Machine Learning - Exercise 1

Introduction to Python and NumPy

Introduction to Python

- Python is a powerful, flexible programming language you can use for scientific computing, in web/Internet development, to write desktop graphical user interfaces (GUIs), create games, and much more.
- Python is an high-level, interpreted, object-oriented language written in C, which means it is compiled on-the-fly, at run-time execution.
- Its syntax is close to C, but without prototyping (whether a variable is an integer or a string will be automatically determined by the context).
- It can be executed either directly in an interpreter or through a script.
- To start the Python interpreter, simply type its name in a terminal under Linux:

```
$  python
Python 2.7.10 (default, Sep  7 2015, 13:51:49)
[GCC 5.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- The documentation on Python can be found at http://docs.python.org.

Basic syntax

Hello World!

- When the interpreter is started, the classical `Hello World!` example is fairly easy. Simply type:

```
>>> print 'Hello World'
Hello World
```

in Python 2, and:

```
>>> print('Hello World')
Hello World
```

in Python 3.

- You can also use the Python 3 style with Python 2 if you declare atthe beginning:

```
from __future__ import print_function
```

Hello World!

- The **print** command simply prints the following arguments on the standard output. A string in Python can be surrounded by either single or double quotes.
- Several high-level operations are possible on strings, such as concatenation:

```
>>> print('Hello' + ' ' + 'World')
Hello World
```

using the + operator, or even more intelligently using the comma operator:

```
>>> print('Hello', 'World')
Hello World
```

which adds automatically a space between the two strings.

- This form is particularly useful as it takes not only strings, but also integers, floats or even objects:

```
>>> pi = 3.14159
>>> print('The value of pi is around:', pi)
The value of pi is around: 3.14159
```

Types are implicit

- As Python is an interpreted language, variables can be assigned without specifying their type: it will be inferred at execution time.
- The only thing that counts is how you initialize them and which operations you perform on them.

```
>>> a = 42 # Integer

>>> b = 3.14159 # Double precision float

>>> c = 'My string' # String

>>> d = False # Boolean

>>> e = True# Boolean
```

Operations

- All the usual C operations (+, -, *, /, =, ==, !=, >, >=, etc) are available:

```
>>> f = a + 12
>>> g = c + ' is bigger'
>>> h = f >= 30
>>> print(h)
True
```

- Integer operands are converted to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

- Warning: if you perform an illegal operation, an error will be raised:

```
>>> i = 12 + 'My string'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Usage as a script

- Instead of using the interpreter, you can run scripts which can be executed sequentially.
- Simply edit a text file called `MyScript.py` containing for example:

```python
# MyScript.py
# Implements the Hello World example.

text = 'Hello World!'
print(text)
```

- To execute this script, type in a Terminal:

```
$ python MyScript.py
Hello World!
```

- As it is a scripted language, each instruction in the script is executed from the beginning to the end, except for the declared functions which can be used later.
- The # character is used for comments.

Functions

- As in most procedural languages, you can define functions:

```python
def hello_world(msg):
    print(msg)

text='Hello World!'
hello_world(text)
```

- Functions are defined by the keyword `def`. Only the parameters of the function are specified (without type), not the return values.
- The main particularity of the Python syntax is that the scope of the different structures (functions, for, if, while, etc...) is defined by the indentation.
- A reasonable choice is to use four spaces for the indentation instead of tabs (configure your text editor).

Functions

- Functions can have several parameters (with default values) and return values. The name of the parameter can be specified during the call, so their order won't matter.

```python
# import the math package
from math import *

def cos_sin(value, freq, phase=0):
    '''
    Returns the cosine and sine functions of a value, given a frequency and a phase.
    '''
    return cos(2*pi*freq*value+phase), sin(2*pi*freq*value+phase)

v = 1.7
f = 4
p = pi/2
print(cos_sin(v, f))
print(cos_sin(value=v, freq=f))
print(cos_sin(value=v, phase=p, freq=f))
```

- Usually, the first part of the script will be the libraries to import, the second part will be the definition of functions, and the code to be executed is placed at the end.

Lists

- Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

- Like in C, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0] # First element
'spam'
>>> a[3] # Fourth element
1234
>>> a[-2] # Negative indices starts from the last element
100
>>> a[1:-1] # Second until last element
['eggs', 100]
>>> a[:2] + ['bacon', 2*2] # Lists can be concatenated
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!'] # Lists can be repeated
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Lists

- Lists are objects, with a lot of different methods (type `help(list)`):
    - list.append(x): Add an item to the end of the list.
    - list.extend(L): Extend the list by appending all the items in the given list.
    - list.insert(i, x): Insert an item at a given position.
    - list.remove(x): Remove the first item from the list whose value is x.
    - list.pop([i]): Remove the item at the given position in the list, and return it.
    - list.index(x): Return the index in the list of the first item whose value is x.
    - list.count(x): Return the number of times x appears in the list.
    - list.sort(): Sort the items of the list, in place.
    - list.reverse(): Reverse the elements of the list, in place.

Lists

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Conditional statement

- Perhaps the most well-known statement type is the `if` statement. For example:

```
x = 3
if x < 0 :
     print 'x is negative'
elif x == 0:
     print 'x is zero'
else:
     print 'x is positive'
```

- There can be zero or more elif parts, and the else part is optional.
- The keyword `elif` is short for `else if`, and is useful to avoid excessive indentation.
- An `if ... elif ... elif ...` sequence is a substitute for the switch or case statements found in other languages.

FOR statement

- The for statement in Python differs a bit from what you may be used to in C or Pascal.
- Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order they appear in the sequence.

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

# cat 3
# window 6
# defenestrate 12
```

FOR statement

- If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
range(10)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in range(10):
   print(i)
# 0 1 2 3 4 5 6 7 8 9
```

- The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10.

- It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the `step`):

```
range(5, 10) # [5, 6, 7, 8, 9]
range(0, 10, 3) # [0, 3, 6, 9]
range(-10, -100, -30) # [-10, -40, -70]
```

FOR statement

- To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
list_words = ['Mary', 'had', 'a', 'little', 'lamb']
for idx in range(len(list_words)):
    print(idx, list_words[idx])

# 0 Mary
# 1 had
# 2 a
# 3 little
# 4 lamb
```

- The `enumerate` function allows to get at the same time the index and the content:

```
list_words = ['Mary', 'had', 'a', 'little', 'lamb']
for idx, word in enumerate(list_words):
    print(idx, word)
```

Dictionaries

- Another useful data type built into Python is the dictionary. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.

- Dictionaries are defines by curly braces instead of squared brackets. Content is defined by key:value pairs:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127

print(tel)
# {'sape': 4139, 'guido': 4127, 'jack': 4098}
```

- The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the sorted() function to it). To check whether a single key is in the dictionary, use the in keyword.

```
tel.keys()
# ['guido', 'irv', 'jack']

'guido' in tel
# True
```

Resources

Many resources to learn Python exist on the Web:

- The official documentation on Python can be found at http://docs.python.org.
- Free book Dive into Python.
- Learn Python the hard way.
- Learn Python on Code academy.
- Scipy lectures note http://www.scipy-lectures.org
- An Introduction to Interactive Programming in Python on Coursera.

Exercise

In cryptography, a Caesar cipher is a very simple encryption techniques in which each letter in the plain text is replaced by a letter some fixed number of positions down the alphabet. For example, with a shift of 3, A would be replaced by D, B would become E, and so on. The method is named after Julius Caesar, who used it to communicate with his generals. ROT-13 ("rotate by 13 places") is a widely used example of a Caesar cipher where the shift is 13. In Python, the key for ROT-13 may be represented by means of the following dictionary:

```
code = {'a':'n', 'b':'o', 'c':'p', 'd':'q', 'e':'r', 'f':'s', 'g':'t', 'h':'u',
        'i':'v', 'j':'w', 'k':'x', 'l':'y', 'm':'z', 'n':'a', 'o':'b', 'p':'c',
        'q':'d', 'r':'e', 's':'f', 't':'g', 'u':'h', 'v':'i', 'w':'j', 'x':'k',
        'y':'l', 'z':'m', 'A':'N', 'B':'O', 'C':'P', 'D':'Q', 'E':'R', 'F':'S',
        'G':'T', 'H':'U', 'I':'V', 'J':'W', 'K':'X', 'L':'Y', 'M':'Z', 'N':'A',
        'O':'B', 'P':'C', 'Q':'D', 'R':'E', 'S':'F', 'T':'G', 'U':'H', 'V':'I',
        'W':'J', 'X':'K', 'Y':'L', 'Z':'M'}
```

Your task in this exercise is to implement an encoder/decoder of ROT-13. Once you're done, you will be able to read the following secret message:

```
BZT! guvf vf fb obevat.
```

NumPy numerical library

NumPy numerical library

- NumPy is a linear algebra library written in Python.
- The reference manual is at http://docs.scipy.org/doc
- A nice tutorial can be found at https://docs.scipy.org/doc/numpy-dev/user/quickstart.html.
- A detailed list of all available functions (with examples) is at https://docs.scipy.org/doc/numpy/reference/routines.html.
- If you already know Matlab, a comparison is available at <>.

Importing the library

- You can simply write at the beginning of your script:

```
from numpy import *
```

- All the availables functions are then imported into your working space.
- It is often better to keep a short namespace, as it could overload other library definitions:

```
import numpy as np
```

- You can get help on any Numpy function:

```
help(np.array)
help(np.array.transpose)
```

Vectors and matrices

- The basic object in NumPy is an array with d-dimensions (1 = vector, 2 = matrix, etc...). They can store either integers or floats.
- In order to create a vector of three floats, you simply have to write:

```
v = np.array( [ 1., 2., 3.] )
```

- Each initial element of the vector has to be given inside a Python list. For a 3*4 matrix of 8 bits unsigned integers, it is:

```
A = np.array( [ [ 1, 2, 3, 4],
                [ 5, 6, 7, 8],
                [ 4, 3, 2, 1] ], dtype=np.uint8)
```

- Most of the time you don't care about the type in Python (the default floating-point precision is what you want), but if you need it, you can always specify it with the parameter dtype={int32, uint16, float64}.

Vectors and matrices

- Matrices should be initialized with a list of lists. The elements of the array are internally stored in a sequence, so you can easily reshape vectors or matrices:

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

A.shape
# (8,)

A.reshape(2,4)
# array([[1, 2, 3, 4],
#        [5, 6, 7, 8]])
```

Attributes of an array

The following attributes of an array can be accessed:

- `A.shape` : returns the shape of the vector or matrix (row, column)
- `A.size` : returns the total number of elements in the array
- `A.ndim` : returns the number of dimensions of the array (vector: 1, matrix:2)
- `A.dtype.name` : returns the type of data stored in the array (int32, uint16, float64...)
- `A.itemsize` : returns the size in bytes of each element.

Initialization of an array

Here are some specific initializations for vectors/matrices:

```python
# A 2*3 matrix filled with 0.0
A = np.zeros((2,3))

# A vector of 12 elements initialized to 1.0
v = np.ones(12)

# A vector of 11 elements whose value
# linearly increases from 0.0 to 1.0
v = np.linspace(0.0, 1.0, 11)

# A 10*10 matrix with values randomly taken
# from an uniform distribution between 0.0 and 1.0.
A = np.random.uniform(0.0, 1.0, (10, 10))

# A 10*10 matrix with values randomly taken
# from a normal (gaussian) distribution
# with a mean of 0.5 and standard deviation of 1.0.
A = np.random.normal(0.5, 1.0, (10, 10))
```

Manipulation of matrix indices

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

- To access a particular element of a matrix, you can use the usual Python list style (the first element has a rank of 0):

```
# The element on the first row and third column
A[0, 2]
# 3

# The second column of A
A[:, 1]
# array([ 2, 6])

# The two middle columns of A
A[:, 1:3]
# array(  [ 2, 3],
#         [ 6, 7])
```

Manipulation of matrix indices

- To perform conditional manipulation of matrix content, one can retrieve the indices easily:

```
A = np.array(
    [ [ -1,  1,   1, -1],
      [  1, -1, -1, -1] ])

negatives = A < 0

A[negatives] = 0

print (A)
# [[0 1 1 0]
#  [1 0 0 0]]
```

or even more simply:

```
>>> A[A < 0] = 0
```

Basic linear algebra

- Let's define some matrices:

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

B = np.array( [ [ 1, 2],
                [ 3, 4],
                [ 5, 6],
                [ 7, 8] ])

C = np.array( [ [  1,  2,  3,  4],
                [  5,  6,  7,  8],
                [  9, 10, 11, 12],
                [ 13, 14, 15, 16] ])
```

Transpose a matrix

- A matrix can be transposed with the `transpose()` method or the `.T` shortcut:

```
D = A.transpose()

# [[1 5]
#  [2 6]
#  [3 7]
#  [4 8]]

D = A.T # equivalent but simpler

# [[1 5]
#  [2 6]
#  [3 7]
#  [4 8]]
```

- `transpose()` does not change A, only the returned value.

Multiply two matrices element-wise

```
A = np.array( [ [ 1, 2, 3, 4],
                [ 5, 6, 7, 8] ])
B = np.array( [ [ 1, 2],
                [ 3, 4],
                [ 5, 6],
                [ 7, 8] ])
```

- Two matrices of **exacty** the same size can be multiplied *element-wise* by using the * operator.

```
C = A * B.T

# [[ 1,  6, 15, 28],
#  [10, 24, 42, 64]]
```

Multiply two matrices algebrically

```
A = np.array( [ [ 1, 2, 3, 4],
                [ 5, 6, 7, 8] ])
B = np.array( [ [ 1, 2],
                [ 3, 4],
                [ 5, 6],
                [ 7, 8] ])
```

- To perform a matrix multiplication, you have to use the `dot()` method:

```
C = np.dot(A, B)

# [[ 50,  60],
#  [114, 140]]


D = np.dot(B, A)

# [[11, 14, 17, 20],
#  [23, 30, 37, 44],
#  [35, 46, 57, 68],
#  [47, 62, 77, 92]]
```

- One dimension must match!

  (m, n) * (n, p) = (m, p)

Summing elements

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

- One can sum the elements of a matrix globally, row-wise or column-wise:

```
# Globally
np.sum(A)
# 36

# Per column
np.sum(A, axis=0)
# [ 6  8 10 12]

# Per row
np.sum(A, axis=1)
# [10 26]
```

Mathematical operations

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

You can apply any usual mathematical operations (cos, sin, exp, etc...) on each element of a matrix:

```
np.exp(A)

# [[  2.71828183e+00,   7.38905610e+00,   2.00855369e+01, 5.45981500e+01],
#  [  1.48413159e+02,   4.03428793e+02,   1.09663316e+03, 2.98095799e+03]]

np.cos(A)

# [[ 0.54030231, -0.41614684, -0.9899925 , -0.65364362],
#  [ 0.28366219,  0.96017029,  0.75390225, -0.14550003]]

np.log(A)

# [[ 0.        ,  0.69314718,  1.09861229,  1.38629436],
#  [ 1.60943791,  1.79175947,  1.94591015,  2.07944154]]
```

Inversing a matrix (when possible)

```
C = np.array( [ [  1,   2,   3,   4],
                [  5,   6,   7,   8],
                [  9, 10, 11, 12],
                [ 13, 14, 15, 16] ])
```

The `inv()` method is in the `linalg` submodule of NumPy:

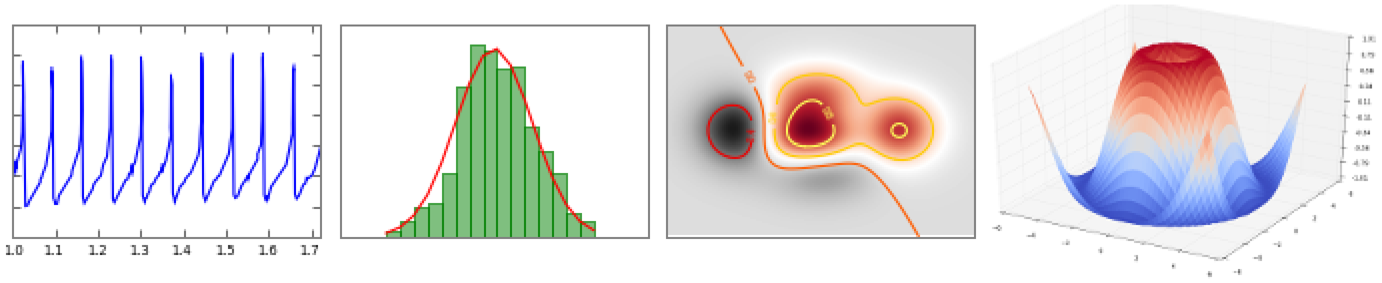
```
D = np.linalg.inv(C)

# [[  6.37193232e+16, -7.70045532e+16, -3.71488632e+16,  5.04340932e+16],
#  [ -5.91679430e+16,  7.27909707e+16,  3.19218875e+16, -4.55449153e+16],
#  [ -7.28220837e+16,  8.54317182e+16,  4.76028147e+16, -6.02124492e+16],
#  [  6.82707035e+16, -8.12181357e+16, -4.23758390e+16,  5.53232712e+16]]
```

Matplotlib

Plotting arrays with Matplotlib

- Matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.



- Reference: http://matplotlib.org
- Tutorial by N. Rougier: http://www.labri.fr/perso/nrougier/teaching/matplotlib
- Simply imported with :

```
import matplotlib.pyplot as plt
```

- In the B202, one has to explicitly specify the backend:

```
import matplotlib
matplotlib.use('TKAgg')
import matplotlib.pyplot as plt
```

Simple example

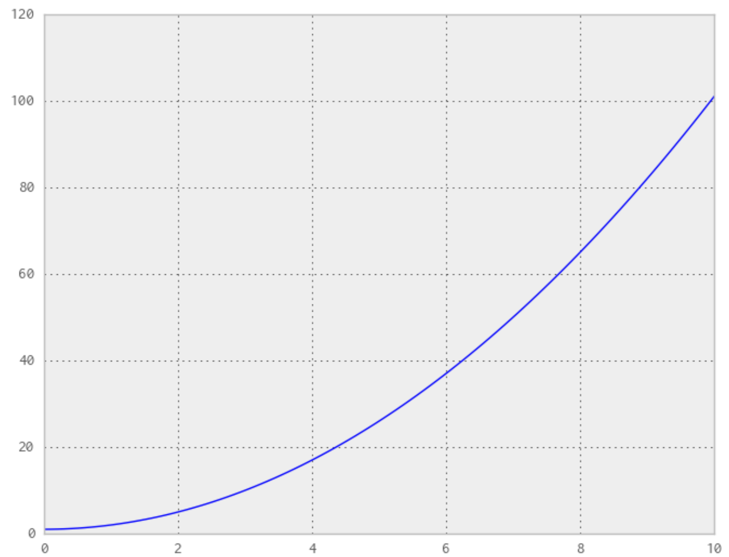
Plot some quadratic function of x on [0, 10]

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

X = np.linspace(0., 10., 100)
Y = X**2 + 1.

plt.plot(X, Y, '-')
plt.show()
```

- `plot()` takes two vectors X and Y and a style (here a line '-', but colored dots are possible).
- The call to `show()` is obligatory at the end to



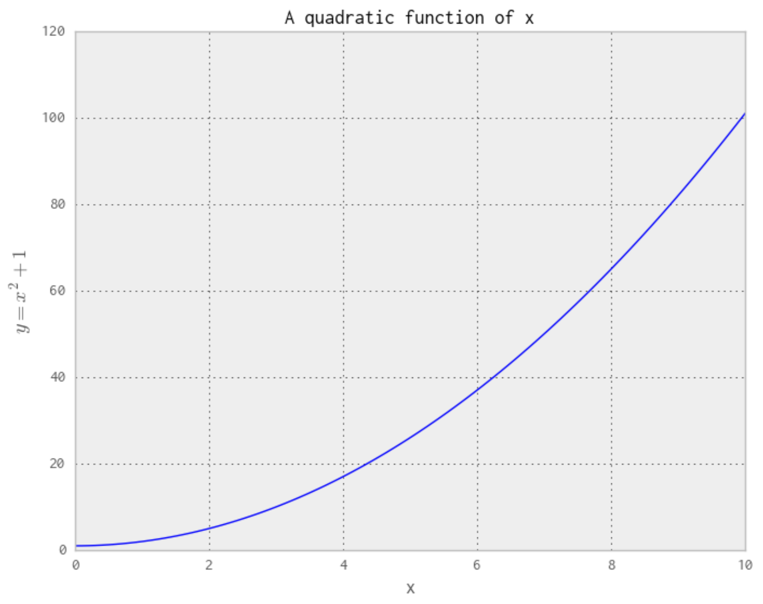
- The call to show() is obligatory at the end to display the window!

Nicer axes

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

X = np.linspace(0., 10., 100)
Y = X**2 + 1.

plt.plot(X, Y, '-')
plt.title('A quadratic function of x')
plt.xlabel('x')
plt.ylabel('$y = x^2 + 1$')
plt.show()
```

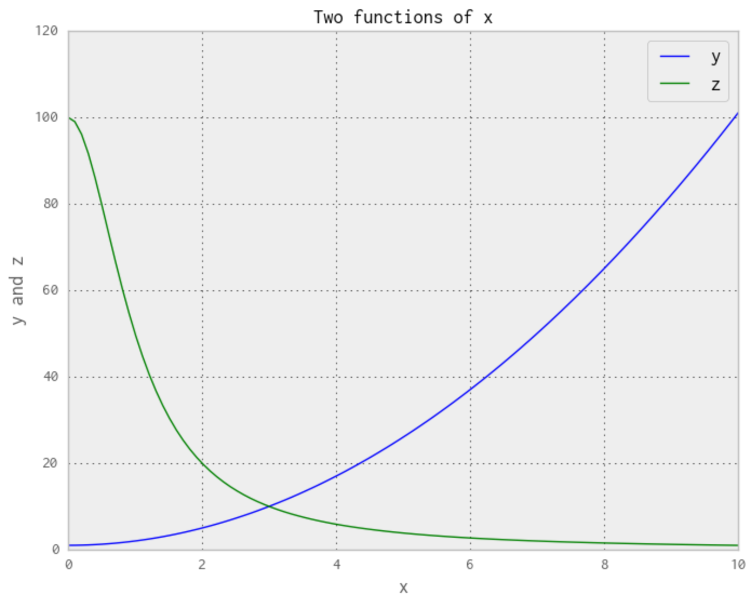


Two plots with a legend

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

X = np.linspace(0., 10., 100)
Y = X**2 + 1.
Z = 100./(X**2 + 1.)

plt.plot(X, Y, '-', label='y')
plt.plot(X, Z, '-', label='z')
plt.title('Two functions of x')
plt.xlabel('x')
plt.ylabel('y and z')
plt.legend()
plt.show()
```

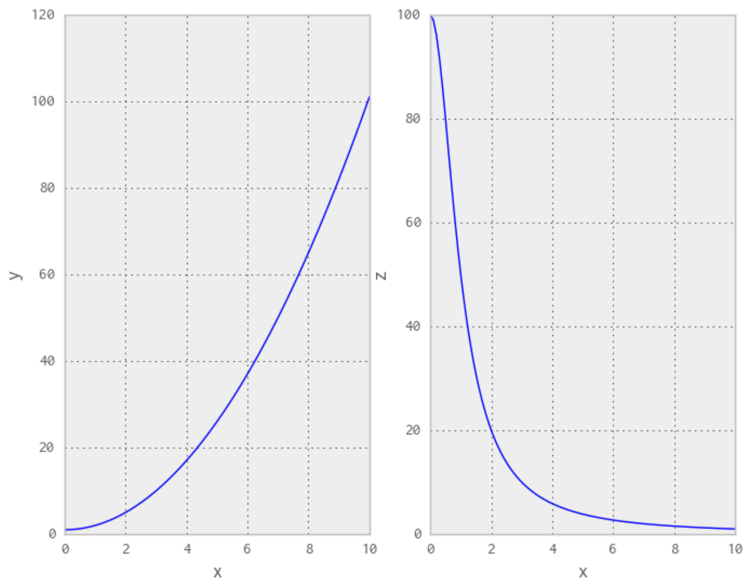


Two plots side by side

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

X = np.linspace(0., 10., 100)
Y = X**2 + 1.
Z = 100./(X**2 + 1.)

plt.subplot(121)
plt.plot(X, Y, '-', label='y')
plt.xlabel('x')
plt.ylabel('y')
plt.subplot(122)
plt.plot(X, Z, '-', label='z')
plt.xlabel('x')
plt.ylabel('z')
plt.show()
```



More features

- Display images with `imshow()`
- Histograms with `hist()`
- Animations, movies...

Exercises

Basic matrix manipulations

- Write the following system of equations in the matrix form ($A \cdot X = B$) and solve it using NumPy. Check that it worked.

$$\begin{cases} 2 \cdot x_1 + x_2 + x_3 = 9 \\ x_1 + 2 \cdot x_2 + x_4 = 8 \\ x_1 + 2 \cdot x_2 + 3 \cdot x_3 + 2 \cdot x_4 = 7 \\ 2 \cdot x_2 + x_3 + 2 \cdot x_4 = 6 \end{cases}$$

- What are the eigenvalues and eigenvectors of the matrix A defined above? (tip: read the doc for the function `np.linalg.eigh`)

Basic matrix manipulations

- Write a function that rotates a 2D vector from an angle θ.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$