Part VI

Scientific Computing in Python

Doing Maths in Python

Standard sequence types (`list`, `tuple`, `...`)

- Can be used as arrays
- Can contain different types of objects
  - Very flexible, but slow
  - Loops are not very efficient either
- For efficient scientific computing, other datatypes and methods required

Modules

- NumPy
- Matplotlib
- SciPy

NumPy

Module numpy

Homogeneous arrays

- NumPy provides arbitrary-dimensional homogeneous arrays
- Example

```
from numpy import *
a = array([[1,2,3],[4,5,6]])
print(a)
type(a)
a.shape
print(a[0,2])
a[0,2] = -1
b = a*2
print(b)
```

Array creation

- Create from (nested) sequence type
- Direct access with method `[]`

```
a = array([1,2,3,4,5,6,7,8])
a[1]
a = array([[1,2,3,4],[5,6,7,8]])
a[1,1]
a = array([[[1,2],[3,4]],[[5,6],[7,8]]])
a[1,1,1]
```

- Properties of arrays

```
a.ndim          # number of dimensions
a.shape         # dimensions
a.size          # number of elements
a.dtype         # data type
a.itemsize      # number of bytes
```

Data Types

- Exact, C/C++-motivated type of array elements can be specified
- Otherwise, defaults are used
- Some types (different storage requirements):
  - `int_`, `int8`, `int16`, `int32`, `int64`,
  - `float_`, `float8`, `float16`, `float32`, `float64`,
  - `complex_`, `complex64`,
  - `bool_`, `character`, `object_`
- Standard python type names result in default behaviour

```
array([[1,2,3],[4,5,6]], dtype=int)
array([[1,2,3],[4,5,6]], dtype=complex)
array([[1,2,3],[4,5,6]], dtype=int8)
array([[1,2,3],[4,5,1000]], dtype=int8)    # wrong
array([[1,2,3],[4,5,"hi"]], dtype=object)
```

Create Arrays

- (Some) default matrices (optional parameter: dtype)

```
arange([a,] b [,stride])   # as range, 1D
zeros( (3,4) )
ones( (1,3,4) )
empty( (3,4) )             # uninitialized (fast)
linspace(a, b [, n])       # n equidistant in [a,b]
logspace(a, b [, n])       # 10**a to 10**b
identity(n)                # 2d

fromfunction(lambda i,j: i+j, (3,4), dtype=int)
def f(i,j):
    return i+j
fromfunction(f, (3,4), dtype=int)
```

Manipulate Arrays

- Reshaping arrays

```
a = arange(12)
b = a.reshape((3,4))
a.resize((3,4))                    # in-place!
a.transpose()
a.flatten()


# Example use-case:
a = arange(144)
a.resize((12,12))
```

Create Arrays (2)

- Create/Copy from existing data

```
a = arange(12); a.resize((3,4))
copy(a)
diag(a); tril(a); triu(a)

empty_like(a)                    # copy shape
zeros_like(a)
ones_like(a)
```

Array Access and Manipulation

- Typical slicing operations can be used
- Separate dimensions by comma

```
a = arange(20); a.resize((4,5))
a[1]
a[1:2,:]
a[:,::2]
a[::2,::2]
a[::2,::2] = [[0, -2, -4],[-10, -12, -14]]
a[1::2,1::2] = -1*a[1::2,1::2]
```

- Selective access

```
a[a > 3]
a[a > 3] = -1
```

Array Access

- Iterating over entries

```
for row in a:
    print(row)

b = arange(30); b.resize((2,3,4))
for row in b:
    for col in row:
        print(col)

for entry in a.flat:
    print(entry)
```

Computing with Arrays

- Fast built-in methods working on arrays

```
a = arange(12); a.resize((3,4))
3*a
a**2
a+a^2
sin(a)
sqrt(a)
prod(a)
sum(a)

it = transpose(a)

x = array([1,2,3])
y = array([10,20,30])
inner(x, y)
dot(it, x)
cross(x,y)
```

Computing with Arrays (2)

- There is much more...

```
var()                cov()                std()
mean()               median()
min()                max()
svd()
tensordot()
...
```

- Matrices (with `mat`) are subclasses of `ndarray`, but strictly two-dimensional, with additional attributes

```
m = mat(a)
m.T     # transpose
m.I     # inverse
m.A     # as 2d array
m.H     # conjugate transpose
```

Submodules

- NumPy has many useful submodules for various purposes.
- See the reference here:
  `https://docs.scipy.org/doc/numpy/reference/`

Module `numpy.linalg`

- Core linear algebra tools

```
norm(a); norm(x)
inv(a)
solve(a, b)       # LAPACK LU decomp.
det(a)
eig(a)
cholesky(a)
```

Module `numpy.fft`

- Fourier transforms

Matplotlib

Matplotlib

What is it?

- Object-oriented library for plotting 2D
- Designed to be similar to the matlab plotting functionality
- Designed to plot scientific data, built on numpy datastructures

Example - First Plot

partially taken from
http://matplotlib.sourceforge.net/users/screenshots.html

```
import matplotlib.pyplot as plt

x = arange(0.0, 2*pi, 0.01)
y = sin(x)
plt.plot(x, y, linewidth=4)
plt.plot(x,y)

plt.xlabel('Label for x axis')
plt.ylabel('Label for y axis')
plt.title('Simple plot of sin')
plt.grid(True)
plt.show()
```

Example – Using Subplots

```
from matplotlib.pyplot import *
def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

show() # gives error but helps ;-)

subplot(2,1,1) # rows, columns, which to show
plot(t1, f(t1), 'go', t2, f(t2), 'k--')
subplot(2,1,2)
plot(t3, cos(2*pi*t3), 'r.')
```

Example – Using Subplots

```
# previous slide continued

subplot(2,1,1)
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

subplot(2,1,2)
grid(True)
xlabel('time (s)')
ylabel('Undamped')
```

SciPy

More than NumPy?

- SciPy depends on NumPy
- Built to work on NumPy arrays
- Providing functionality for mathematics, science and engineering
- Still under development
- NumPy is mostly about (N-dimensional) arrays
- SciPy comprises a large number of tools using these arrays
- SciPy includes the NumPy functionality (only one import necessary)
- A lot more libraries for scientific computing are available, some of them using NumPy and SciPy
- Here, just a short overview will be given
- `www.scipy.org` for more material (incl. the content of the following slides)

SciPy Organisation - Subpackages

cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
maxentropy	Maximum entropy methods
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions
weave	C/C++ integration

Special Functions

- Airy functions
- Elliptic functions
- Bessel functions (+ Zeros, Integrals, Derivatives, Spherical, Ricatti-)
- Struve functions
- A large number of statistical functions
- Gamma functions
- Legendre functions
- Orthogonal polynomials (Legendre, Chebyshev, Jacobi,...)
- Hypergeometric functios
- parabolic cylinder functions
- Mathieu functions
- Spheroidal wave functions
- Kelvin functions
- ...

Example: Interpolation - Linear

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

x = np.arange(0, 2.25*np.pi, np.pi/4)
y = np.sin(x)
f = interpolate.interp1d(x, y)

xnew = np.arange(0,2.0*np.pi,np.pi/100)
plt.plot(x,y,'o',xnew,f(xnew),'-')
plt.title('Linear␣interpolation')
plt.show()
```

Example: Interpolation - Cubic Spline

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

x = np.arange(0, 2.25*np.pi, np.pi/4)
y = np.sin(x)
spline = interpolate.splrep(x,y,s=0)
xnew = np.arange(0,2.02*np.pi,np.pi/50)
ynew = interpolate.splev(xnew, spline)

plt.plot(x,y,'o',xnew,ynew)
plt.legend(['Linear','Cubic␣Spline'])
plt.axis([-0.05,6.33,-1.05,1.05])
plt.title('Cubic-spline␣interpolation')
plt.show()
```

Potential Topics for Part VIII (Selected Topics)

recommended

- regular expressions
- exceptions
- unit tests

options

- logging
- anonymous functions (lambda) and `map` & `filter`
- module `time`
- module `argparse`
- Jupyter notebook
- function decorators
- advanced plotting
- extending Python using C/C++
- faster code