0.1 Jupyter tutorial

February 16, 2025

1 Jupyter Notebook Tutorial

Course: *Analyse Numérique pour SV* (MATH-251(c))

Prof *Simone Deparis*

SSV, BA4, 2022

Adapted from the Jupyter tutorial by Allison Parrish and the version of Prof. Felix Naef for BIO-341

Jupyter Notebook gives you a convenient way to experiment with Python code, interspersing your experiments with notes and documentation. You can do all of this without having to muck about on the command line, and the resulting file can be easily published and shared with other people. In this course, I'll be using Jupyter Notebook to do in-class examples, and the notes will be made available as Jupyter Notebooks. Some of the homeworks will be assigned in the form of Jupyter Notebooks as well.

A Jupyter Notebook consists of a number of "cells," stacked on the page from top to bottom. Cells can have text or code in them. You can change a cell's type using the "Cell" menu at the top of the page; go to `Cell > Cell Type` and select either `Code` for Python code or `Markdown` for text. (You can also change this for the current cell using the drop-down menu in the toolbar.)

1.1 Text cells

Make a new cell, change its type to `Markdown`, type some stuff and press `Ctrl-Enter`. Jupyter Notebook will "render" the text and display it on the page in rendered format. You can hit `Enter` or click in the cell to edit its contents again. Text in `Markdown` cells is rendered according to a set of conventions called Markdown. Markdown is a simple language for marking up text with basic text formatting information (such as bold, italics, hyperlinks, tables, etc.). Here's a tutorial. You'll also be learning Markdown in more detail in the Foundations course.

1.2 Code cells

You can also press `Alt-Enter` to render the current cell and create a new cell. New cells will by default be `Code` cells. Try it now!

[1]:
```
print("This is a code cell.")
print("")
print("Any Python code you type in this cell will be run when you press the␣
 ↪'Run' button,")
print("or when you press Ctrl-Enter.")
```

```
print("")
print("If the code evaluates to something, or if it produces output, that␣
↪output will be")
print("shown beneath the cell after you run it.")
```

```
This is a code cell.

Any Python code you type in this cell will be run when you press the 'Run'
button,
or when you press Ctrl-Enter.

If the code evaluates to something, or if it produces output, that output will
be
shown beneath the cell after you run it.
```

[2]:
```
print("If your Python code generates an error, the error will be displayed in␣
↪addition")
print("to any output already produced.")

1 / 0
```

```
If your Python code generates an error, the error will be displayed in addition
to any output already produced.
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[2], line 4
      1 print("If your Python code generates an error, the error will be␣
↪displayed in addition")
      2 print("to any output already produced.")
----> 4 1 / 0

ZeroDivisionError: division by zero
```

Any variables you define or modules you import in one code cell will be available in subsequent code cells. Start with this:

[ ]:
```
import random
stuff = ["cheddar", "daguerrotype", "elephant", "flea market"]
```

… and in subsequent cells you can do this:

[ ]:
```
print(random.choice(stuff))
```

1.3 Keyboard shortcuts

As mentioned above, `Ctrl-Enter` runs the current cell; `Alt-Enter` runs the current cell and then creates a new cell. `Enter` will start editing whichever cell is currently selected. To quit editing a cell, hit `Esc`. If the cursor isn't currently active in any cell (i.e., after you've hit `Esc`), a number of other keyboard shortcuts are available to you:

- `m` converts the selected cell to a Markdown cell
- `b` inserts a new cell below the selected one
- `x` "cuts" the selected cell; `v` pastes a previously cut cell below the selected cell
- `h` brings up a help screen with many more shortcuts.

1.4 Saving your work

Hit `Cmd-S` at any time to save your notebook. Jupyter Notebook also automatically saves occasionally. Make sure to give your notebook a descriptive title by clicking on "Untitled0" at the top of the page and replacing the text accordingly. Notebooks you save will be available on your server whenever you log in again, from wherever you log into the server.

You can "download" your notebook in various formats via `File > Download as`. You can download your notebook as a static HTML file (for, e.g., uploading to a web site), or as a `.ipynb` file, which you can share with other people who have Jupyter Notebook or make available online through, e.g., nbviewer.

0.2 Python tutorial

February 16, 2025

1 Python tutorial

Course: *Analyse Numérique pour SV* (MATH-251(c)) **Professor**: *Simone Deparis*

SSV, BA4, 2022

Adapted from the `CS228` Python tutorial by by Volodymyr Kuleshov and Isaac Caswell and the version of Prof. Felix Naef for BIO-341.

1.1 Introduction

Python is a great general-purpose programming language on its own, but with the help of a few popular libraries (numpy, scipy, matplotlib) it becomes a powerful environment for scientific computing.

We don't expect that many of you will have some experience with Python and numpy; this section will serve as a quick crash course both on the Python programming language and on the use of Python for scientific computing.

Some of you may have previous knowledge in Matlab, in which case we also recommend the numpy for Matlab users page (https://docs.scipy.org/doc/numpy-1.15.0/user/numpy-for-matlab-users.html).

In this tutorial, we will cover:

- Basic Python: Basic data types (Containers, Lists, Dictionaries, Sets, Tuples), Functions
- Numpy: Arrays, Array indexing, Datatypes, Array math, Broadcasting
- Matplotlib: Plotting, Subplots, Images

1.2 Basics of Python

Python is a high-level, dynamically typed multiparadigm programming language. Python code is often said to be almost like pseudocode, since it allows you to express very powerful ideas in very few lines of code while being very readable. As an example, here is an implementation of the classic quicksort algorithm in Python:

[1]:
```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
```

```
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

1.2.1 Python versions

There are currently two different supported versions of Python, 2.7 and 3.7. Somewhat confusingly, Python 3.0 introduced many backwards-incompatible changes to the language, so code written for 2.7 may not work under 3.7 and vice versa. For this class all code will use Python 3.7.

You can check your Python version at the command line by running `python --version`.

1.2.2 Basic data types

Numbers Integers and floats work as you would expect from other languages:

[2]:
```
x = 3
print(x, type(x))
```

```
3 <class 'int'>
```

[3]:
```
print(x + 1)   # Addition;
print(x - 1)   # Subtraction;
print(x * 2)   # Multiplication;
print(x**2)  # Exponentiation;
```

```
4
2
6
9
```

[4]:
```
x += 1
print(x)  # Prints "4"
x *= 2
print(x)  # Prints "8"
```

```
4
8
```

[5]:
```
y = 2.5
print(type(y)) # Prints "<type 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'float'>
2.5 3.5 5.0 6.25
```

Note that unlike many languages, Python does not have unary increment (x++) or decrement (x–) operators.

Booleans Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

[6]:
```python
t, f = True, False
print(type(t)) # Prints "<type 'bool'>"
```

```
<class 'bool'>
```

Now we let's look at the operations:

[7]:
```python
print(t and f) # Logical AND;
print(t or f)  # Logical OR;
print(not t)   # Logical NOT;
print(t != f)  # Logical XOR;
```

```
False
True
False
True
```

Strings

[8]:
```python
hello = 'hello'    # String literals can use single quotes
world = "world"    # or double quotes; it does not matter.
print(hello, len(hello))
```

```
hello 5
```

[9]:
```python
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
```

```
hello world
```

String objects have a bunch of useful methods; for example:

[10]:
```python
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())       # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))      # Right-justify a string, padding with spaces; prints " ␣
  ↪hello"
print(s.center(7))     # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with␣
  ↪another;
                               # prints "he(ell)(ell)o"
print('  world '.strip())  # Strip leading and trailing whitespace; prints␣
  ↪"world"
```

```
Hello
HELLO
  hello
 hello
```

```
he(ell)(ell)o
world
```

1.2.3 Containers

Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Lists A list is the Python equivalent of an array, but is resizeable and can contain elements of different types:

[11]:
```python
xs = [3, 1, 2]    # Create a list
print(xs, xs[2])
print(xs[-1])     # Negative indices count from the end of the list; prints "2"
```

```
[3, 1, 2] 2
2
```

[12]:
```python
xs[2] = 'foo'    # Lists can contain elements of different types
print(xs)
```

```
[3, 1, 'foo']
```

[13]:
```python
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

```
[3, 1, 'foo', 'bar']
```

[14]:
```python
x = xs.pop()     # Remove and return the last element of the list
print(x, xs)
```

```
bar [3, 1, 'foo']
```

Slicing In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing:

[15]:
```python
nums = list(range(5))    # range is a built-in function that creates an␣
 ↪interator of integers.
                         #It has to be explicitely converted to a list to do␣
 ↪slicing
print(nums)          # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])     # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])      # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])      # Get a slice from the start to index 2 (exclusive); prints␣
 ↪"[0, 1]"
print(nums[:])      # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
print(nums[:-1])     # Slice indices can be negative; prints ["0, 1, 2, 3]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print(nums)          # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

Loops You can loop over the elements of a list like this:

[16]:
```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

```
cat
dog
monkey
```

If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

[17]:
```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print(idx + 1, animal)
```

```
1 cat
2 dog
3 monkey
```

List comprehensions: When programming, frequently we want to transform one type of data into another. As a simple example, consider the following code that computes square numbers:

[18]:
```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

```
[0, 1, 4, 9, 16]
```

You can make this code simpler using a list comprehension:

[19]:
```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

List comprehensions can also contain conditions:

[20]:
```python
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

```
[0, 4, 16]
```

Dictionaries A dictionary stores (key, value) pairs, similar to a `Map` in Java or an object in Javascript. You can use it like this:

[21]:
```python
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])       # Get an entry from a dictionary; prints "cute"
print('cat' in d)     # Check if a dictionary has a given key; prints "True"
```

```
cute
True
```

[22]:
```python
d['fish'] = 'wet'     # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
```

```
wet
```

[23]:
```python
print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[23], line 1
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

[24]:
```python
print(d.get('monkey', 'N/A'))  # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))    # Get an element with a default; prints "wet"
```

```
N/A
wet
```

[25]:
```python
del(d['fish'])        # Remove an element from a dictionary
print(d.get('fish', 'N/A')) # "fish" is no longer a key; prints "N/A"
```

```
N/A
```

It is easy to iterate over the keys in a dictionary:

[26]:
```python
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A', animal, 'has', legs, 'legs')
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

If you want access to keys and their corresponding values, use the items method:

[27]:
```python
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A', animal, 'has', legs, 'legs')
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Dictionary comprehensions: These are similar to list comprehensions, but allow you to easily construct dictionaries. For example:

[28]:
```python
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)
```

```
{0: 0, 2: 4, 4: 16}
```

Sets A set is an unordered collection of distinct elements. As a simple example, consider the following:

[29]:
```python
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"
```

```
True
False
```

[30]:
```python
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

[31]:
```python
animals.add('cat')       # Adding an element that is already in the set does␣
 ↪nothing
print(len(animals))
animals.remove('cat')    # Remove an element from a set
print(len(animals))
```

```
3
2
```

Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

[32]:
```python
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print(idx + 1,':', animal)
# Prints "1 : fish", "2 : dog", "3 : cat"
```

```
1 : dog
2 : fish
3 : cat
```

Set comprehensions: Like lists and dictionaries, we can easily construct sets using set comprehensions:

[33]:
```python
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

```
{0, 1, 2, 3, 4, 5}
```

Tuples A tuple is an (immutable) ordered list of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot. Here is a trivial example:

[34]:
```python
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)       # Create a tuple
print(type(t))
print(d[t])
print(d[(1, 2)])
```

```
<class 'tuple'>
5
1
```

[35]:
```python
t[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 1
----> 1 t[0] = 1

TypeError: 'tuple' object does not support item assignment
```

1.2.4 Functions

Python functions are defined using the `def` keyword. For example:

[36]:
```python
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
```

```
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

```
negative
zero
positive
```

We will often define functions to take optional keyword arguments, like this:

[37]:
```
def hello(name, loud=False):
    if loud:
        print('HELLO', name.upper())
    else:
        print('Hello', name)

hello('Bob')
hello('Fred', loud=True)
```

```
Hello Bob
HELLO FRED
```

1.3 Numpy

Numpy is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

To use Numpy, we first need to import the `numpy` package:

[38]:
```
import numpy as np
```

1.3.1 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

[39]:
```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

[40]:
```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b)
```

```
[[1 2 3]
 [4 5 6]]
```

[41]:
```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

[42]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

```
[[0. 0.]
 [0. 0.]]
```

[43]:
```
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

```
[[1. 1.]]
```

[44]:
```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

[45]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

[46]:
```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.40761394 0.33088024]
 [0.07710091 0.76909554]]
```

1.3.2 Array indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[47]:
```
import numpy as np
```

```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

[48]:
```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
2
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

[49]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Two ways of accessing the data in the middle row of the array. Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:

[50]:
```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
row_r3 = a[[1], :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

[51]:
```python
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing: When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

[52]:
```python
a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

```
[1 4 5]
[1 4 5]
```

[53]:
```python
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

```
[2 2]
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

[54]:
```python
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

[55]:
```python
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

```
[ 1  6  7 11]
```

[56]:
```python
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

[57]:
```python
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

[58]:
```python
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

1.3.3 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

[59]:
```
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

You can read all about numpy datatypes in the documentation.

1.3.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[60]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

[61]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

[62]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

[63]:
```python
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

[64]:
```python
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, `*` is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot or @ is available both as a function in the numpy module and as an instance method of array objects:

[65]:
```python
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(v@w)
print(np.dot(v, w))
```

```
219
219
219
```

[66]:
```python
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
```

```
[29 67]
[29 67]
```

[67]:
```python
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
```

```
print(np.dot(x, y))
```

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

[68]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

[69]:
```
print(x)
print(x.T)
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
```

[70]:
```
v = np.array([[1,2,3]])
print(v)
print(v.T)
```

```
[[1 2 3]]
[[1]
 [2]
 [3]]
```

1.3.5 Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

[71]:
```python
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

[72]:
```python
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

[73]:
```python
y = x + vv  # Add x and vv elementwise
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

[74]:
```python
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

Here are some applications of broadcasting:

[75]:
```python
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

[76]:
```python
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

[77]:
```python
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
```

```
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[78]:
```
# Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.

print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

[79]:
```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:

print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

1.3.6 Extracting (generic) matrix blocks

Extracting (generic) matrix blocks is very easily accomplished in Matlab, but in Python/Numpy, normally this functionality is hidden in tutorials.

[80]:
```
print ( "A = ")
A = np.array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5],
       [3, 4, 5, 6]])
print(A)

print ( "\nA[::2,1::2]  # easy because you have a regular structure in the␣
 ↪sequences")
print ( A[::2,1::2] )
# array([[1, 3],
#        [3, 5]])

print ( "\nA[[0,2],[1,3]]  # This extracts the diagonal and not the block ")
```

```python
print ( A[[0,2],[1,3]])
# array([1, 5])

print ( "\nA[np.ix_([0,2],[1,3])] # np.ix_ allows to extract a block ")
print ( A[np.ix_([0,2],[1,3])] )
# array([[1, 3],
#        [3, 5]])

print ( "\nA[(np.array([0,2]).reshape(2,1),np.array([1,3]).reshape(1,2))] #␣
 ↪hidden manipulation ")
print ( A[(np.array([0,2]).reshape(2,1),np.array([1,3]).reshape(1,2))] )
# array([[1, 3],
#        [3, 5]])

print ( "\nA[np.ix_([0,1],[0,3])] # something hard to accomplish without np.ix_␣
 ↪")
print ( A[np.ix_([0,1],[0,3])] )
# array([[0, 3],
#        [1, 4]])
```

```
A =
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]

A[::2,1::2]  # easy because you have a regular structure in the sequences
[[1 3]
 [3 5]]

A[[0,2],[1,3]]  # This extracts the diagonal and not the block
[1 5]

A[np.ix_([0,2],[1,3])] # np.ix_ allows to extract a block
[[1 3]
 [3 5]]

A[(np.array([0,2]).reshape(2,1),np.array([1,3]).reshape(1,2))] # hidden
manipulation
[[1 3]
 [3 5]]

A[np.ix_([0,1],[0,3])] # something hard to accomplish without np.ix_
[[0 3]
 [1 4]]
```

1.4 Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

[81]:
```
import matplotlib.pyplot as plt
```

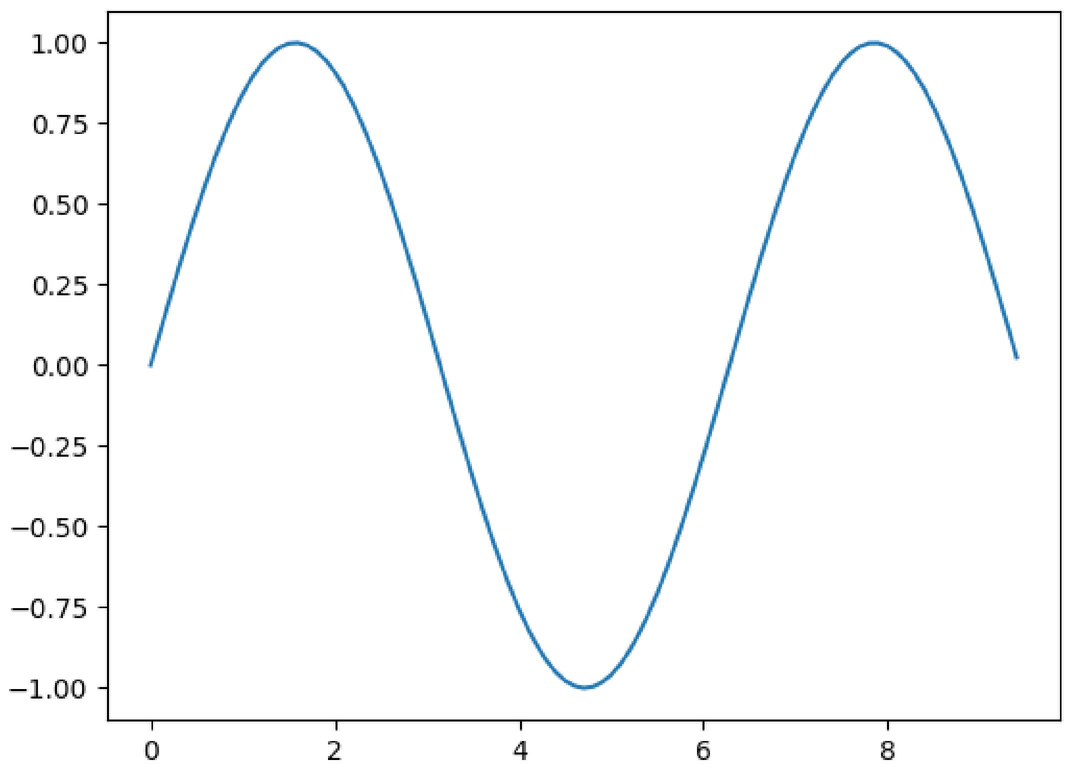
By running this special iPython command, we will be displaying plots inline:

[82]:
```
%matplotlib inline
```

1.4.1 Plotting

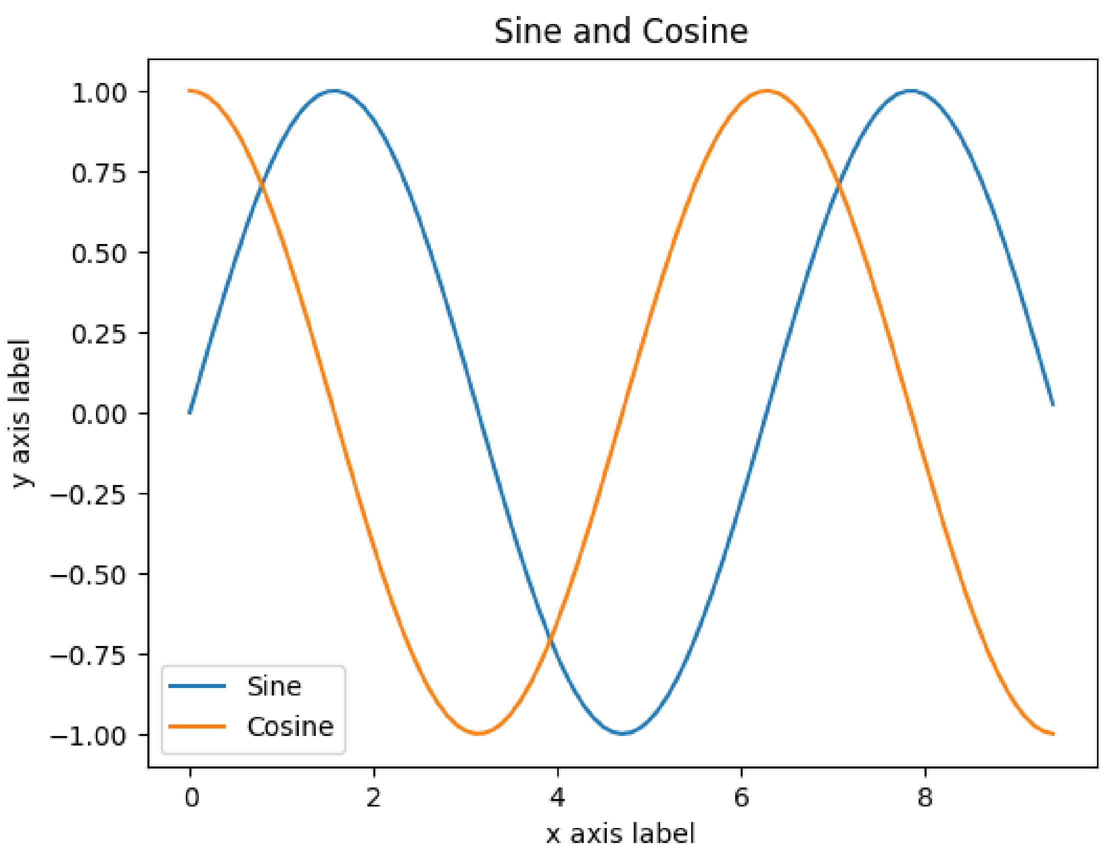
The most important function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

[83]:
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```



With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

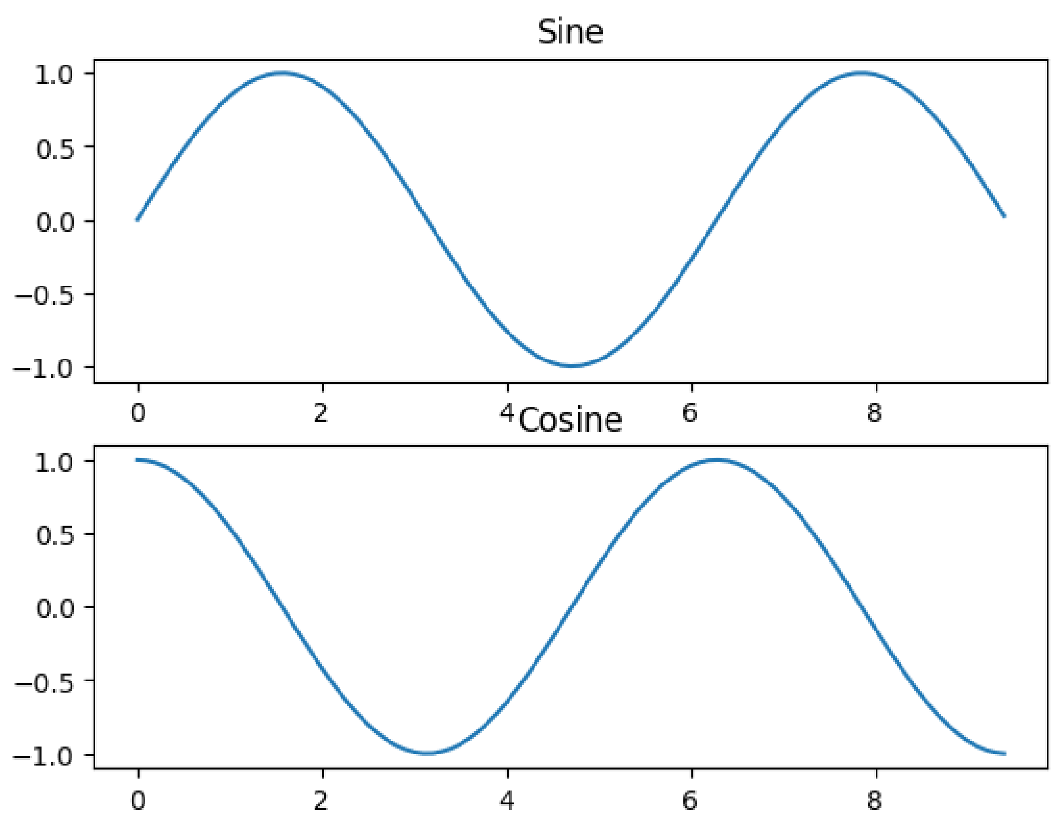
[84]:
```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



1.4.2 Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

[85]:
```python
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

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



```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

0.4 Representation des nombres

February 16, 2025

1 Représentation des nombres

[ ]:
```python
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt
```

1.1 Représentation de nombres : Exemple e

L'expression suivante du nombre d'Euler e est connue :

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n.$$

On s'attend donc à ce que $e_n = \left(1+\frac{1}{n}\right)^n$ donne des approximations de plus en plus bonnes de e. En arithmétique exacte, c'est effectivement le cas. Sur l'ordinateur, la suite *calculée* $\hat{e}_n$ se comporte tout à fait différemment :

[ ]:
```python
# Approximation of e
print(r'10^i \t\t e_n \t\t\t e_n - e')
for i in range(1,16):
    n = 10.0 ** i; en = (1 + 1/n) ** n
    print('10^%2d \t %20.15f \t %20.15f' % (i, en, en-np.e))
```

1.2 Représentation de nombres : Exemple e^x

La série de Taylor pour la fonction exponentielle converge pour tout $x \in \mathbb{R}$:

$$e^x = \sum_{k=0}^{\infty}\frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots.$$

L'ordinateur peut seulement calculer la série partielle

$$s_i(x) = \sum_{k=0}^{i}\frac{x^k}{k!}.$$

Le reste de Taylor est donné par

$$e^x - s_i(x) = \frac{e^{\xi}x^{i+1}}{(i+1)!} \text{ pour un } \xi \in \mathbb{R} \text{ avec } 0 < |\xi| < |x|$$

Si on choisi une tolerance $\tau > 0$ et i tel que $|x|^{i+1}/(i+1)! \leq \tau s_i(x)$, pour $x < 0$ on obtient

$$|e^x - s_i(x)| \leq \frac{|x|^{i+1}}{(i+1)!} \leq \tau s_i(x) \approx \tau e^x.$$

Au même temps, **l'erreur rélative** $|e^x - s_i(x)|/|e^x|$ est bornée par τ.

[ ]:
```
def expeval(x, tol):
    #Approximation of e^x
    s = 1; k = 1
    term = 1
    while (abs(term)>tol*abs(s)):
        term = term * x / k
        s += term ; k += 1
    return s
```

[ ]:
```
# reproduisez ici le tableau
# $x$ & $s_i(x)$ & $\exp(x)$ & $\frac{|\exp(x)-s_i(x)|}{\exp(x)}$
```

1.3 IEEE 754 in Python

En Python, toutes les opérations sur les nombres réels sont exécutées par défaut en double précision. Les variables en simple précision sont générées par la commande `numpy.float32()`.

[ ]:
```
import sys
sys.float_info.min              # 2.2251e-308
```

[ ]:
```
sys.float_info.max              # 1.7977e+308
```

[ ]:
```
1 / 0                           # Divide by zero error
```

[ ]:
```
3 * float('inf')                # np.inf
```

[ ]:
```
-1 / 0                          # Divide by zero error
```

[ ]:
```
0 / 0                          # Divide by zero error
```

[ ]:
```
float('inf') - float('inf')     # nan
```

De manière assez surprenante, et non en accord avec le standard IEEE 754 https://wusun.name/blog/2017-12-18-python-zerodiv/, Python renvoit une erreur lorsqu'une division par zéro se produit, au lieu de retourner un ∞ signé. Pour éviter cela, on peut utiliser `float64` de `numpy`. Par exemple, `1/np.float64(0)` renvoie `inf` (comme il se doit).

1.4 Arrondis

1.4.1 Partial sum and sums

Comme exemple, regardons ce qu'il se passe avec de `float16` et une somme de nombre de plus en plus petits.

[ ]:
```python
# see what happens with n = 2, 10, 100, 1000, 10000
n = 10000

# t and y are 64 bit float (double) numbers
t = np.sin(np.linspace(0,1,n))
y = np.linspace(0,1,n)

y[1::2] = -1 * y[1::2] # odd numbers set to negatives

# reduced accuracy
x= np.float16(y)
r= np.float16(t)
```

[ ]:
```python
# Use the sum function
print('Sum in np.float64: %1.20f'
      % sum(y*t))
print('Sum in np.float16: %1.20f'
      % sum(x*r))
print('Sum in np.float64 converted in np.float16: %1.20f'
      % np.float16(sum(y*t)))
print('Sum in np.float16 converted in np.float16: %1.20f'
      % np.float16(sum(x*r)))
```

[ ]:
```python
# Use the sum function, reverse order
print('Reverse sum in np.float64: %1.20f'
      % sum(y[::-1]*t[::-1]))
print('Reverse sum in np.float16: %1.20f'
      % sum(x[::-1]*r[::-1]))
print('Reverse sum in np.float64 converted in np.float16: %1.20f'
      % np.float16(sum(y[::-1]*t[::-1])))
print('Reverse sum in np.float16 converted in np.float16: %1.20f'
      % np.float16(sum(x[::-1]*r[::-1])))
```

[ ]:
```python
# loop for the sum from the first to the last, in float16
s1 = np.float16(0)
for (xk,rk) in zip(x,r) :
    s1 += xk * rk
print('Sum first to last: %1.20f' % s1)

# loop for the sum from the last to the first, in float 16
s2 = np.float16(0)
for (xk,rk) in zip(reversed(x), reversed(r)) :
    s2 += xk * rk
print('Sum last to first: %1.20f' % s2)
```

1.5 Cancellation, exemple

Considérons

$$f(x) = \frac{1-\cos(x)}{x^2}\ , \quad x = b^{-k},\ k = 5, ..., 9.$$

[ ]:
```
def f(x):
    return (1 - np.cos(x))/ (x**2)

# Try with b = 2,3,4,10
b = 10

k = np.array(np.arange(6,20))
x = 1.0 / b**k
print('k = \t ', k)
print('x = 10^-k = \t ', x)
print('f(x) = \t ',  f(x))

## Plot a semilog-x graph with x and f(x)
plt.semilogx(x, f(x),'b*-')

# Labels
plt.xlabel('x'); plt.ylabel('f(x)')
plt.title('f(x) with respect to x', fontweight='bold')
plt.legend(['error'])
plt.grid()
plt.show()
```

1.5.1 Exemple, différences finies

Considérons $f(x) = e^x$, $x_0 = 0$ et

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h) - f(x_0)}{h}.$$

Attente : L'approximation de $f'(x_0)$ par un quotient de différences finies tend à s'améliorer lorsque h s'approche de 0.

[ ]:
```
def f(x) :
    return np.exp(x)
def df(x) :
    return np.exp(x)
x0 = 0

# prenez h = 10^{-k} avec k=0,1,...,16
k = np.array(range(0,17))
h = 1/10**k

# calculez la difference finie (f(x0+h) - f(x0))/h pour tout les h
```

```
fprime = (f(x0+h) - f(x0)) / h

# calculez l'erreur avec la solution exacte errH = |df(x0) - ..|
errH = np.abs(fprime - df(x0))

# dessinez l'erreur en log-log
plt.loglog(h,errH, 'b-*')
plt.xlabel('h'); plt.ylabel('ErrH')
plt.title('Error in finite difference', fontweight='bold')
plt.legend(['error'])
plt.grid()
plt.show()
```

1.5.2 Exemple, le nombre e

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

Attente : plus n est grand, plus $e_n = \left(1+\frac{1}{n}\right)^n$ s'approche de e.

[ ]:
```
def limN(n) :
    return (1+1/n)**n

# prenez n = 10^{k} avec k=0,1,...,16
k = np.array(range(0,17))
n = 10**k

# calculez (1+1/n)**n
approxE = limN(n)

# calculez l'erreur avec la solution exacte errN = |e - ..|
errN = np.abs(np.e - approxE)

# dessinez l'erreur en log-log
plt.loglog(n, errN, 'b-*')
plt.xlabel('n'); plt.ylabel('ErrN')
plt.title('Error in finite difference', fontweight='bold')
plt.legend(['error'])
plt.grid()
plt.show()
```

[ ]: