TD1 Yubo Cai

2022 年 4 月 7 日

Bachelor of Ecole Polytechnique

MAA106: Introduction to Numerical Analysis, year 1, semester 2

Maxime Breden

Based on Notebooks created by Aline Lefebvre-Lepot

1 Introduction to Numpy and Matplotlib

This tutorial gives a quick overview on how to use the python packages Numpy and Matplotlib.

1.1 Table of contents

## Numpy

NumPy is the fundamental package for scientific computing with Python (see http://www.numpy.org/). It contains powerful N-dimensional array objects, linear algebra, Fourier transform, random numbers and also sophisticated tools for integrating C++ and Fortran code.

In this class, we are going to use this package essentially to manipulate arrays. Lot of documentations about numpy can be found on the web, we give below basic examples that will be useful in the following.

First, you need to import the numpy package:

[1]:
```python
import numpy as np
```

1.1.1 Creating arrays

[2]:
```python
# One-dimensional arrays
a = np.array([0, 1, 2, 3])
print('a =',a)
print('a.shape =',a.shape)
print('a.size =',a.size)
```

```
a = [0 1 2 3]
a.shape = (4,)
a.size = 4
```

[3]:
```python
# Two-dimensional arrays
L1 = [0, 1, 2, 3]
L2 = [4, 5, 6, 7]
a = np.array([L1, L2])
print('a =', a)
print('a.shape =', a.shape)
print('a.size =', a.size)
```

```
a = [[0 1 2 3]
 [4 5 6 7]]
a.shape = (2, 4)
a.size = 8
```

You can use tabulation in notebooks for autocompletion or to obtain the list of possible completions. You can also obtain interactive help:

[4]:
```python
np.array?
```

```
Docstring:
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)

Create an array.

Parameters
----------
```

```
object : array_like
    An array, any object exposing the array interface, an object whose
    __array__ method returns an array, or any (nested) sequence.
dtype : data-type, optional
    The desired data-type for the array.  If not given, then the type will
    be determined as the minimum type required to hold the objects in the
    sequence.
copy : bool, optional
    If true (default), then the object is copied.  Otherwise, a copy will
    only be made if __array__ returns a copy, if obj is a nested sequence,
    or if a copy is needed to satisfy any of the other requirements
    (`dtype`, `order`, etc.).
order : {'K', 'A', 'C', 'F'}, optional
    Specify the memory layout of the array. If object is not an array, the
    newly created array will be in C order (row major) unless 'F' is
    specified, in which case it will be in Fortran order (column major).
    If object is an array the following holds.

    ===== ========= ===================================================
    order  no copy                     copy=True
    ===== ========= ===================================================
    'K'   unchanged F & C order preserved, otherwise most similar order
    'A'   unchanged F order if input is F and not C, otherwise C order
    'C'   C order   C order
    'F'   F order   F order
    ===== ========= ===================================================

    When ``copy=False`` and a copy is made for other reasons, the result is
    the same as if ``copy=True``, with some exceptions for `A`, see the
    Notes section. The default order is 'K'.
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be pre-pended to the shape as
    needed to meet this requirement.
```

```
like : array_like
    Reference object to allow the creation of arrays which are not
    NumPy arrays. If an array-like passed in as ``like`` supports
    the ``__array_function__`` protocol, the result will be defined
    by it. In this case, it ensures the creation of an array object
    compatible with that passed in via this argument.

    .. note::
        The ``like`` keyword is an experimental feature pending on
        acceptance of :ref:`NEP 35 <NEP35>`.

    .. versionadded:: 1.20.0

Returns
-------
out : ndarray
    An array object satisfying the specified requirements.

See Also
--------
empty_like : Return an empty array with shape and type of input.
ones_like : Return an array of ones with shape and type of input.
zeros_like : Return an array of zeros with shape and type of input.
full_like : Return a new array with shape of input filled with value.
empty : Return a new uninitialized array.
ones : Return a new array setting values to one.
zeros : Return a new array setting values to zero.
full : Return a new array of given shape filled with value.


Notes
-----
When order is 'A' and `object` is an array in neither 'C' nor 'F' order,
and a copy is forced by a change in dtype, then the order of the result is
not necessarily 'C' as expected. This is likely a bug.

Examples
```

```
--------
>>> np.array([1, 2, 3])
array([1, 2, 3])

Upcasting:

>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])

More than one dimension:

>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

Minimum dimensions 2:

>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])

Type provided:

>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])

Data-type consisting of more than one element:

>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])

Creating an array from sub-classes:

>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
Type:      builtin_function_or_method
```

There exists several functions to create arrays:

[5]:
```
# imposing the number of elements
c = np.linspace(0, 1, 6)   # start, end, number of points
print('c =', c)
```

```
c = [0.  0.2 0.4 0.6 0.8 1. ]
```

[6]:
```
# imposing the step between the elements
a = np.arange(10) # 0 .. n-1  (!) start=0 and step=1 by default
b = np.arange(1, 9, 2) # start, end (exclusive), step
c = np.arange(1, 2, 0.1) # start, end (exclusive), step
print('a =', a)
print('b =', b)
print('c =', c)
```

```
a = [0 1 2 3 4 5 6 7 8 9]
b = [1 3 5 7]
c = [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
```

[7]:
```
# arrays of 0's or 1's with given size
a = np.zeros(2) # 1 dimensional, number of elements
b = np.ones(3) # 1 dimensional, number of elements
c = np.zeros((2, 3)) # 2-dimensional, tuple (number of rows, number of columns)
print('a =', a)
print('b =', b)
print('c =', c)
```

```
a = [0. 0.]
b = [1. 1. 1.]
c = [[0. 0. 0.]
 [0. 0. 0.]]
```

[8]:
```
# random initialization
a = np.random.rand(4) # uniform in [0,1]
```

```
b = np.random.randn(4) # gaussian
print('a =', a)
print('b =', b)
```

```
a = [0.75341254 0.86129285 0.70865046 0.44282376]
b = [-1.23850026 -0.71283446  0.43590017  0.66430625]
```

- Create a null vector of size 10.
- Create a vector with values ranging from 10 to 49.
- create a vector with values ranging from 0 to 1 with 100 points.
- Create a 3x3 random matrix.
- print the size of the previous arrays.

```
[9]: a = np.array([0]*10)
     print('The answer of the question-1: ' + '\n' , a)
```

```
The answer of the question-1:
 [0 0 0 0 0 0 0 0 0 0]
```

```
[10]: b = np.arange(19,50)
      print('The answer of the question-2: ' + '\n' , b)
```

```
The answer of the question-2:
 [19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
 43 44 45 46 47 48 49]
```

```
[11]: c = np.arange(0, 1, 0.01)
      print('The answer of the question-3:' + '\n' , c)
```

```
The answer of the question-3:
 [0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13
 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41
 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55
 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97
 0.98 0.99]
```

```
[12]: d = np.random.random((3,3))
      print('The answer of the question-4:' + '\n' , d)
```

```
The answer of the question-4:
 [[0.58220577 0.03534246 0.56653464]
 [0.68943627 0.48097966 0.14406022]
 [0.16173301 0.25080886 0.31736754]]
```

```
[13]: print('The answer of the question-5: ')
      print('a.size =', a.size)
      print('b.size =', b.size)
      print('c.size =', c.size)
      print('d.size =', d.size)
```

```
The answer of the question-5:
a.size = 10
b.size = 31
c.size = 100
d.size = 9
```

1.1.2 Indexing of arrays

The elements are indexed from 0: - the first element has index 0 - the second element has index 1 etc…

The elements can also been indexed from the end: - the last element can be obtained using index -1 - the second to last can be obtained using index -2 etc…

```
[14]: # One dimensional arrays
      a = np.arange(8)
      print('a =', a)
      print('elements =', a[0], a[4], a[-1], a[-3])  # extract an elements, 0 =␣
       ↪first, 1 = second..., -1 = last
```

```
a = [0 1 2 3 4 5 6 7]
elements = 0 4 7 5
```

```
[15]: # Two dimensional arrays
      L1 = [0, 1, 2, 3, 4, 5]
      L2 = [10, 11, 12, 13, 14, 15]
```

```
a = np.array([L1, L2])
print(a)
print('element =', a[1, 3]) # extract an element, 0 = beginning, -1 = end (2nd␣
↪row, 4th column)
print('column =', a[:, 2]) # extract a column, 0 = beginning, -1 = end
print('row =', a[1, :]) # extract a row, 0 = beginning, -1 = end
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]]
element = 13
column = [ 2 12]
row = [10 11 12 13 14 15]
```

1.1.3 Slicing of arrays

Slicing of arrays is based on the following indexing where [: is the default start and :] the default end. Except if using the default end, the last element is excluded from the selection.

It is a powerful way of extracting sub-arrays from a given array.

- Successively uncomment the various extractions of the array a defined in the following cell, by using "ctrl+/" or by deleting the "#" symbol at the beginning of each line, to see how slicing works in practice.

[16]:
```
## Slicing one dimensional arrays
a = np.arange(10)
print('a =', a)
# print('extract 1 =', a[1:7:2])   # start, end (exclusive), step
# print('extract 2 =', a[1:7:])    # default step = 1
# print('extract 3 =', a[:7:2])    # default start = first element (= 0)
# print('extract 4 =', a[3::1])    # default end = until last element
# print('extract 5 =', a[3:-1:1])  # !!! if end = -1, the last element is␣
↪excluded (see figure)
# print('extract 6 =', a[6:2:-1])  # negative step

## short commands:
# print('extract 7 =', a[1:7])     # same result as extract 2: default step=1
# print('extract 8 =', a[3:])      # same result as extract 4: default step=1,␣
↪default end=last element
```

```
# print('extract 9 =', a[:3])      # default step=1, default start=1st element
```

```
a = [0 1 2 3 4 5 6 7 8 9]
```

[17]:
```
# Slicing two dimensional arrays
L1 = [0, 1, 2, 3, 4, 5]
L2 = [10, 11, 12, 13, 14, 15]
a = np.array([L1, L2])
print('a =', a)
print('extract =', a[0, ::2])
```

```
a = [[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]]
extract = [0 2 4]
```

[18]:
```
# slicing can also be used to assign values in an array
a = np.zeros(10)
a[::2] = 1
print('a =', a)
```

```
a = [1. 0. 1. 0. 1. 0. 1. 0. 1. 0.]
```

- Create a 2-d array of size 10x10, with 1's on the borders and 0's inside.

[19]:
```
mat1 = np.ones((10,10))
print("Original array:")
print(mat1)
print('\n')

mat1[1:-1,1:-1] = 0
print("New array:")
print(mat1)
```

```
Original array:
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]


New array:
[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

- Create a 8x8 matrix and fill it with a checkerboard pattern of 0's and 1's.

[20]:
```python
mat1 = np.zeros((8,8))
print("Original array:")
print(mat1)
print('\n')

mat1[1::2, ::2] = 1 # 第一个是横坐标，第二个是纵坐标，把偶数行的变成交叉棋盘
mat1[0::2, 1::2] = 1
print("New array:")
print(mat1)
```

```
Original array:
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]



New array:
[[0. 1. 0. 1. 0. 1. 0. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [0. 1. 0. 1. 0. 1. 0. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [0. 1. 0. 1. 0. 1. 0. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]
 [0. 1. 0. 1. 0. 1. 0. 1.]
 [1. 0. 1. 0. 1. 0. 1. 0.]]
```

- Create a random vector of size 5 and exchange the max and min values

[21]:
```python
mat = np.random.random(5)
print("Original array:")
print(mat)
print('\n')

num_max = max(mat)
num_min = min(mat)
index_max = mat.argmax()
index_min = mat.argmin()
mat[index_max] = num_min
mat[index_min] = num_max
print("New array:")
print(mat)
```

```
Original array:
[0.6792957  0.39087074 0.23891079 0.11619112 0.17158821]



New array:
[0.11619112 0.39087074 0.23891079 0.6792957  0.17158821]
```

1.1.4 Loops and computations on arrays

- Save your notebook…

Loop counters are designed so that one can scan an array using a.shape:

[22]:
```
a = np.zeros((2, 3))
print(a)
print(a.shape[0])
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        a[i, j] = (i + 1)*(j + 1)
print('a =', a)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
2
a = [[1. 2. 3.]
 [2. 4. 6.]]
```

However, numpy has built-in element-wise array operations, and one should typically avoid using loops on arrays when possible.

[23]:
```
a = np.linspace(0, 1, 10000000)
b = np.zeros(10000000)
for i in range(a.size): b[i] = 3*a[i] - 1
print("Done")
```

```
Done
```

[24]:
```
a = np.linspace(0, 1, 10000000)
b = np.zeros(10000000)
b[:] = 3*a - 1
print("Done")
```

```
Done
```

[25]:
```
# b = 3a-1 with a loop
a = np.linspace(0, 1, 100000)
b = np.zeros(100000)
%timeit for i in range(a.size): b[i] = 3*a[i] - 1
```

```
53.9 ms ± 5.49 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[26]:
```
# b = 3a-1 with arrays
a = np.linspace(0, 1, 100000)
b = np.zeros(100000)
%timeit b[:] = 3*a - 1
```

```
103 µs ± 11.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Numpy also provides various optimized functions as sin, exp... for arrays

[27]:
```
def f(x):
    return np.exp(-x*x)*np.log(1 + x*np.sin(x))

print("test 1")
x = np.linspace(0, 1, 100000)
a = np.zeros(x.size)
%timeit for i in range(x.size): a[i] = f(x[i])

print("test 2")
x = np.linspace(0, 1, 100000)
%timeit a = f(x)
```

```
test 1
399 ms ± 31.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
test 2
2.28 ms ± 395 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Save your notebook...
- Create two random vectors x and y of size $n = 10$, representing 10 cartesian coordinates (x_k, y_k).
- Convert them to polar coordinates (r_k, θ_k).
- Optimize and test your code for vectors x and y of size $n = 10^5$ (compare efficiency of a loop, list comprehension and numpy element-wise operations on arrays).

[28]:
```
x = []
y = []
for i in range(10):
    x.append(np.random.randn())
    y.append(np.random.randn())
print(x)
```

```python
print(y)
```

```
[0.08484854711490612, 0.45635108367308136, -0.10825794533288598,
-0.7295682413710257, 1.391713315107128, -0.9998047020768233,
-1.1263087577243358, -0.0645652651004269, 0.45589412358273995,
0.663370880252072]
[1.695703900133224, 0.18473057738646279, 0.6699904582145844, 0.6474156058816912,
-0.7881769863840389, -0.7168283204193191, -0.3416138772903736,
-1.1088335125392468, 0.3773791925849734, 1.8788960776756582]
```

[29]:
```python
polar = []
for i in range(10):
    rk = np.sqrt(x[i]**2 + y[i]**2)
    theta_k = np.arctan(y[i]/x[i])
    polar.append((rk, theta_k))

print("polar coordinates:")
print(polar)
```

```
polar coordinates:
[(1.6978253717254133, 1.5208006639503446), (0.4923227577424505,
0.3846368044121827), (0.6786803347867735, -1.4105995934914175),
(0.9754059593607006, -0.725807508051048), (1.59940267391045,
-0.5152984786695023), (1.2302244849010717, 0.6220235682733026),
(1.1769755557716137, 0.2944851636471979), (1.1107116781539719,
1.512633904120331), (0.5918230368221239, 0.6914508355621198),
(1.9925640254384256, 1.2313943487913608)]
```

[30]:
```python
x = np.random.random(100000)
y = np.random.random(100000)
%timeit r = np.sqrt(x**2+y**2)
%timeit theta = np.arctan2(y,x)
```

```
281 µs ± 63.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.03 ms ± 178 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

1.2 Matplotlib

Matplotlib is a Python package for 2 dimensional plots. It can be used to produce figures inline in jupyter notebooks. Lot of documentations about numpy can be found on the web, we give below basic examples that will be useful in the following.

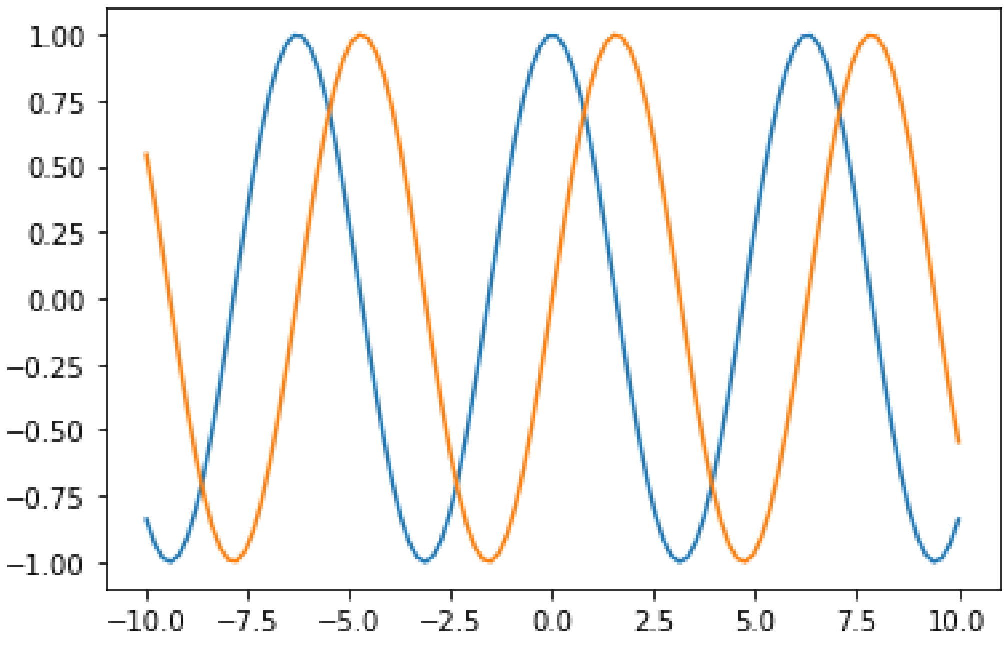
First, we need to import the matplotlib package.

[31]:
```
# In order to display plots inline:
%matplotlib inline

# load the libraries
import matplotlib.pyplot as plt # 2D plotting library
import numpy as np              # package for scientific computing
```

1.2.1 First plot

We plot on the same figure sine and cosine on the interval $[-\pi, \pi]$.

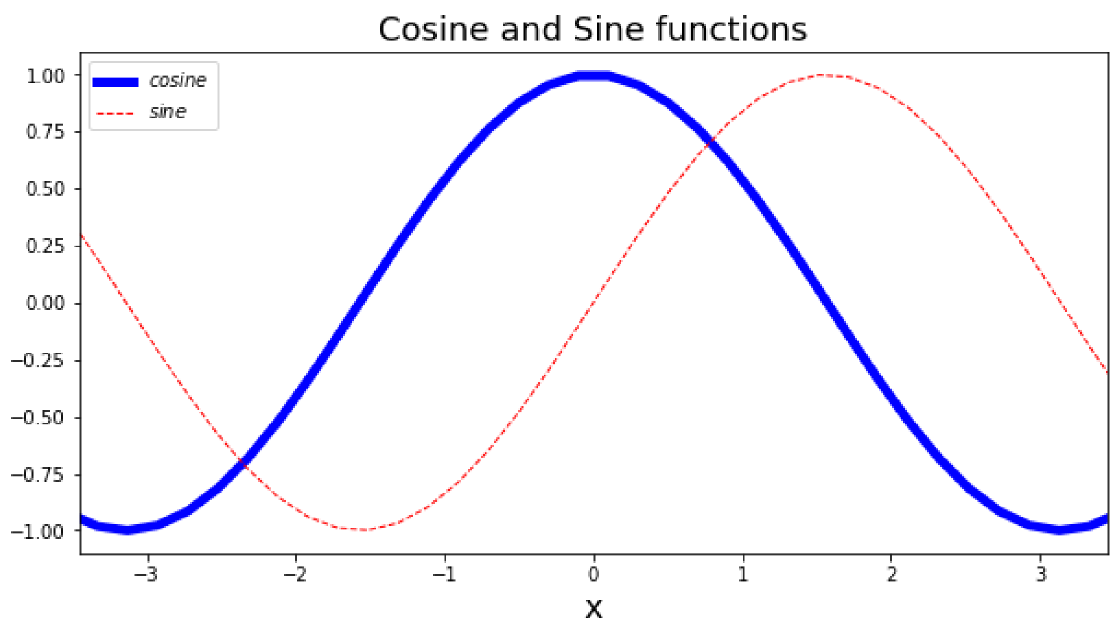
[34]:
```
X = np.linspace(-10, 10, 100)
COS, SIN = np.cos(X), np.sin(X)
plt.plot(X,COS)
plt.plot(X, SIN)
plt.show()
```



1.2.2 Modifying properties of the figure

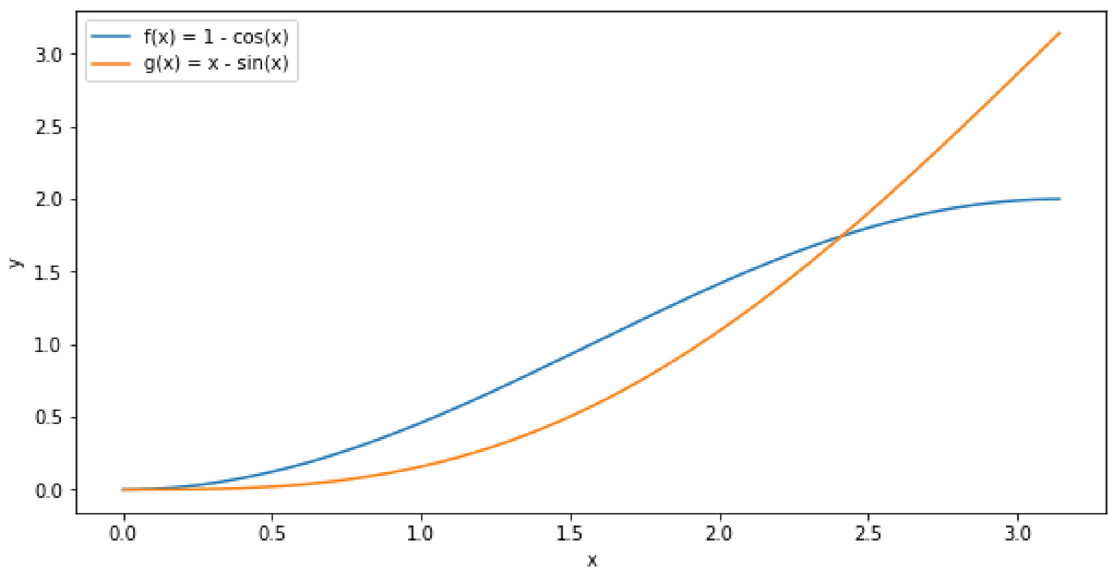
Most of the properties of a figure can be customized: color and style of the graphs, title, legend, labels… Bellow, we plot on the same figure sine and cosine on the interval $[-\pi, \pi]$, specifying some of the properties of the figure. You can try to comment or modify these properties…

[35]:
```
# creates a figure with size 10x5 inches
fig = plt.figure(figsize=(10, 5))

# plot cosine, blue continuous line, width=1, label=cosine
plt.plot(X, COS, color="blue", linestyle="-", linewidth=5, label='$cosine$')

# plot sine, red dotted line, width=1, label=sine
plt.plot(X, SIN, color="red", linestyle="--", linewidth=1, label='$sine$')

# specify x and y limits
plt.xlim(-np.pi*1.1, np.pi*1.1)
plt.ylim(-1.1, 1.1)

# add a label on x-axis
plt.xlabel('x', fontsize=18)

# add a title
plt.title('Cosine and Sine functions', fontsize=18)

# add a legend located at the top-left of the figure
plt.legend(loc='upper left')

# show the figure
plt.show()
```



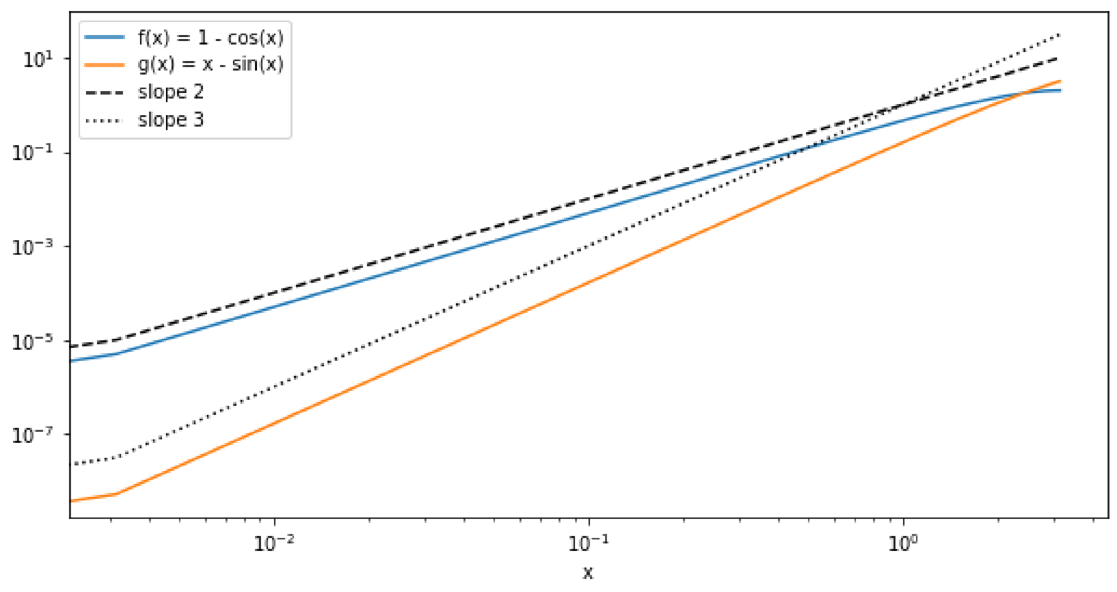
- On the same figure, plot $x \to 1 - \cos(x)$ and $x \to x - \sin(x)$ for $x \in [0, \pi]$.
- Add a legend.
- On another figure, plot the same functions using a loglog scale (use plt.loglog, to plot log(f(x)) versus log(x)).

[45]:
```
from math import pi

x = np.linspace(0, pi, 1000)  # 构建一个从 0 到 pi 的 1000 个点
f = 1 - np.cos(x)  # 构建第一个函数
g = x - np.sin(x)  # 构建第二个函数

fig = plt.figure(figsize=(10, 5))  # 创建一个图表，大小为 10x5
plt.plot(x, f, label="f(x) = 1 - cos(x)")  # 绘制第一个函数
plt.plot(x, g, label="g(x) = x - sin(x)")  # 绘制第二个函数
plt.xlabel('x')  # 设置 x 轴的标签
plt.ylabel('y')  # 设置 y 轴的标签
plt.legend()  # 显示图例
plt.figure()
plt.show()  # 显示图表
```



```
<Figure size 432x288 with 0 Axes>
```

[48]:
```
x = np.linspace(0, pi, 1000)
f = 1 - np.cos(x)
g = x - np.sin(x)

fig = plt.figure(figsize=(10, 5))
plt.loglog(x, f, label="f(x) = 1 - cos(x)")  # 运用对数绘制第一个函数
plt.loglog(x, g, label="g(x) = x - sin(x)")  # 运用对数绘制第二个函数
plt.loglog(x,x**2,'k--',label="slope 2")
plt.loglog(x,x**3,'k:',label="slope 3")
plt.xlabel('x')
plt.legend()
plt.show()
```



The curves seem almost straight in this scale. Indeed, we have $f(x) \approx x^2/2$ and $g(x) \approx x^3/6$ when x goes to zero. Therefore $\ln f(x) \approx 2\ln x - \ln 2$ and $\ln g(x) \approx 3\ln x - \ln 6$, and we expect the two curves to be close to lines having slope 2 and 3 respectively. We observe on the picture that the slopes are the expected ones.

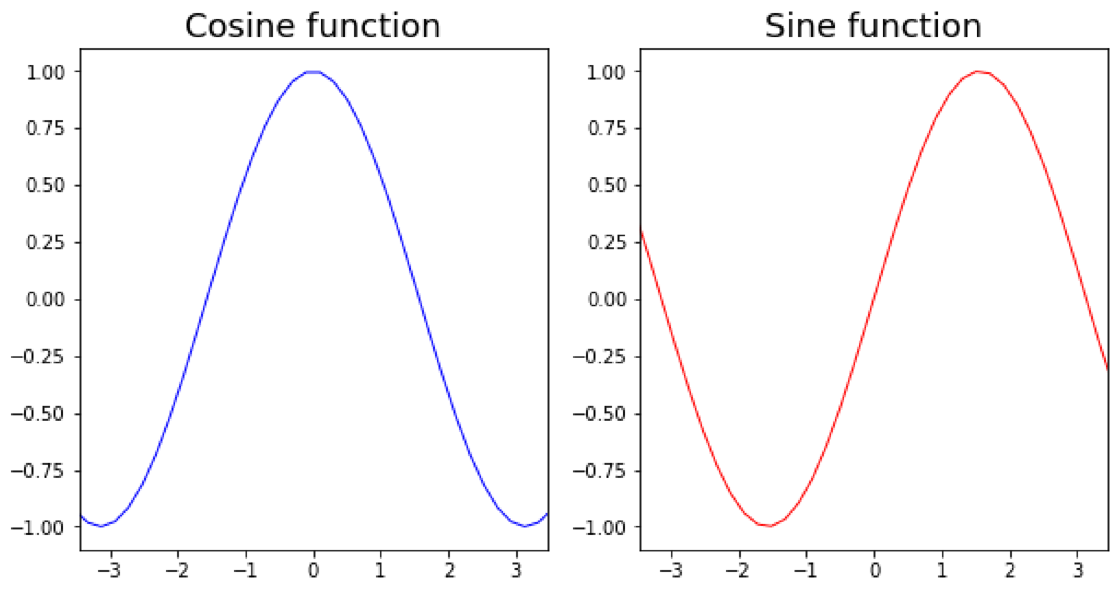
感觉不太理解 $f(x) \approx x^2/2$ 和 $g(x) \approx x^3/6$ 在这个地方时怎么得到的

1.2.3 Creating several subfigures

- What do you notice on the loglog plot? Can you explain it? Create a text-cell with style "Answer" to write your comments.

A "figure" in matplotlib means the whole window in the user interface. Within this figure there can be "subplots". In the following, we use subplots to draw sine and cosine in two different plots.

[49]:
```
# create the figure
fig = plt.figure(figsize=(10, 5))


# Create a new subplot
# input: (num-lines, num-columns, num of the current plot)
plt.subplot(1, 2, 1)  # grid 1 x 2, current plot =1
plt.plot(X, COS, color="blue", linestyle="-", linewidth=1)
plt.xlim(-np.pi*1.1, np.pi*1.1)
plt.ylim(-1.1, 1.1)
plt.title('Cosine function', fontsize=18) # add a title and the size of the␣
↪title

# Create a new subplot
plt.subplot(1, 2, 2)  # grid 1 x 2, current plot = 2
plt.plot(X, SIN, color="red", linestyle="-", linewidth=1)
plt.xlim(-np.pi*1.1, np.pi*1.1)
plt.ylim(-1.1, 1.1)
plt.title('Sine function', fontsize=18) # add a title and the size of the title

plt.show()
```

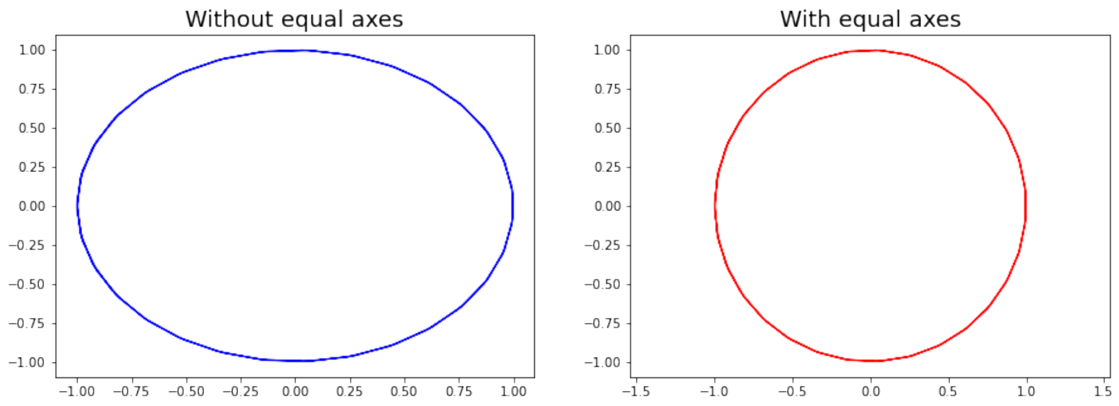


1.2.4 Using equal axes

It is also possible to impose the same scale on both axes. As an example, we plot below a circle of radius 1:

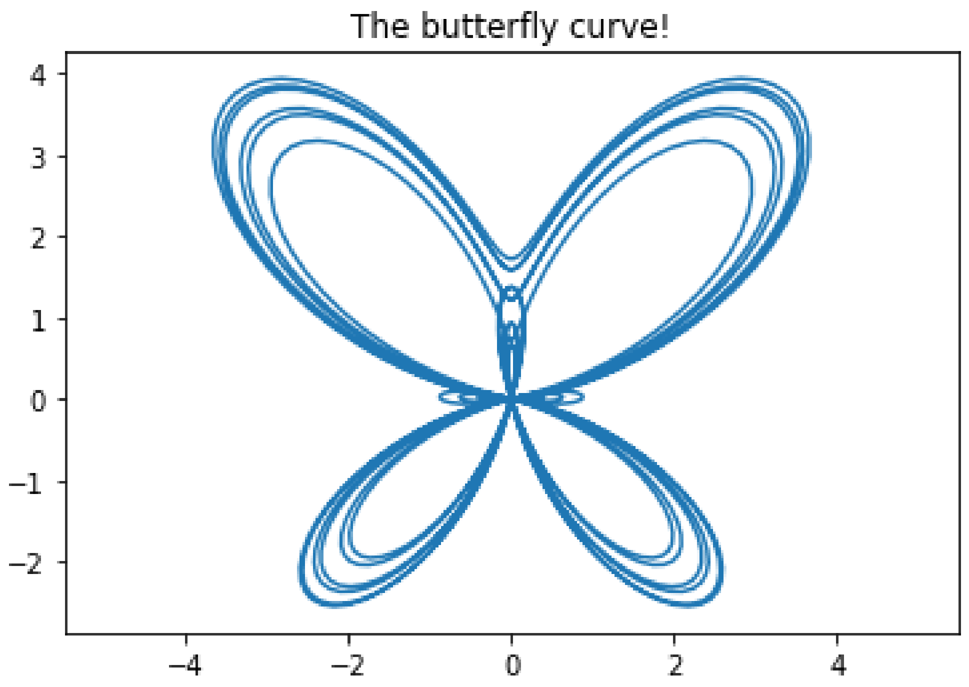
[50]:
```
# create the figure
fig = plt.figure(figsize=(15, 5))
```

```
# Create a new subplot
plt.subplot(1, 2, 1)  # grid 1 x 2, current plot =1
plt.plot(COS, SIN, color="blue", linestyle="-", linewidth=1)
plt.title('Without equal axes', fontsize=18)

# Create a new subplot
plt.subplot(1, 2, 2)  # grid 1 x 2, current plot = 2
plt.axis('equal')
plt.plot(COS, SIN, color="red", linestyle="-", linewidth=1)
plt.title('With equal axes', fontsize=18)

plt.show()
```



- Plot the butterfly curve.

[54]:
```
t = np.linspace(0, 12 * pi, 1000)
x = np.sin(t) * (np.exp(np.cos(t)) - 2 * np.cos(4 * t) + np.sin(t / 12))
y = np.cos(t) * (np.exp(np.cos(t)) - 2 * np.cos(4 * t) + np.sin(t / 12))

plt.figure()
plt.plot(x, y)  # 绘制图像
plt.axis('equal')  # 使 x 轴和 y 轴的单位长度相等
plt.title("The butterfly curve!")
plt.show()
```



蝴蝶曲线的公式:

$$x = sint[e^{cost} - 2cos(4t) + sin^5(\frac{1}{12}t)]$$
$$y = cost[e^{cost} - 2cos(4t) + sin^5(\frac{1}{12}t)]$$

For more information: https://mathworld.wolfram.com/ButterflyCurve.html

1.3 To go further...

Lots of documentation and tutorials can be found on the web about Numpy and Matplotlib. A very complete tutorial for scientific python can be found here http://www.scipy-lectures.org/index.html. In particular, you can have a look at the two following chapters:

- Numpy: http://www.scipy-lectures.org/intro/numpy/index.html
- Matplotlib: http://www.scipy-lectures.org/intro/matplotlib/index.html

[ ]:
```
# execute this part to modify the css style
from IPython.core.display import HTML
def css_styling():
    styles = open("./style/custom3.css").read()
    return HTML(styles)
css_styling()
```

[ ]:

```

```