10000, 15000, 20000, 25000

Data Analysis using Pandas

Pandas – Data Frames for python

- 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.

Pandas in a nutshell

The two primary data structures of pandas

- Series (1-dimensional)
- DataFrame (2-dimensional)

handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering.

For R users:

- DataFrame provides everything that R's data.frame provides
- pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Dataframes and Series

DataFrame is a container for Series, and Series is a container for scalars.

```
for col in df.columns:
    series = df[col]
    # do something with series

s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

pandas

- Object Creation
- Viewing Data
- Selection
- Missing Data
- Operations
- Merge
- Grouping
- Reshaping
- Time Series
- Categorials
- Plotting
- Data I/O

Object Creation

Creating a Series by passing a list of values, letting pandas create a default integer index:

```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
```

Creating a DataFrame by passing a NumPy array, with a datetime index and labeled columns:

```
df = pd.DataFrame(np.random.randn(6, 4),
index=dates, columns=list('ABCD'))
```

Viewing Data

```
df.head()
df.tail(3)
df.index
df.columns
df.to_numpy()
df.describe()
```

Selection

```
df['A']
df[0:3]
df.loc[:, ['A', 'B']]
df.iloc[3:5, 0:2]
df[df.A > 0]
df[df > 0]
df2[df2['E'].isin(['two', 'four'])]
df.loc[:, 'D'] = np.array([5] * len(df))
df2[df2 > 0] = -df2
```

Missing Data

```
df1 = df.reindex(index=dates[0:4],
columns=list(df.columns) + ['E'])
df1.dropna(how='any')
df1.fillna(value=5)
pd.isna(df1)
```

Operations

```
df.mean()
df.mean(1)
df.apply(np.cumsum)
df.apply(lambda x: x.max() - x.min())
s.value_counts()
s.str.lower()
```

Merge

```
pieces = [df[:3], df[3:7], df[7:]]
pd.concat(pieces)
pd.merge(left, right, on='key')
df.append(s, ignore_index=True)
```

Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria
- Applying a function to each group independently
- Combining the results into a data structure

```
>>> df.groupby('A').sum()
>>> df.groupby(['A', 'B']).sum()
```

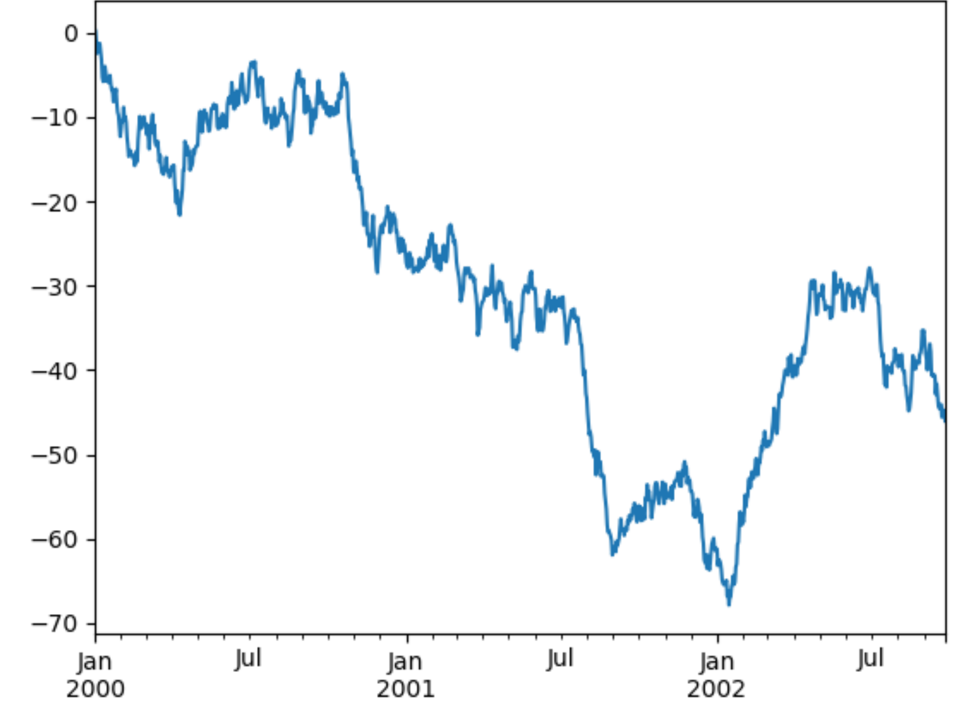
Reshaping

- Stack
  The stack() method “compresses” a level in the DataFrame’s columns.
- Pivot Table

```
>>> pd.pivot_table(df, values='D', index=['A', 'B'],
columns=['C'])
```

Plotting

```
>>> ts = pd.Series(np.random.randn(1000),
index=pd.date_range('1/1/2000', periods=1000))
>>> ts = ts.cumsum()
>>> ts.plot()
```



Getting Data In/Out

- CSV

```
>>> pd.read_csv('foo.csv')
>>>  df.to_csv('foo.csv')
```

- Excel

```
>>> pd.read_excel('foo.xlsx', 'Sheet1', index_col=None,
na_values=['NA'])
>>> df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

- HDF5

```
>>>  pd.read_hdf('foo.h5', 'df')
>>> df.to_hdf('foo.h5', 'df')
```

Machine Learning Packages

Python Packages

Machine Learning with python

- Theano (discontinued)
- Tensorflow (Google)
- Torch/PyTorch (Facebook)
- MXnet (Apache, Amazon)
- CNTK (Microsoft)
- Keras (on top of TensoFlow, Tehano or CNTK)
- Caffe / Caffe2 (Facebook, lightweight)
- PaddlePaddle (Baidu for text mining in English and Chinese)
- Scikit Learn (google summer of code)

theano

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
- Developers now at goolge

tensorflow

TensorFlow

- 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

tensorflow

```
# 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
```

tensorflow

TensorFlow

```
# load TensorFlow
>>> import tensorflow as tf
# Initialize two vectors
>>> x = tf.constant([1,2,3,4])
>>> y = tf.consta
# Multiply
z= tf.multiply(x,
# Initialize Sess
>>> with tf.Sessi
. . . out = sess.
. . . print(out)
6
```

How does TensorFlow work

Python
Golang
Java
...

compile

Graph
(DAG)

run

C++ Core

Keras

- 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

Keras Applications

ResNet-50

PRETRAINED MODEL

```
# resnet50 pretrained application in keras

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

How-to go parallel

Why?

- You have many independent tasks (easy)

or

- You want to accerelate single complex task (hard)

Recipe:

Turn the single complex task into many independent simple tasks, but how?

How-to go parallel

Why?

- You have many independent tasks (easy)

or

- You want to accerelate single complex task (hard)

Recipe:

Turn the single complex task into many independent simple tasks, but how?

Why parallel programming?

End of the free lunch

Moore's law means no longer faster processors, only more of them. But beware!

2 x 3 GHz < 6 GHz

(cache consistency, multi-threading, etc)

Supercomputer scaling

Future of Supercomputing

Moore's Law for Supercomputers

100 Eflop/s
10 Eflop/s
1 Eflop/s
100 Pflop/s
10 Pflop/s
1 Pflop/s
100 Tflop/s
10 Tflop/s
1 Tflop/s
100 Gflop/s
10 Gflop/s
1 Gflop/s
100 Mflop/s

1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018 2020

1 Eflop/s
1 Pflop/s
1 Tflop/s
SUM
N=1
N=500
6-8 years
8-10 years
Notebook
Notebook
Notebook

Supercomputer Layout

Supercomputer aka HPC Cluster

Pruned Tree

Switch

Fat Tree

Island

Node

Accelerator: GPU, MIC

Core (4)

Socket (2)

Parallel and Distributed Programming

- multiprocessing
- Mpi4py
- Ipython parallel
- dask

See also:

https://chryswoods.com/parallel_python/README.html

Global Interpreter Lock (GIL)

- 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

Embarrassingly parallel (step-by-step)

- Take as example the following script

***myscript.sh*:**

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

You can run it interactively by:

```
$ ./myscript.sh
```

Embarrassingly parallel (step-by-step)

Please do not block the login nodes with production jobs, but run the script in an interactive slurm shell:

```
$ salloc –pmpp2_inter –n1 myscript.sh
```

Change the last line in the script:

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

Embarrassingly parallel (step-by-step)

Run multiple copies of the the script in an interactive slurm shell:

```
$ salloc –pmpp2_inter –n4 myscript.sh
```

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

To use different input files you can use the environment variable:

`os.environ['SLURM_PROCID']` (it is set to 0,1,2,3,...)

Use this variable to select your workload.

Example:

```
$ salloc –pmpp2_inter –n2 srun
python –c "import os; os.environ['SLURM_PROCID']"
0
1
```

Embarrassingly parallel (step-by-step)

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
```

Embarrassingly parallel (step-by-step)

For serial (single node, multithreaded but not MPI) loads use the serial queue and add options for the runtime:

```
#!/bin/bash
#SBATCH --clusters=serial
#SBATCH -n4     # 4 tasks
#SBATCH --time=01:00:00 # 1hour
source /etc/profile.d/modules.sh
module load python
cd ~/mydir
srun python myscript.py

$ sbatch myscript.slurm
```

lrz

SLURM Job Arrays

If you want to send a large number of jobs then use Job Arrays.

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

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

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

The maximum size of array job is 1000

lrz

Important 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

lrz

Using ipcluster

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!']
```

lrz

Ipcluster on SLURM

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
```

Cont.

ipcluster_config.py:

```
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=""
"""
```

lrz

Usage of ipcluster

Start a python shell and import the client function

```
>>> from ipyparallel import Client
```

Connect to the ipcluster

```
>>> c=Client(profile="slurm")
```

Generate a view on the cluster

```
>>> dview=c[:]
```

The view can now be used to perform parallel computations on the cluster

lrz

Usage of ipcluster

Run a string containing python code on the ipcluster:

```
>>> dview.execute("import time")
```

Run a single function and wait for the result:

```
>>> dview.apply_sync(time.sleep, 10)
```

Or return immediately:

```
>>> dview.apply_async(time.sleep, 10)
```

Map a function on a list by reusing the nores of the cluster:

```
>>> dview.map_sync(lambda x: x**10, range(32))
```

Defining parallel functions

Define a function that executes in parallel on the ipcluster:

```
In [10]: @dview.remote(block=True)
    ....: def getpid():
    ....:     import os
    ....:     return os.getpid()
    ....:
In [11]: getpid()
Out[11]: [12345, 12346, 12347, 12348]
```

Usage of ipcluster with NumPy

The @parallel decorator parallel functions, that break up an element-wise operations and distribute them, reconstructing the result.

```
In [12]: import numpy as np
In [13]: A = np.random.random((64,48))
In [14]: @dview.parallel(block=True)
    ....: def pmul(A,B):
    ....:     return A*B
```

lrz

Loadbalancing

You can create a view of the ipcluster that allows for loadbalancing of the work:

```
>>> lv=c.load_balanced_view()
```

This view can be used with all the above mentioned methods, auch as: execute, apply, map and the decorators.

The load balancer can even have different scheduling strategies like "Least Recently Used", "Plain Random", "Two-Bin Random", "Least Load" and "Weighted"

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

- 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.

multiprocessing

```
# 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.
```

Multiprocessing pool

```
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:

```
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=1) as ex:
    future = ex.submit(pow, 323, 1235)
    print(future.result())
```

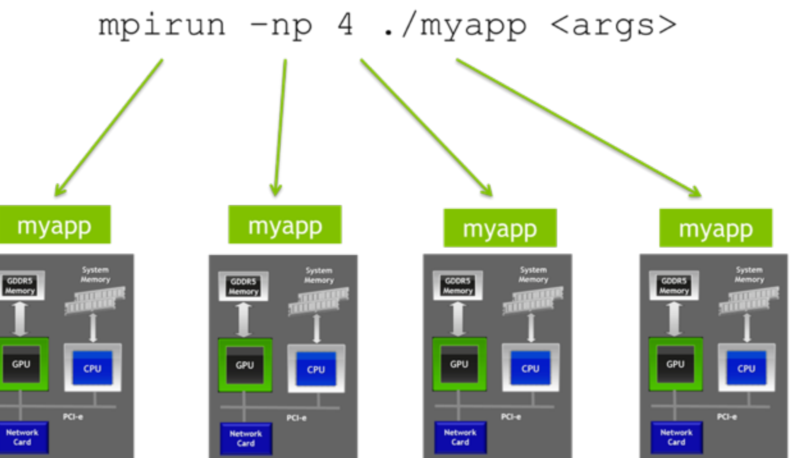
Message Passing

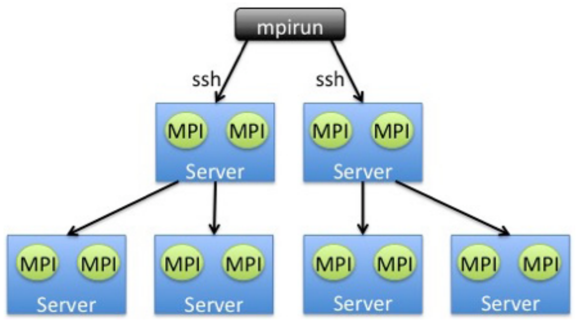
- many independent processes (10 - 100.000)
- one tasklist for all (program)
- everyone can talk to each other (in principle)
- private memory
- needs communication strategy in order to scale out
- very often: nearest neighbor communication
- beware of deadlocks!

mpi4py

```
$ 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)
```

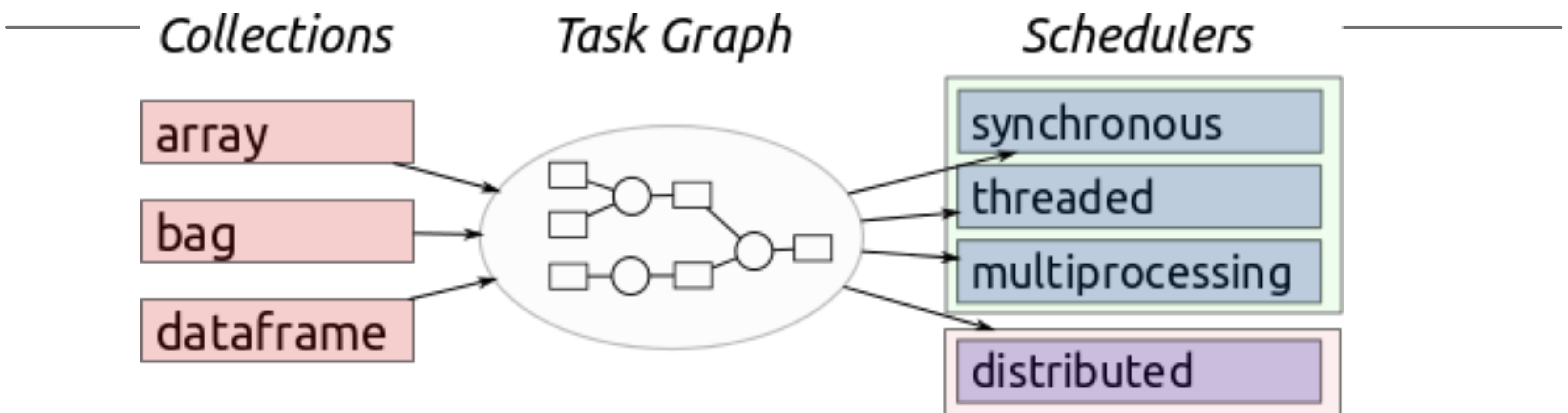




Worker queue

- 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

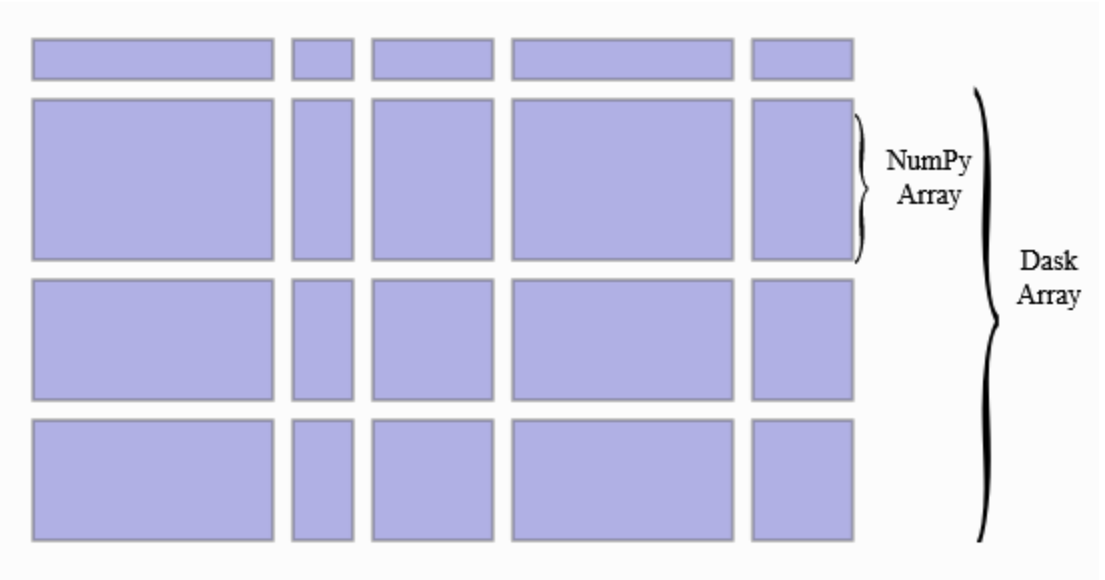
dask



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



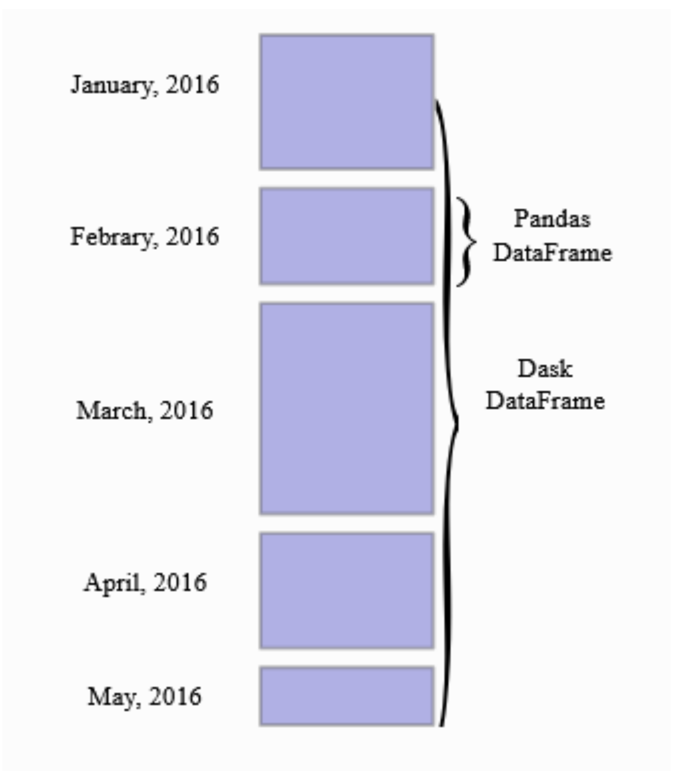

```
>>> import dask.array as da
>>> a=da.random.uniform(size=1000, chunks=100)
```

https://docs.dask.org/en/latest/array-api.html

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

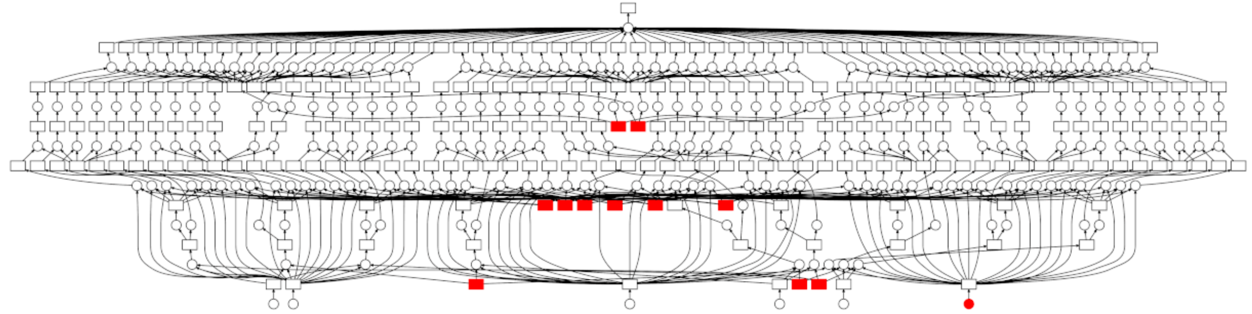
```
>>> import dask.dataframe as dd
>>> df=dd.read_csv('2014-*.csv')
```



dask execution graph

```
>>> 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.



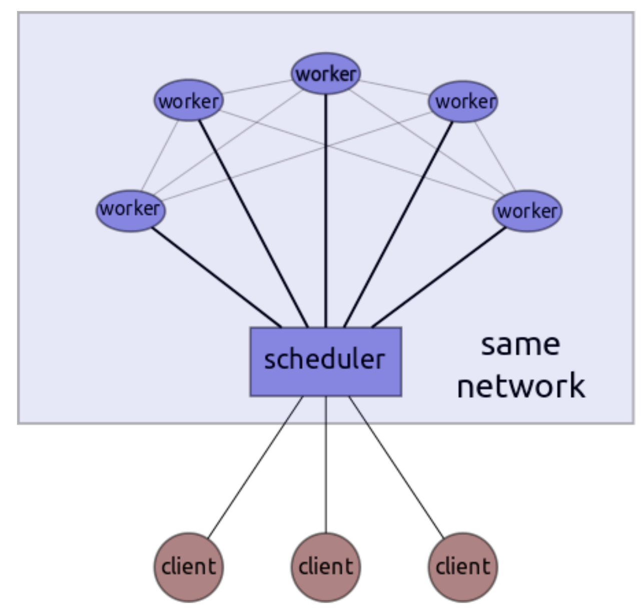
dask.distributed

- 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
```

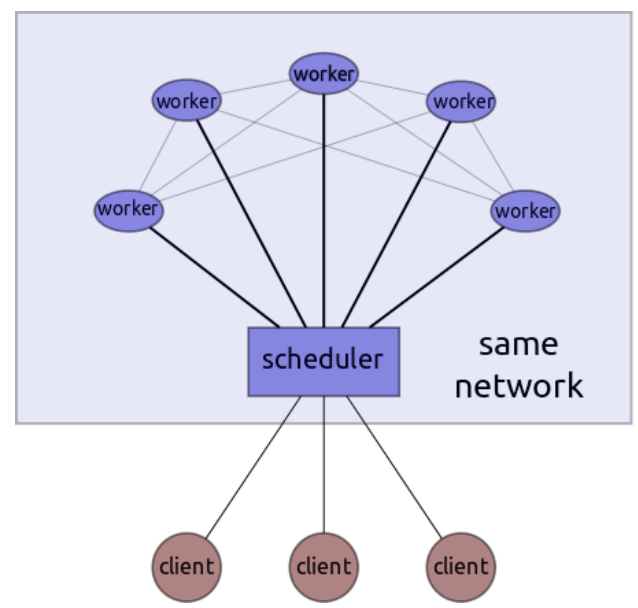


dask.distributed

- Start a client

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

now all dask operations will be distributed to the scheduler which distributes them to the cluster



DASK mobile

- install qpython
- open pip console
- install dask
- install toolz
- install ipython

```
pace(__name__)
IPython on QPython

IPython 5.5.0 - An enhanced Interactive Python.
    Type 'copyright', 'credits' or 'license' for more information, Type '?' for help, 'exit' for exit.

In <1>: import dask.array as da

In <2>: %timeit da.random.uniform(size=100000,chunks=10000).sum().compute()
10 loops, best of 3: 69.6 ms per loop

In <3>:
```

Dask dataframe limitations

Dask DataFrame has the following limitations:

- Setting a new index from an unsorted column is expensive
- Many operations like groupby-apply and join on unsorted columns require setting the index, which as mentioned above, is expensive
- The Pandas API is very large. Dask DataFrame does not attempt to implement many Pandas features or any of the more exotic data structures like NDFrames
- Operations that were slow on Pandas, like iterating through row-by-row, remain slow on Dask DataFrame

When to use dask

Dask DataFrame is used in situations where Pandas is commonly needed, usually when Pandas fails due to data size or computation speed:

- Manipulating large datasets, even when those datasets don’t fit in memory
- Accelerating long computations by using many cores
- Distributed computing on large datasets with standard Pandas operations like groupby, join, and time series computations

When NOT to use dask

Dask DataFrame may not be the best choice in the following situations:

- If your dataset fits comfortably into RAM on your laptop, then you may be better off just using Pandas. There may be simpler ways to improve performance than through parallelism
- If your dataset doesn't fit neatly into the Pandas tabular model, then you might find more use in dask.bag or dask.array
- If you need functions that are not implemented in Dask DataFrame, then you might want to look at dask.delayed which offers more flexibility
- If you need a proper database with all that databases offer you might prefer something like Postgres

Low level programming

Python numerical libraries

MOPS

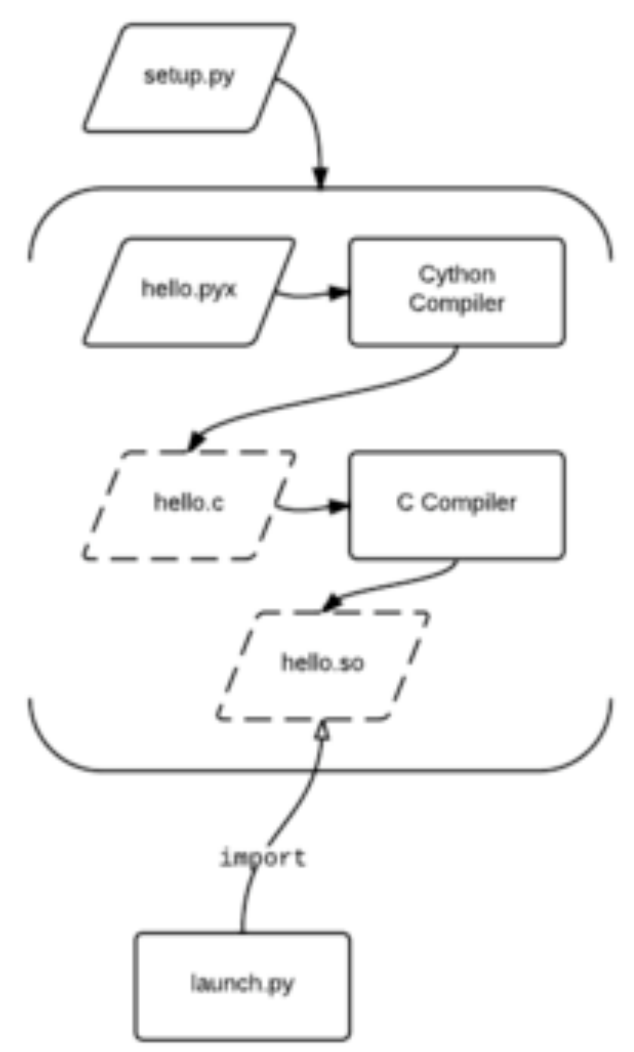
6000
5000
4000
3000
2000
1000
0

4800 MOPS
C/C++
Cython
3400 MOPS
NumExpr
1200 MOPS
Numpy
440 MOPS

1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432
67108864
134217728

cython

- 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



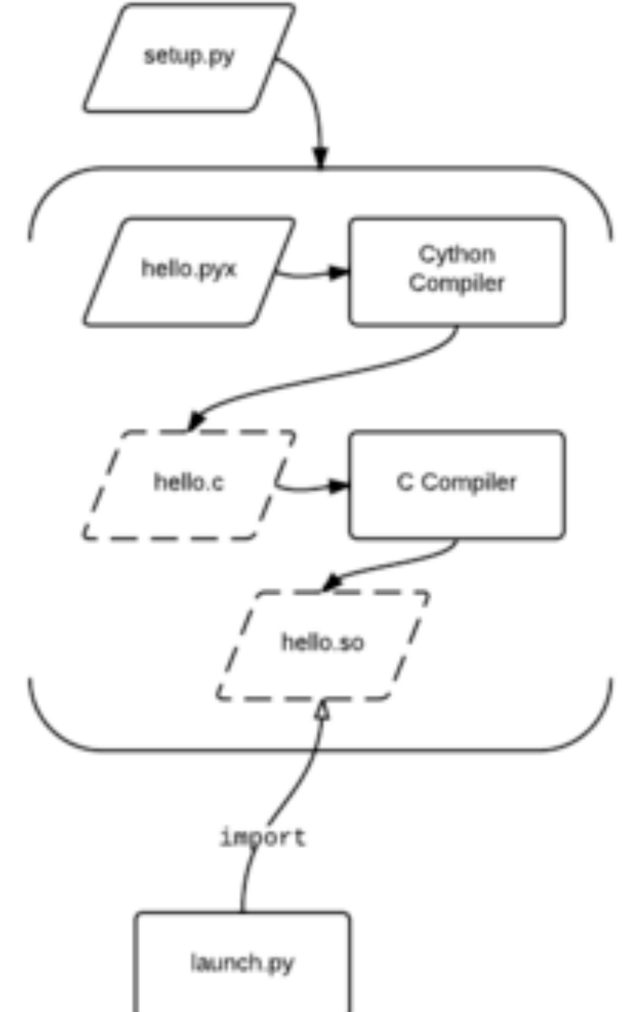
cython

hello.pyx:

```
def say_hello():
    print "Hello World!"
```

launch.py:

```
import hello
hello.say_hello()
```



cython in ipython/jupyter notebooks

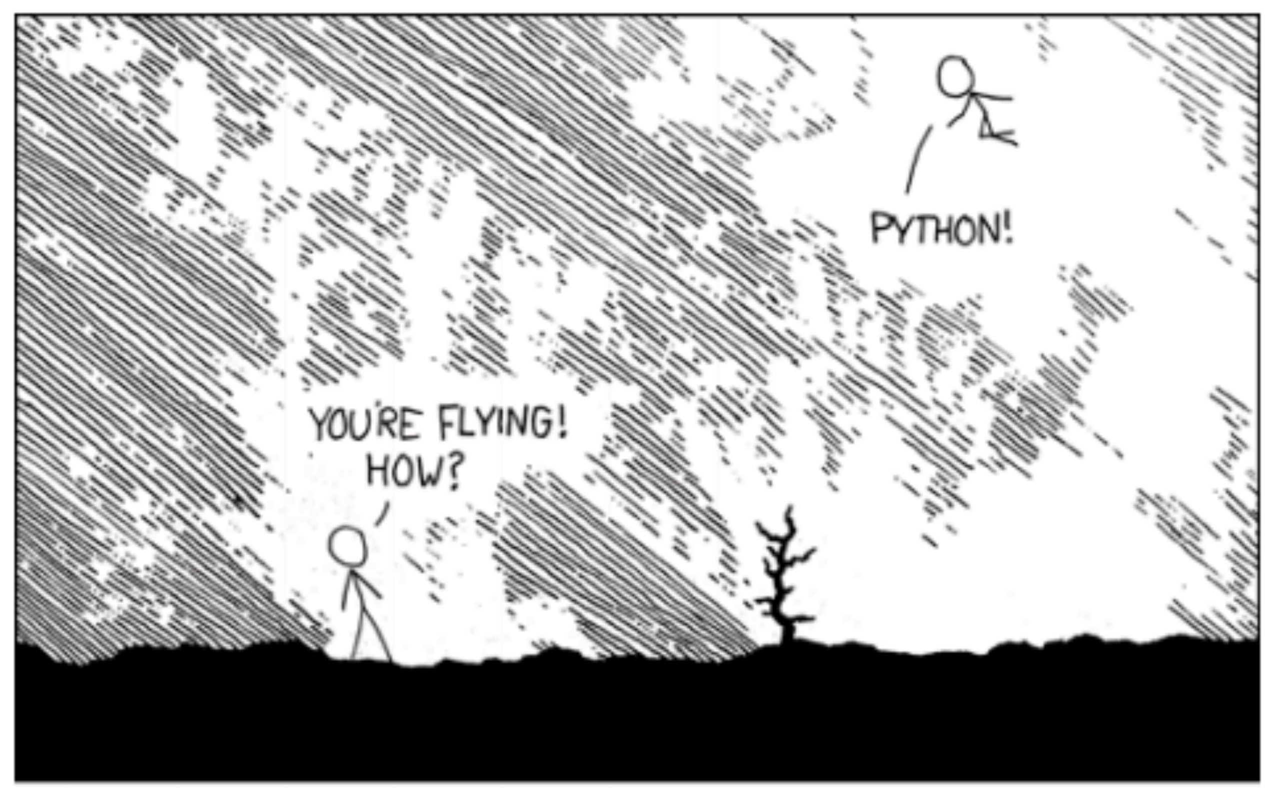
```
In [1]: %load_ext Cython

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)
```

The End: XKCD



Course Evaluation

Please visit

https://survey.lrz.de/index.php/248382

and rate this course.

Your feedback is highly appreciated!

Thank you!