Introduction to Python

Today, three parts:

1. Basic Python
2. Numpy Arrays
3. Plotting

Python Recap

Python is:

- interpreted, dynamic, object-oriented

Python Recap

Python is:

- interpreted, dynamic, object-oriented

Two working modes:

1. Interactive Shell
   - IPython shell
   - line-by-line execution
   - Matlab-like coding
   - Good for quick tests, few-line-solutions, checking docs

Python Recap

Python is:

- interpreted, dynamic, object-oriented

Two working modes:

1. Interactive Shell
   - IPython shell
   - line-by-line execution
   - Matlab-like coding
   - Good for quick tests, few-line-solutions, checking docs
2. Source files
   - Edit source code and save as `*.py` files
   - Execute separatly, e.g. in terminal via: `python *.py`
   - Traditional programming style (like C/C++, Fortran, Java, ...)
   - Good for anything more complicated

Part I: Basic Python

Python structure

White-spaces matter!

- Line break indicates the end of a statement (no semicolon)
- Indention level defines blocks (no { }, no begin/end)
- Preferred indention mode: 4 spaces, no tabs
- Any text after '#' is ignored (comments)
- Multi-line comments start and end with """ (triple quotes)
- Every function should have a **doc-string**, a multi-line comment describing the use of the function

Python Basics

`if` statements:

```
if x < 0:
    print "x negative"
else:
    print "x positive"
```

Python Basics

`if` statements:

```
if x < 0:
    print "x negative"
else:
    print "x positive"
```

```
if x < 0:
    print "x negative"
elif x == 0:
    print "x zero"
else:
    print "x positive"
```

Python Basics

`for` statements:

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print w, len(w)
```

Python Basics

`for` statements:

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print w, len(w)
```

```
for n in range(2, 20):
    for x in range(2, n):
        if n == x*x:
            print n, 'equals', x, '*', x
            break
    else:
        # loop ended not by break
        print n, 'is not a square'
```

Python Basics

`while` statements:

```
# Fibonacci series
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

Python Basics

`while` statements:

```
# Fibonacci series
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

```
# machine precision
epsilon = 1.0
while 1.0 + epsilon > 1.0:
    epsilon = epsilon/2.0

print "machine precision:", epsilon
```

Python Basics

Function definitions:

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a+b

# Now call the function we just defined:
fib(2000)
```

Function definitions:

```
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print a,
        a, b = b, a+b

# Now call the function we just defined:
fib(2000)
```

```
f = fib   # functions are objects
f(100)    # and can be assigned
```

Functions with return values:

```
def fib2(n):
    """Return a list containing the Fibonacci
    series up to n."""
    result = []  # an empty list (not array)
    a, b = 0, 1
    while a < n:
        result.append(a)  # add a value
        a, b = b, a+b
    return result

f100 = fib2(100)     # call it
```

Default values and named parameters:

```
def load_data(file_name, separator=' ',
              comment='#', sort=True):
    """Loads spike trains from a file"""
    # implementation ...
    return data

load_data("spikes.txt", ';')
# equivalent:
load_data("spikes.txt", separator=';')

load_data("spikes.txt", comment='$')
load_data("spikes.txt", ';', sort=False)
```

Python Basics

Default values and named parameters:

```
def load_data(file_name, separator=' ',
              comment='#', sort=True):
    """Loads spike trains from a file"""
    # implementation ...
    return data

load_data("spikes.txt", ';')
# equivalent:
load_data("spikes.txt", separator=';')

load_data("spikes.txt", comment='$')
load_data("spikes.txt", ';', sort=False)
```

also: Arbitrary parameter lists, Lambda expressions

Basic Coding Exercise

Write a small Python program containing the following functions:

1. `factorial`: Given an integer, returns its factorial
2. `is_prime`: Given an integer, returns `True` if the number is prime, `False` otherwise. Hint: Python modulo operator: `x % 2`.
3. `deriv_fwd`: Given a function f, a position x and a step size h, returns the numerical derivative $(f(x+h) - f(x))/h$.

Basic Coding Exercise: factorial

```
def factorial(n):
    """ Computes the factorial of n. """
    if n > 0:
        # recursive
        return factorial(n-1)*n
    else:
        # stop recursion
        return 1

print "Factorial 5:", factorial(5)
print "Factorial 10:", factorial(10)
print "Factorial 0:", factorial(0)
```

Basic Coding Exercise: is_prime

```
from math import sqrt

def is_prime(n):
    """ Checks if n is prime"""
    # check up to sqrt(n), including sqrt(n)!
    for i in range(2, int(sqrt(n))+1):
        if n % i == 0:
            # found divisor, break
            break
    else:
        # no break occured -> prime
        return True
    return False

print "13 is prime:", is_prime(13)
print "49 is prime:", is_prime(49)
print "1301081 is prime:", is_prime(1301081)
```

Basic Coding Exercise: deriv_fwd

```
def deriv_fwd(f, x, h):
    """ Computes the numerical derivative of
    f at x using forward difference"""
    return (f(x+h)-f(x))/h

# import some functions for testing
from math import sqrt, sin

print "Derivative of sqrt at x=1.0:", \
    deriv_fwd(sqrt, 1.0, 1E-4)
print "Derivative of sin at x=0:", \
    deriv_fwd(sin, 0.0, 1E-4)
```

Part II: Numpy Arrays

Quick recap: Modules

Load a module:

```
import numpy
x = numpy.linspace(0.0, 1.0, 100)
```

Quick recap: Modules

Load a module:

```
import numpy
x = numpy.linspace(0.0, 1.0, 100)
```

Load a module with an alias:

```
import numpy as np
x = np.linspace(0.0, 1.0, 100)
```

Quick recap: Modules

Load a module:

```
import numpy
x = numpy.linspace(0.0, 1.0, 100)
```

Load a module with an alias:

```
import numpy as np
x = np.linspace(0.0, 1.0, 100)
```

Load specific functions into global namespace:

```
from numpy import linspace
x = linspace(0.0, 1.0, 100)
```

Quick recap: Modules

Load a module:

```
import numpy
x = numpy.linspace(0.0, 1.0, 100)
```

Load a module with an alias:

```
import numpy as np
x = np.linspace(0.0, 1.0, 100)
```

Load specific functions into global namespace:

```
from numpy import linspace
x = linspace(0.0, 1.0, 100)
```

Load whole module into global namespace:

```
from numpy import *
x = linspace(0.0, 1.0, 100)
```

Array creating

```
from numpy import *
# in longer programs, better use
# import numpy as np

x = arange(10)  # integer array
# enforce double:
x = arange(2, 16, 2, dtype='float')

x = zeros(10)  # gives double arrays
x = ones(12)
x = zeros((10,10))  # 2D array

x = linspace(0.0, 2.0*pi, 10)
# compare with:
x = linspace(0.0, 2.0*pi, 10, endpoint=False)

x = logspace(-5.0, 5.0, 100) # 10^-5 ... 10^5
```

Array computations

```
x = arange(10)
y = arange(2, 22, 2)
print len(y)  # should also be 10
print x+y
z = 2*x-y
print z, abs(z)
print x**2
print sin(y)

print sum(x)
print mean(x), var(x)  # mean, variance

x = arange(10*10)
x = reshape(x, (10, 10))  # make x it 2D
print dot(x , y )  # matrix-vector product
```

Array Indexing

```
print x[0]
# negative indexes count from the end
print x[-1]

for i in arange(len(x)):
    print x[i]

x = ones((10, 10))          # 2D array
print x[0,1], x[-1,-1] # 2D indexing
```

Array Slicing

```
x = linspace(0.0, 1.0, 100)
print x[1:]       # start at 1
print x[:-1]      # end 1 before last
print x[1:-1:2] # every second

print x[1:]-x[:-1]

# index arrays:
indices = [0, 5, 7, 10]
print x[indices]

# boolean arrays as indices
indices = x > 0.5
print indices
print x[indices]
```

Array Exercises

Note: Prefer array computations over `for` loops, as they are 100–300 times faster!

Array Exercises

Note: Prefer array computations over `for` loops, as they are 100–300 times faster!

Given a function f and a discretized interval x:

1. Write a function that computes the numerical derivative $(f(x+h) - f(x))/h$ for all values in x.

2. Write a function that returns the positions of all local minima in x.

Here, a discretized interval means an array of equidistantly spaced values, e.g. obtained from `x = linspace(0.0, 10.0, 100)`.

Hint: Both functions can be implemented without loops.

Forward derivative

```
def deriv_fwd(f, x):
    """Computes the derivative of f at x. """
    h = x[1]-x[0]  # get the step size
    dx = zeros_like(x)  # array for result
    # compute the first N-1 values at once
    dx[:-1] = (f(x[1:])-f(x[:-1]))/h
    # the last one separately
    dx[-1] = (f(x[-1]+h)-f(x[-1]))/h
    return dx

N = 1000  # number of discretization points
x = linspace(0.0, 10.0, N)
dx = deriv_fwd(sin, x)
error = sum(abs(dx-cos(x)))/len(x)
print "Numerical error of deriv:", error
```

Local Minima

```
def get_minima(f, x):
    """ finds local minima of f """
    f_x = f(x[1:-1])     # values
    f_prev = f(x[:-2])   # values before
    f_next = f(x[2:])    # values after
    # local minima: value before and value
    # after is larger
    indices = logical_and(f_x < f_prev,
                          f_x < f_next)
    x = x[1:-1]  # the values used above
    return x[indices]

x = linspace(0.0, 15.0, N)
print "Local minima:", get_minima(sin, x)
print "Expected:", 1.5*pi, 3.5*pi
```

Part III: Plotting

Matplotlib

`matplotlib` is a Python library for generating 2D plots with a MATLAB inspired syntax.

www.matplotlib.org

Matplotlib

`matplotlib` is a Python library for generating 2D plots with a MATLAB inspired syntax.

www.matplotlib.org

Minimal example:

```
from numpy import *
import matplotlib.pyplot as plt

x = linspace(0, 2*pi, 100)
plt.plot(x, sin(x))   # plot a sin graph
plt.plot(x, cos(x))   # plot a cos graph
plt.show()            # show the plot
```

Plot styles

```
x = linspace(0, 2*pi, 20)
# red circle markers
plt.plot(x, sin(x), 'or')
# blue crosses with lines
plt.plot(x, cos(x), '-xb')
# black dashed lines
plt.plot(x, sin(x) - cos(x), '--k')
plt.show()
```

```
# marker size and line width
plt.plot(x, sin(x), 'o-r',
    markersize=10, linewidth=5)
plt.show()
```

Plot styles

```
x = linspace(0, 2*pi, 20)
# red circle markers
plt.plot(x, sin(x), 'or')
# blue crosses with lines
plt.plot(x, cos(x), '-xb')
# black dashed lines
plt.plot(x, sin(x) - cos(x), '--k')
plt.show()
```

```
# marker size and line width
plt.plot(x, sin(x), 'o-r',
    markersize=10, linewidth=5)
plt.show()
```

Many other possibilities, check the `matplotlib` website.

Axis and legend

```
x = linspace(0, 2*pi, 100)
plt.plot(x, sin(x), label="sin")
plt.plot(x, cos(x), label="cos")
plt.plot(x, sin(x) - cos(x), label="sin-cos")

# axis bounds x=0..5, y=-1.5..1.5
plt.axis([0.0, 5.0, -1.5, 1.5])
plt.xlabel("x")
plt.ylabel("y")
plt.title("Example Plot")
plt.legend(loc="upper right")
plt.show()
```

Suplots

```
x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```

Much more...

- Animations
- Figure size
- Text, annotations
- Export to png, jpg, eps, pdf, ...
- Font size, style
- Latex symbols
- ...

Check the gallery on www.matplotlib.org.

Final exercise: Precision of numerical derivative

Different numerical methods for approximate derivatives:

1. forward difference: $f'(x) = \frac{1}{h}\left(f(x+h) - f(x)\right) + \mathcal{O}(h)$
2. central difference: $f'(x) = \frac{1}{2h}\left(f(x+h) - f(x-h)\right) + \mathcal{O}(h^2)$
3. extrapolated difference:

$$f'(x) = \frac{1}{6h}\left[8\left(f(x+\frac{h}{2}) - f(x-\frac{h}{2})\right) - \left(f(x+h) - f(x-h)\right)\right] + \mathcal{O}(h^4)$$

.

- Compute the derivate of $\sin(x)$ at the point $x_0 = \frac{\pi}{6}$ using the three methods above for different values of $h = 10^{-14} \ldots 1$.
- Compute the *relative* error to the analytic result and visualize this error in a double-logarithmic plot.
- Additionally, plot the expected scaling behavior h^α in the same figure. Identify the optimal value h (order of magnitude).