Analyse Numérique

Simone Deparis

June 4, 2023

Contents

1 Jupyter Notebook Tutorial **1**
1.1 Text cells . . . 1
1.2 Code cells . . . 1
1.3 Keyboard shortcuts . . . 3
1.4 Saving your work . . . 3

2 Python tutorial **3**
2.1 Introduction . . . 3
2.2 Basics of Python . . . 4
2.2.1 Python versions . . . 4
2.2.2 Basic data types . . . 4
2.2.3 Containers . . . 6
2.2.4 Functions . . . 11
2.3 Numpy . . . 11
2.3.1 Arrays . . . 12
2.3.2 Array indexing . . . 13
2.3.3 Datatypes . . . 16
2.3.4 Array math . . . 16
2.3.5 Broadcasting . . . 19
2.3.6 Extracting (generic) matrix blocks . . . 21
2.4 Matplotlib . . . 23
2.4.1 Plotting . . . 23
2.4.2 Subplots . . . 25

3 Some exercises **26**
3.0.1 Exercise 0 – Learning Python using Python. . . . 26
3.0.2 Exercise 1 . . . 45
3.0.3 Exercise 2 . . . 45
3.0.4 Exercise 3 . . . 46
3.0.5 Exercise 4 . . . 48
3.0.6 Exercise 5 . . . 49
3.0.7 Exercise 6 . . . 52

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)
/var/folders/4q/yg0r5zvs6yv_8m88_ywfzxvh0000gn/T/ipykernel_36053/19277447.py in␣
 ↪<module>
      2 print("to any output already produced.")
      3
----> 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:

[3]:
```
import random
stuff = ["cheddar", "daguerrotype", "elephant", "flea market"]
```

. . . and in subsequent cells you can do this:

[4]:
```
print(random.choice(stuff))
```

```
cheddar
```

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.

2 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.

2.1 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

2.2 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:

[5]:
```
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]
```

2.2.1 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`.

2.2.2 Basic data types

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

[6]:
```
x = 3
print(x, type(x))
```

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

[7]:
```
print(x + 1)   # Addition;
print(x - 1)   # Subtraction;
print(x * 2)   # Multiplication;
print(x**2)  # Exponentiation;
```

```
4
2
6
9
```

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

```
4
8
```

[9]:
```
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.):

[10]:
```
t, f = True, False
print(type(t)) # Prints "<type 'bool'>"
```

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

Now we let's look at the operations:

[11]:
```
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

[12]:
```
hello = 'hello'   # String literals can use single quotes
world = "world"   # or double quotes; it does not matter.
print(hello, len(hello))
```

```
hello 5
```

[13]:
```
hw = hello + ' ' + world  # String concatenation
print(hw)  # prints "hello world"
```

```
hello world
```

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

[14]:
```
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
```

2.2.3 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:

[15]:
```
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
```

[16]:
```
xs[2] = 'foo'    # Lists can contain elements of different types
print(xs)
```

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

[17]:
```
xs.append('bar') # Add a new element to the end of the list
print(xs)
```

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

[18]:
```
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:

[19]:
```
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:

[20]:
```
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:

[21]:
```
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:

[22]:
```
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:

[23]:
```
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:

[24]:
```
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:

[25]:
```
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
```

[26]:
```
d['fish'] = 'wet'    # Set an entry in a dictionary
print(d['fish'])      # Prints "wet"
```

```
wet
```

[27]:
```
print(d['monkey'])  # KeyError: 'monkey' not a key of d
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/4q/yg0r5zvs6yv_8m88_ywfzxvh0000gn/T/ipykernel_36053/3521650589.py in
 ↪<module>
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

[28]:
```
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
```

[29]:
```
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:

[30]:
```
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:

[31]:
```
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:

[32]:
```
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:

[33]:
```
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
```

[34]:
```
animals.add('fish')      # Add an element to a set
print('fish' in animals)
print(len(animals))       # Number of elements in a set;
```

```
True
3
```

[35]:
```
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:

[36]:
```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print(idx + 1,':', animal)
# Prints "1 : fish", "2 : dog", "3 : cat"
```

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

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

[37]:
```
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:

[38]:
```
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
```

[39]:
```
t[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/4q/yg0r5zvs6yv_8m88_ywfzxvh0000gn/T/ipykernel_36053/1253691622.py in
 ↪<module>
----> 1 t[0] = 1

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

2.2.4 Functions

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

[40]:
```
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:

[41]:
```
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
```

2.3 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:

[42]:
```
import numpy as np
```

2.3.1 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:

[43]:
```
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]
```

[44]:
```
b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b)
```

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

[45]:
```
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:

[46]:
```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)
```

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

[47]:
```
b = np.ones((1,2))   # Create an array of all ones
print(b)
```

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

[48]:
```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

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

[49]:
```
d = np.eye(2)        # Create a 2x2 identity matrix
print(d)
```

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

[50]:
```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.32214401 0.947642  ]
 [0.12820167 0.82107516]]
```

2.3.2 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:

[51]:
```
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.

[52]:
```
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:

[53]:
```
# 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:

[54]:
```
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)
```

[55]:
```
# 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:

[56]:
```
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]
```

[57]:
```
# 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:

[58]:
```
# 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]]
```

[59]:
```
# 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]
```

[60]:
```
# 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:

[61]:
```
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]]
```

[62]:
```
# 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.

2.3.3 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:

[63]:
```
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.

2.3.4 Array math

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

[64]:
```
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.]]
```

[65]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

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

[66]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

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

[67]:
```
# 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       ]]
```

[68]:
```
# 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:

[69]:
```
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
```

[70]:
```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
```

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

[71]:
```
# 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`:

[72]:
```
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:

[73]:
```
print(x)
print(x.T)
```

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

[74]:
```
v = np.array([[1,2,3]])
print(v)
print(v.T)
```

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

2.3.5 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:

[75]:
```
# 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:

[76]:
```
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]]
```

[77]:
```
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:

[78]:
```
# 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:

[79]:
```
# 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]]
```

[80]:
```
# 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]]
```

[81]:
```
# 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]]
```

[82]:
```
# 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]]
```

[83]:
```
# 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.

2.3.6 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.

[84]:
```
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 ")
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]]
```

2.4 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.

[85]:
```
import matplotlib.pyplot as plt
```

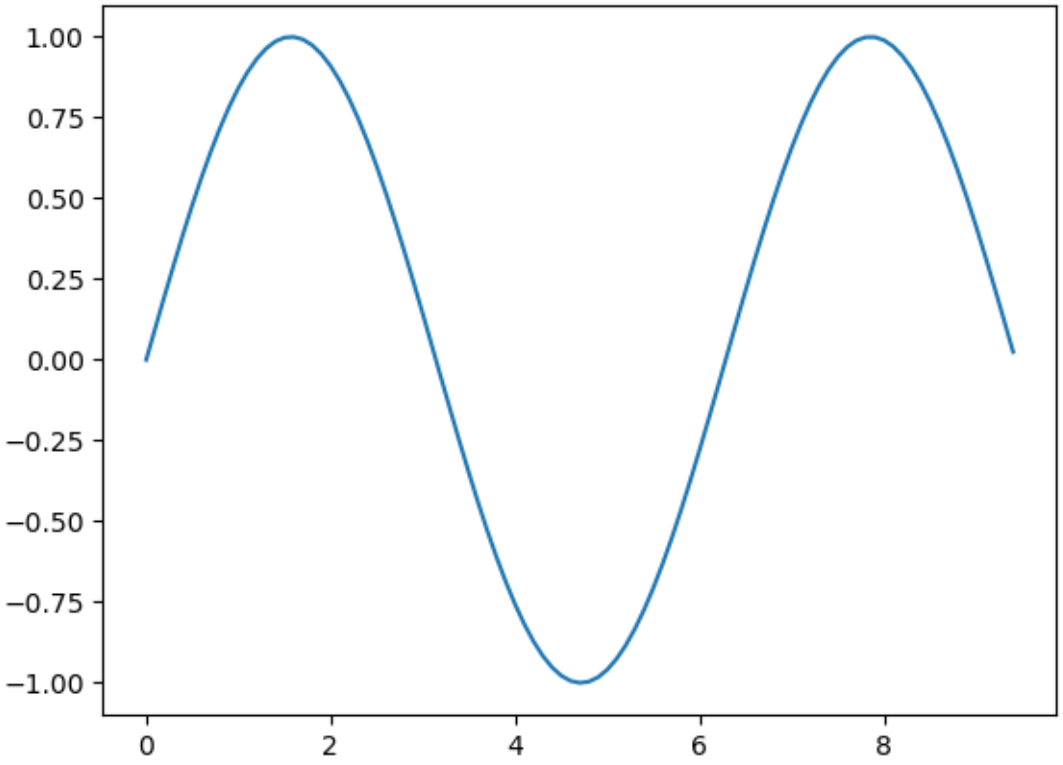
By running this special iPython command, we will be displaying plots inline:

[86]:
```
%matplotlib inline
```

2.4.1 Plotting

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

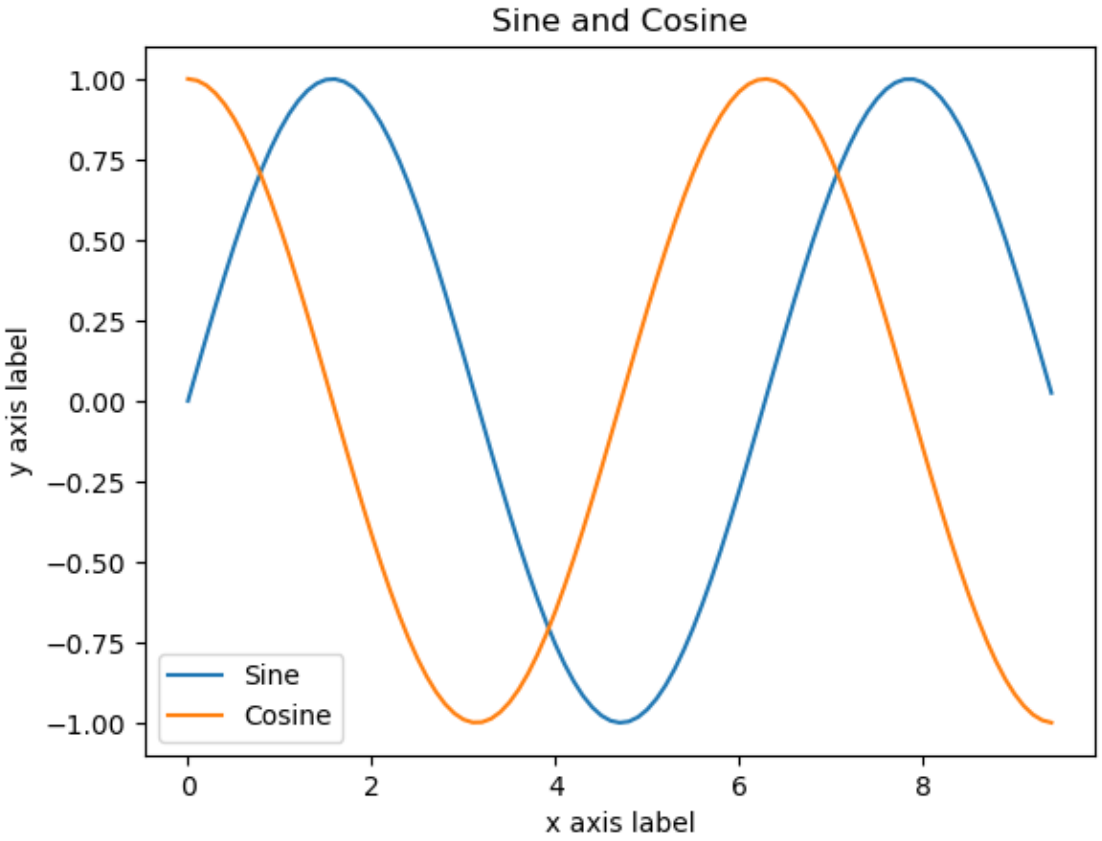
[87]:
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# 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:

[88]:
```
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()
```

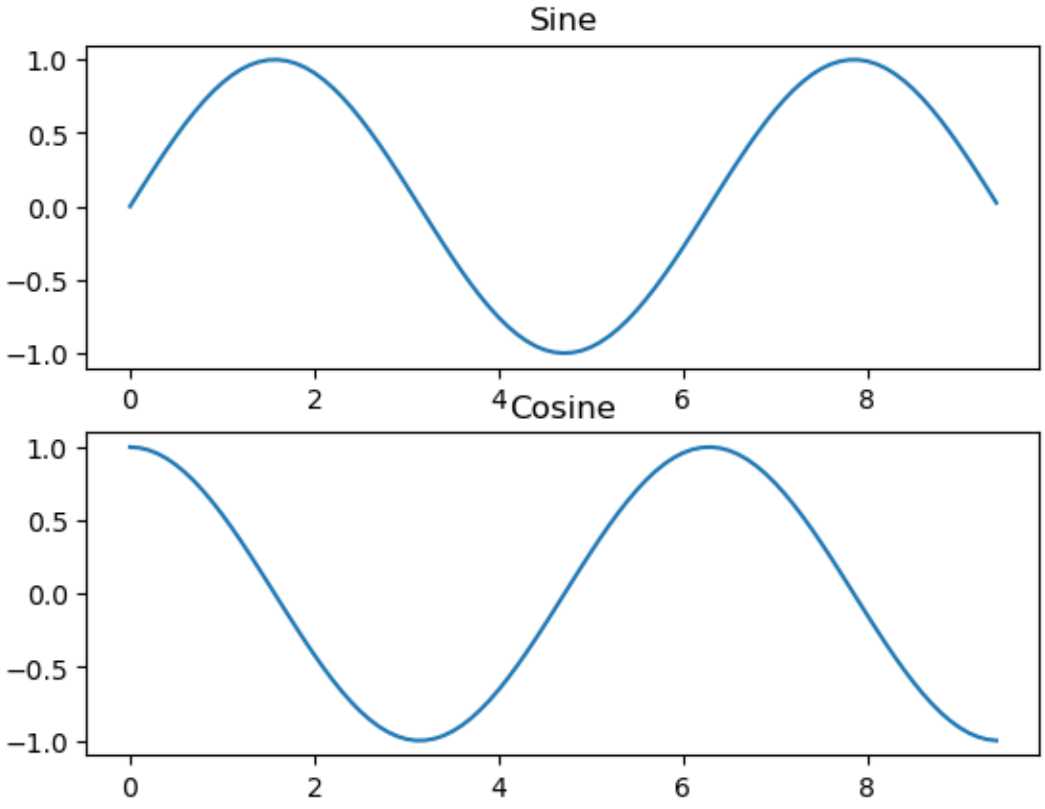


2.4.2 Subplots

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

[89]:
```
# 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()
```



3 Some exercises

Course: *Analyse Numérique pour SV* (MATH-2xx)

Original by prof. Fabio Nobile

- Before starting to work on this exercise session, you should review the material in the short `Python tutorial` available on Moodle.

3.0.1 Exercise 0 – Learning Python using Python.

Type the following commands and look at the results. Begin by importing numpy and matplotlib.pyplot as

[90]:
```
import numpy as np
import matplotlib.pyplot as plt
```

These are the libraries that python relies upon for scientific computing and plotting.

No.	Code	Instruction
1	`a=7`	Define a as a scalar variable which has type `int`
2	`a=7.1`	Define a as a scalar variable which has type `float`
3	`b=np.array([1,2,3])`	Define $b \in \mathbb{R}^{1\times 3}$. What happens if you replace `,` by `''` or `;`?
4	`g=b[2]`	Return the **third** element of b. Note: Indexation starts at 0.
5	`E=np.array([[1, 2, 3],[4, 5, 6]])`	Create matrix $E \in \mathbb{R}^{2\times 3}$. Note: Python is case sensitive.
6	`E[0,:]`	Return first row of E; $(1,4)^T$
7	`h=E[1,2]`	Return the element $E_{2,3}$ of the matrix E ($E_{2,3} = 6$).
8	`%whos`	Fetch information about the used variables.
9	`help(np.sin)`	Fetch help for the command `np.sin`. Same as `? np.sin` in iPython.
10	`A=np.eye(3)`	Return the identity matrix 3×3
11	`I=np.ones((4,4))`	Return matrix 4×4 for which each value is 1
12	`F=np.zeros((2,3))`	Return $F \in \mathbb{R}^{2\times 3}$ for which each value is 0
13	`F.T`	Perform transpose of F
14	`x=np.linspace(0,1,3)`	Return a vector of length 3 whose values are equally distributed between 0 and 1
15	`D=np.diag(b)`	Return diagonal matrix with diagonal given by vector b.
16	`np.shape(F)`	Return the number of lines and columns of F in a line vector.
17	`A@D`	Return product of two matrices
18	`E@x`	Return matrix vector product

No.	Code	Instruction
19	`A[-1,1]`	Return element of A which is on the last line, second column
20	`H=np.array([[1, 3],[9, 11]])[U+FF1B]np.linalg.solve(H,E)`	Compute $H^{-1}E$. Note: Never `np.linalg.inv(H)@E`
21	`np.linalg.det(H)`	Compute determinant of H

[91]:
```
## type here and execute the 21 commands above
#1
a=7
print('#1',a)
#2,
a=7.1
print('#2',a)
#3
b=np.array([1,2,3])
print('#3',b)
#4
g=b[2]
print('#4',g)
#5
E=np.array([[1, 2, 3],[4, 5, 6]])
print('#5',E)
#6
print('#6',E[0,:])
#7
print('#7',E[1,2])
#8
%whos
#9
print('#9')
help(np.sin)
#10
A=np.eye(3)
print('#10',A)
#11
I=np.ones(4)
print('#11',I)
#12
F=np.zeros((2,3))
print('#12',F)
#13
print('#13',F.T)
#14
x=np.linspace(0,1,3)
print('#14',x)
```

```
#15
D=np.diag(b)
print('#15',D)
#16
print('#16',np.shape(F))
#17
print('#17',A@D)
#18
g=E@x
print('#18',g)
#19
print('#19',A[-1,1])
#20
H=np.array([[1, 3],[9, 11]])
print('#20',np.linalg.solve(H,E))
#21
print('#21',np.linalg.det(H))
```

```
#1 7
#2 7.1
#3 [1 2 3]
#4 3
#5 [[1 2 3]
 [4 5 6]]
#6 [1 2 3]
#7 6
Variable             Type                          Data/Info
-------------------------------------------------------------
A                    ndarray                       4x4: 16 elems, type `int64`,
128 bytes
E                    ndarray                       2x3: 6 elems, type `int64`,
48 bytes
a                    float                         7.1
animal               str                           dog
animals              set                           {'fish', 'cat', 'dog'}
b                    ndarray                       3: 3 elems, type `int64`, 24
bytes
bool_idx             ndarray                       3x2: 6 elems, type `bool`, 6
bytes
c                    ndarray                       2x2: 4 elems, type `int64`,
32 bytes
col_r1               ndarray                       3: 3 elems, type `int64`, 24
bytes
col_r2               ndarray                       3x1: 3 elems, type `int64`,
24 bytes
d                    ndarray                       2x2: 4 elems, type `float64`,
32 bytes
e                    ndarray                       2x2: 4 elems, type `float64`,
```

```
32 bytes
even_num_to_square   dict                          n=3
even_squares         list                          n=3
f                    bool                          False
g                    int64                         3
hello                function                      <function hello at
0x7f88ec67a4d0>
hw                   str                           hello world
i                    int                           3
idx                  int                           2
legs                 int                           8
np                   module                        <module 'numpy' from
'/Us<...>kages/numpy/__init__.py'>
nums                 list                          n=5
plt                  module                        <module
'matplotlib.pyplo<...>es/matplotlib/pyplot.py'>
quicksort            function                      <function quicksort at
0x7f88ec644d40>
random               module                        <module 'random' from
'/U<...>lib/python3.7/random.py'>
row_r1               ndarray                       4: 4 elems, type `int64`, 32
bytes
row_r2               ndarray                       1x4: 4 elems, type `int64`,
32 bytes
row_r3               ndarray                       1x4: 4 elems, type `int64`,
32 bytes
s                    str                           hello
sign                 function                      <function sign at
0x7f88ec67a320>
sqrt                 builtin_function_or_method    <built-in function sqrt>
squares              list                          n=5
stuff                list                          n=4
t                    tuple                         n=2
v                    ndarray                       3: 3 elems, type `int64`, 24
bytes
vv                   ndarray                       4x3: 12 elems, type `int64`,
96 bytes
w                    ndarray                       2: 2 elems, type `int64`, 16
bytes
world                str                           world
x                    ndarray                       95: 95 elems, type `float64`,
760 bytes
xs                   list                          n=3
y                    ndarray                       95: 95 elems, type `float64`,
760 bytes
y_cos                ndarray                       95: 95 elems, type `float64`,
760 bytes
y_sin                ndarray                       95: 95 elems, type `float64`,
```

```
760 bytes
z                    ndarray                        2: 2 elems, type `int64`, 16
bytes
#9
Help on ufunc object:

sin = class ufunc(builtins.object)
 |  Functions that operate element by element on whole arrays.
 |
 |  To see the documentation for a specific ufunc, use `info`.  For
 |  example, ``np.info(np.sin)``.  Because ufuncs are written in C
 |  (for speed) and linked into Python with NumPy's ufunc facility,
 |  Python's help() function finds this page whenever help() is called
 |  on a ufunc.
 |
 |  A detailed explanation of ufuncs can be found in the docs for :ref:`ufuncs`.
 |
 |  **Calling ufuncs:** ``op(*x[, out], where=True, **kwargs)``
 |
 |  Apply `op` to the arguments `*x` elementwise, broadcasting the arguments.
 |
 |  The broadcasting rules are:
 |
 |  * Dimensions of length 1 may be prepended to either array.
 |  * Arrays may be repeated along dimensions of length 1.
 |
 |  Parameters
 |  ----------
 |  *x : array_like
 |      Input arrays.
 |  out : ndarray, None, or tuple of ndarray and None, optional
 |      Alternate array object(s) in which to put the result; if provided, it
 |      must have a shape that the inputs broadcast to. A tuple of arrays
 |      (possible only as a keyword argument) must have length equal to the
 |      number of outputs; use None for uninitialized outputs to be
 |      allocated by the ufunc.
 |  where : array_like, optional
 |      This condition is broadcast over the input. At locations where the
 |      condition is True, the `out` array will be set to the ufunc result.
 |      Elsewhere, the `out` array will retain its original value.
 |      Note that if an uninitialized `out` array is created via the default
 |      ``out=None``, locations within it where the condition is False will
 |      remain uninitialized.
 |  **kwargs
 |      For other keyword-only arguments, see the :ref:`ufunc docs
<ufuncs.kwargs>`.
 |
 |  Returns
```

```
|  -------
|  r : ndarray or tuple of ndarray
|      `r` will have the shape that the arrays in `x` broadcast to; if `out` is
|      provided, it will be returned. If not, `r` will be allocated and
|      may contain uninitialized values. If the function has more than one
|      output, then the result will be a tuple of arrays.
|
|  Methods defined here:
|
|  __call__(self, /, *args, **kwargs)
|      Call self as a function.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __str__(self, /)
|      Return str(self).
|
|  accumulate(...)
|      accumulate(array, axis=0, dtype=None, out=None)
|
|      Accumulate the result of applying the operator to all elements.
|
|      For a one-dimensional array, accumulate produces results equivalent to::
|
|        r = np.empty(len(A))
|        t = op.identity        # op = the ufunc being applied to A's  elements
|        for i in range(len(A)):
|            t = op(t, A[i])
|            r[i] = t
|        return r
|
|      For example, add.accumulate() is equivalent to np.cumsum().
|
|      For a multi-dimensional array, accumulate is applied along only one
|      axis (axis zero by default; see Examples below) so repeated use is
|      necessary if one wants to accumulate over multiple axes.
|
|      Parameters
|      ----------
|      array : array_like
|          The array to act on.
|      axis : int, optional
|          The axis along which to apply the accumulation; default is zero.
|      dtype : data-type code, optional
|          The data-type used to represent the intermediate results. Defaults
|          to the data-type of the output array if such is provided, or the
|          the data-type of the input array if no output array is provided.
```

```
|      out : ndarray, None, or tuple of ndarray and None, optional
|          A location into which the result is stored. If not provided or None,
|          a freshly-allocated array is returned. For consistency with
|          ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
|          1-element tuple.
|
|          .. versionchanged:: 1.13.0
|             Tuples are allowed for keyword argument.
|
|      Returns
|      -------
|      r : ndarray
|          The accumulated values. If `out` was supplied, `r` is a reference to
|          `out`.
|
|      Examples
|      --------
|      1-D array examples:
|
|      >>> np.add.accumulate([2, 3, 5])
|      array([ 2,  5, 10])
|      >>> np.multiply.accumulate([2, 3, 5])
|      array([ 2,  6, 30])
|
|      2-D array examples:
|
|      >>> I = np.eye(2)
|      >>> I
|      array([[1.,  0.],
|             [0.,  1.]])
|
|      Accumulate along axis 0 (rows), down columns:
|
|      >>> np.add.accumulate(I, 0)
|      array([[1.,  0.],
|             [1.,  1.]])
|      >>> np.add.accumulate(I) # no axis specified = axis zero
|      array([[1.,  0.],
|             [1.,  1.]])
|
|      Accumulate along axis 1 (columns), through rows:
|
|      >>> np.add.accumulate(I, 1)
|      array([[1.,  1.],
|             [0.,  1.]])
|
|  at(...)
|      at(a, indices, b=None, /)
```

```
 |
 |      Performs unbuffered in place operation on operand 'a' for elements
 |      specified by 'indices'. For addition ufunc, this method is equivalent to
 |      ``a[indices] += b``, except that results are accumulated for elements
that
 |      are indexed more than once. For example, ``a[[0,0]] += 1`` will only
 |      increment the first element once because of buffering, whereas
 |      ``add.at(a, [0,0], 1)`` will increment the first element twice.
 |
 |      .. versionadded:: 1.8.0
 |
 |      Parameters
 |      ----------
 |      a : array_like
 |          The array to perform in place operation on.
 |      indices : array_like or tuple
 |          Array like index object or slice object for indexing into first
 |          operand. If first operand has multiple dimensions, indices can be a
 |          tuple of array like index objects or slice objects.
 |      b : array_like
 |          Second operand for ufuncs requiring two operands. Operand must be
 |          broadcastable over first operand after indexing or slicing.
 |
 |      Examples
 |      --------
 |      Set items 0 and 1 to their negative values:
 |
 |      >>> a = np.array([1, 2, 3, 4])
 |      >>> np.negative.at(a, [0, 1])
 |      >>> a
 |      array([-1, -2,  3,  4])
 |
 |      Increment items 0 and 1, and increment item 2 twice:
 |
 |      >>> a = np.array([1, 2, 3, 4])
 |      >>> np.add.at(a, [0, 1, 2, 2], 1)
 |      >>> a
 |      array([2, 3, 5, 4])
 |
 |      Add items 0 and 1 in first array to second array,
 |      and store results in first array:
 |
 |      >>> a = np.array([1, 2, 3, 4])
 |      >>> b = np.array([1, 2])
 |      >>> np.add.at(a, [0, 1], b)
 |      >>> a
 |      array([2, 4, 3, 4])
 |
```

```
|  outer(...)
|      outer(A, B, /, **kwargs)
|
|      Apply the ufunc `op` to all pairs (a, b) with a in `A` and b in `B`.
|
|      Let ``M = A.ndim``, ``N = B.ndim``. Then the result, `C`, of
|      ``op.outer(A, B)`` is an array of dimension M + N such that:
|
|      .. math:: C[i_0, ..., i_{M-1}, j_0, ..., j_{N-1}] =
|         op(A[i_0, ..., i_{M-1}], B[j_0, ..., j_{N-1}])
|
|      For `A` and `B` one-dimensional, this is equivalent to::
|
|        r = empty(len(A),len(B))
|        for i in range(len(A)):
|            for j in range(len(B)):
|                r[i,j] = op(A[i], B[j])  # op = ufunc in question
|
|      Parameters
|      ----------
|      A : array_like
|          First array
|      B : array_like
|          Second array
|      kwargs : any
|          Arguments to pass on to the ufunc. Typically `dtype` or `out`.
|          See `ufunc` for a comprehensive overview of all available arguments.
|
|      Returns
|      -------
|      r : ndarray
|          Output array
|
|      See Also
|      --------
|      numpy.outer : A less powerful version of ``np.multiply.outer``
|                    that `ravel`\ s all inputs to 1D. This exists
|                    primarily for compatibility with old code.
|
|      tensordot : ``np.tensordot(a, b, axes=((), ()))`` and
|                  ``np.multiply.outer(a, b)`` behave same for all
|                  dimensions of a and b.
|
|      Examples
|      --------
|      >>> np.multiply.outer([1, 2, 3], [4, 5, 6])
|      array([[ 4,  5,  6],
|             [ 8, 10, 12],
```

```
 |              [12, 15, 18]])
 |
 |      A multi-dimensional example:
 |
 |      >>> A = np.array([[1, 2, 3], [4, 5, 6]])
 |      >>> A.shape
 |      (2, 3)
 |      >>> B = np.array([[1, 2, 3, 4]])
 |      >>> B.shape
 |      (1, 4)
 |      >>> C = np.multiply.outer(A, B)
 |      >>> C.shape; C
 |      (2, 3, 1, 4)
 |      array([[[[ 1,  2,  3,  4]],
 |              [[ 2,  4,  6,  8]],
 |              [[ 3,  6,  9, 12]]],
 |             [[[ 4,  8, 12, 16]],
 |              [[ 5, 10, 15, 20]],
 |              [[ 6, 12, 18, 24]]]])
 |
 |  reduce(...)
 |      reduce(array, axis=0, dtype=None, out=None, keepdims=False, initial=<no
value>, where=True)
 |
 |      Reduces `array`'s dimension by one, by applying ufunc along one axis.
 |
 |      Let :math:`array.shape = (N_0, ..., N_i, ..., N_{M-1})`.  Then
 |      :math:`ufunc.reduce(array, axis=i)[k_0, ..,k_{i-1}, k_{i+1}, ..,
k_{M-1}]` =
 |      the result of iterating `j` over :math:`range(N_i)`, cumulatively
applying
 |      ufunc to each :math:`array[k_0, ..,k_{i-1}, j, k_{i+1}, .., k_{M-1}]`.
 |      For a one-dimensional array, reduce produces results equivalent to:
 |      ::
 |
 |       r = op.identity # op = ufunc
 |       for i in range(len(A)):
 |         r = op(r, A[i])
 |       return r
 |
 |      For example, add.reduce() is equivalent to sum().
 |
 |      Parameters
 |      ----------
 |      array : array_like
 |          The array to act on.
 |      axis : None or int or tuple of ints, optional
 |          Axis or axes along which a reduction is performed.
```

```
|          The default (`axis` = 0) is perform a reduction over the first
|          dimension of the input array. `axis` may be negative, in
|          which case it counts from the last to the first axis.
|
|          .. versionadded:: 1.7.0
|
|          If this is None, a reduction is performed over all the axes.
|          If this is a tuple of ints, a reduction is performed on multiple
|          axes, instead of a single axis or all the axes as before.
|
|          For operations which are either not commutative or not associative,
|          doing a reduction over multiple axes is not well-defined. The
|          ufuncs do not currently raise an exception in this case, but will
|          likely do so in the future.
|      dtype : data-type code, optional
|          The type used to represent the intermediate results. Defaults
|          to the data-type of the output array if this is provided, or
|          the data-type of the input array if no output array is provided.
|      out : ndarray, None, or tuple of ndarray and None, optional
|          A location into which the result is stored. If not provided or None,
|          a freshly-allocated array is returned. For consistency with
|          ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
|          1-element tuple.
|
|          .. versionchanged:: 1.13.0
|             Tuples are allowed for keyword argument.
|      keepdims : bool, optional
|          If this is set to True, the axes which are reduced are left
|          in the result as dimensions with size one. With this option,
|          the result will broadcast correctly against the original `array`.
|
|          .. versionadded:: 1.7.0
|      initial : scalar, optional
|          The value with which to start the reduction.
|          If the ufunc has no identity or the dtype is object, this defaults
|          to None - otherwise it defaults to ufunc.identity.
|          If ``None`` is given, the first element of the reduction is used,
|          and an error is thrown if the reduction is empty.
|
|          .. versionadded:: 1.15.0
|
|      where : array_like of bool, optional
|          A boolean array which is broadcasted to match the dimensions
|          of `array`, and selects elements to include in the reduction. Note
|          that for ufuncs like ``minimum`` that do not have an identity
|          defined, one has to pass in also ``initial``.
|
|          .. versionadded:: 1.17.0
```

```
 |
 |      Returns
 |      -------
 |      r : ndarray
 |          The reduced array. If `out` was supplied, `r` is a reference to it.
 |
 |      Examples
 |      --------
 |      >>> np.multiply.reduce([2,3,5])
 |      30
 |
 |      A multi-dimensional array example:
 |
 |      >>> X = np.arange(8).reshape((2,2,2))
 |      >>> X
 |      array([[[0, 1],
 |              [2, 3]],
 |             [[4, 5],
 |              [6, 7]]])
 |      >>> np.add.reduce(X, 0)
 |      array([[ 4,  6],
 |             [ 8, 10]])
 |      >>> np.add.reduce(X) # confirm: default axis value is 0
 |      array([[ 4,  6],
 |             [ 8, 10]])
 |      >>> np.add.reduce(X, 1)
 |      array([[ 2,  4],
 |             [10, 12]])
 |      >>> np.add.reduce(X, 2)
 |      array([[ 1,  5],
 |             [ 9, 13]])
 |
 |      You can use the ``initial`` keyword argument to initialize the reduction
 |      with a different value, and ``where`` to select specific elements to
include:
 |
 |      >>> np.add.reduce([10], initial=5)
 |      15
 |      >>> np.add.reduce(np.ones((2, 2, 2)), axis=(0, 2), initial=10)
 |      array([14., 14.])
 |      >>> a = np.array([10., np.nan, 10])
 |      >>> np.add.reduce(a, where=~np.isnan(a))
 |      20.0
 |
 |      Allows reductions of empty arrays where they would normally fail, i.e.
 |      for ufuncs without an identity.
 |
 |      >>> np.minimum.reduce([], initial=np.inf)
```

```
 |      inf
 |      >>> np.minimum.reduce([[1., 2.], [3., 4.]], initial=10., where=[True,
False])
 |      array([ 1., 10.])
 |      >>> np.minimum.reduce([])
 |      Traceback (most recent call last):
 |          ...
 |      ValueError: zero-size array to reduction operation minimum which has no
identity
 |
 |  reduceat(...)
 |      reduceat(array, indices, axis=0, dtype=None, out=None)
 |
 |      Performs a (local) reduce with specified slices over a single axis.
 |
 |      For i in ``range(len(indices))``, `reduceat` computes
 |      ``ufunc.reduce(array[indices[i]:indices[i+1]])``, which becomes the i-th
 |      generalized "row" parallel to `axis` in the final result (i.e., in a
 |      2-D array, for example, if `axis = 0`, it becomes the i-th row, but if
 |      `axis = 1`, it becomes the i-th column).  There are three exceptions to
this:
 |
 |      * when ``i = len(indices) - 1`` (so for the last index),
 |        ``indices[i+1] = array.shape[axis]``.
 |      * if ``indices[i] >= indices[i + 1]``, the i-th generalized "row" is
 |        simply ``array[indices[i]]``.
 |      * if ``indices[i] >= len(array)`` or ``indices[i] < 0``, an error is
raised.
 |
 |      The shape of the output depends on the size of `indices`, and may be
 |      larger than `array` (this happens if ``len(indices) >
array.shape[axis]``).
 |
 |      Parameters
 |      ----------
 |      array : array_like
 |          The array to act on.
 |      indices : array_like
 |          Paired indices, comma separated (not colon), specifying slices to
 |          reduce.
 |      axis : int, optional
 |          The axis along which to apply the reduceat.
 |      dtype : data-type code, optional
 |          The type used to represent the intermediate results. Defaults
 |          to the data type of the output array if this is provided, or
 |          the data type of the input array if no output array is provided.
 |      out : ndarray, None, or tuple of ndarray and None, optional
 |          A location into which the result is stored. If not provided or None,
```

```
|          a freshly-allocated array is returned. For consistency with
|          ``ufunc.__call__``, if given as a keyword, this may be wrapped in a
|          1-element tuple.
|
|          .. versionchanged:: 1.13.0
|             Tuples are allowed for keyword argument.
|
|      Returns
|      -------
|      r : ndarray
|          The reduced values. If `out` was supplied, `r` is a reference to
|          `out`.
|
|      Notes
|      -----
|      A descriptive example:
|
|      If `array` is 1-D, the function `ufunc.accumulate(array)` is the same as
|      ``ufunc.reduceat(array, indices)[::2]`` where `indices` is
|      ``range(len(array) - 1)`` with a zero placed
|      in every other element:
|      ``indices = zeros(2 * len(array) - 1)``,
|      ``indices[1::2] = range(1, len(array))``.
|
|      Don't be fooled by this attribute's name: `reduceat(array)` is not
|      necessarily smaller than `array`.
|
|      Examples
|      --------
|      To take the running sum of four successive values:
|
|      >>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
|      array([ 6, 10, 14, 18])
|
|      A 2-D example:
|
|      >>> x = np.linspace(0, 15, 16).reshape(4,4)
|      >>> x
|      array([[ 0.,   1.,   2.,   3.],
|             [ 4.,   5.,   6.,   7.],
|             [ 8.,   9.,  10.,  11.],
|             [12.,  13.,  14.,  15.]])
|
|      ::
|
|       # reduce such that the result has the following five rows:
|       # [row1 + row2 + row3]
|       # [row4]
```

```
 |       # [row2]
 |       # [row3]
 |       # [row1 + row2 + row3 + row4]
 |
 |      >>> np.add.reduceat(x, [0, 3, 1, 2, 0])
 |      array([[12.,  15.,  18.,  21.],
 |             [12.,  13.,  14.,  15.],
 |             [ 4.,   5.,   6.,   7.],
 |             [ 8.,   9.,  10.,  11.],
 |             [24.,  28.,  32.,  36.]])
 |
 |      ::
 |
 |       # reduce such that result has the following two columns:
 |       # [col1 * col2 * col3, col4]
 |
 |      >>> np.multiply.reduceat(x, [0, 3], 1)
 |      array([[   0.,     3.],
 |             [ 120.,     7.],
 |             [ 720.,    11.],
 |             [2184.,    15.]])
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  identity
 |      The identity value.
 |
 |      Data attribute containing the identity element for the ufunc, if it has
one.
 |      If it does not, the attribute value is None.
 |
 |      Examples
 |      --------
 |      >>> np.add.identity
 |      0
 |      >>> np.multiply.identity
 |      1
 |      >>> np.power.identity
 |      1
 |      >>> print(np.exp.identity)
 |      None
 |
 |  nargs
 |      The number of arguments.
 |
 |      Data attribute containing the number of arguments the ufunc takes,
including
```

```
 |      optional ones.
 |
 |      Notes
 |      -----
 |      Typically this value will be one more than what you might expect because
all
 |      ufuncs take  the optional "out" argument.
 |
 |      Examples
 |      --------
 |      >>> np.add.nargs
 |      3
 |      >>> np.multiply.nargs
 |      3
 |      >>> np.power.nargs
 |      3
 |      >>> np.exp.nargs
 |      2
 |
 |  nin
 |      The number of inputs.
 |
 |      Data attribute containing the number of arguments the ufunc treats as
input.
 |
 |      Examples
 |      --------
 |      >>> np.add.nin
 |      2
 |      >>> np.multiply.nin
 |      2
 |      >>> np.power.nin
 |      2
 |      >>> np.exp.nin
 |      1
 |
 |  nout
 |      The number of outputs.
 |
 |      Data attribute containing the number of arguments the ufunc treats as
output.
 |
 |      Notes
 |      -----
 |      Since all ufuncs can take output arguments, this will always be (at
least) 1.
 |
 |      Examples
```

```
 |      --------
 |      >>> np.add.nout
 |      1
 |      >>> np.multiply.nout
 |      1
 |      >>> np.power.nout
 |      1
 |      >>> np.exp.nout
 |      1
 |
 |  ntypes
 |      The number of types.
 |
 |      The number of numerical NumPy types - of which there are 18 total - on
which
 |      the ufunc can operate.
 |
 |      See Also
 |      --------
 |      numpy.ufunc.types
 |
 |      Examples
 |      --------
 |      >>> np.add.ntypes
 |      18
 |      >>> np.multiply.ntypes
 |      18
 |      >>> np.power.ntypes
 |      17
 |      >>> np.exp.ntypes
 |      7
 |      >>> np.remainder.ntypes
 |      14
 |
 |  signature
 |      Definition of the core elements a generalized ufunc operates on.
 |
 |      The signature determines how the dimensions of each input/output array
 |      are split into core and loop dimensions:
 |
 |      1. Each dimension in the signature is matched to a dimension of the
 |         corresponding passed-in array, starting from the end of the shape
tuple.
 |      2. Core dimensions assigned to the same label in the signature must have
 |         exactly matching sizes, no broadcasting is performed.
 |      3. The core dimensions are removed from all inputs and the remaining
 |         dimensions are broadcast together, defining the loop dimensions.
 |
```

```
 |      Notes
 |      -----
 |      Generalized ufuncs are used internally in many linalg functions, and in
 |      the testing suite; the examples below are taken from these.
 |      For ufuncs that operate on scalars, the signature is None, which is
 |      equivalent to '()' for every argument.
 |
 |      Examples
 |      --------
 |      >>> np.core.umath_tests.matrix_multiply.signature
 |      '(m,n),(n,p)->(m,p)'
 |      >>> np.linalg._umath_linalg.det.signature
 |      '(m,m)->()'
 |      >>> np.add.signature is None
 |      True  # equivalent to '(),()->()'
 |
 |  types
 |      Returns a list with types grouped input->output.
 |
 |      Data attribute listing the data-type "Domain-Range" groupings the ufunc
can
 |      deliver. The data-types are given using the character codes.
 |
 |      See Also
 |      --------
 |      numpy.ufunc.ntypes
 |
 |      Examples
 |      --------
 |      >>> np.add.types
 |      ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 |      'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
 |      'GG->G', 'OO->O']
 |
 |      >>> np.multiply.types
 |      ['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
 |      'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
 |      'GG->G', 'OO->O']
 |
 |      >>> np.power.types
 |      ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
 |      'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
 |      'OO->O']
 |
 |      >>> np.exp.types
 |      ['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']
 |
 |      >>> np.remainder.types
```

```
 |      ['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
 |      'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']

#10 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
#11 [1. 1. 1. 1.]
#12 [[0. 0. 0.]
 [0. 0. 0.]]
#13 [[0. 0.]
 [0. 0.]
 [0. 0.]]
#14 [0.  0.5 1. ]
#15 [[1 0 0]
 [0 2 0]
 [0 0 3]]
#16 (2, 3)
#17 [[1. 0. 0.]
 [0. 2. 0.]
 [0. 0. 3.]]
#18 [4.  8.5]
#19 0.0
#20 [[ 0.0625 -0.4375 -0.9375]
 [ 0.3125  0.8125  1.3125]]
#21 -16.000000000000007
```

3.0.2 Exercise 1

Fibonacci's series is defined as follows:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}, \quad \forall n \geq 2$$

* Complete the `Python` function below which computes the 30^{th} Fibonacci's number.

- Write a function which takes as an argument a natural number n and returns the n^{th} Fibonacci's number.

[92]:
```
def Fibonacci(n):
    F0=1;F1=1
    for n in range(2,n+1):
        Fn=F1+F0;F0=F1;F1=Fn  #you can alternatively do it recurvely, but it's a␣
 ↪bad idea
    return Fn

F30=Fibonacci(30)
print(F30)
```

```
1346269
```

3.0.3 Exercise 2

Let us consider the matrices

$$A = \begin{bmatrix} 5 & 3 & 0 \\ 1 & 1 & -4 \\ 3 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 3 & 2 \\ 0 & 1 & 0 \\ 5 & 0 & 1/2 \end{bmatrix}.$$

Compute (without using any loops), the matrix $C = AB$ (matrix product) and the matrix D which has as elements $D_{ij} = A_{ij}B_{ij}$ (element-wise product).

Solution: We can use the Python commands `@` for matrix multiplication and `*` for element-wise multiplication, as shown below:

[93]:
```
A=np.array([ [5,3,0],
             [1,1,-4],
            [3,0,0]])
B=np.array([ [4,3,2],
             [0,1,0],
            [5,0,1/2]])
C=A@B
D=A*B

print('C matrix')
print(C)

print('D matrix')
print(D)
```

```
C matrix
[[ 20.  18.  10.]
 [-16.   4.   0.]
 [ 12.   9.   6.]]
D matrix
[[20.  9.  0.]
 [ 0.  1. -0.]
 [15.  0.  0.]]
```

3.0.4 Exercise 3

Define (without using any loops) the bidiagonal matrix of size $n = 5$ whose main diagonal is a vector of equally distributed points between 3 and 6, i.e.

$$D = (3, 3.75, 4.5, 5.25, 6)^T,$$

and the sub-diagonal is the vector of equally distributed points between 2 and 3.5, i.e.

$$S = (2, 2.5, 3, 3.5)^T.$$

Tip: See `https://numpy.org/doc/stable/reference/generated/numpy.diag.html`

Solution Following the reference for the`diag` method of numpy we obtain:

[94]:
```
D=np.linspace(3,6,5)
S=np.linspace(2,3.5,4)
M=np.diag(D)+np.diag(S,k=-1) #Here k=-1 indicates that we are one entry below␣
 ↪the main diagonal
print('The matrix M is')
print(M)
```

```
The matrix M is
[[3.   0.   0.   0.   0.  ]
 [2.   3.75 0.   0.   0.  ]
 [0.   2.5  4.5  0.   0.  ]
 [0.   0.   3.   5.25 0.  ]
 [0.   0.   0.   3.5  6.  ]]
```

Remember that you can also obtain help from a python function using the command `help(NAME OF FUNCTION)`. In this case: `help(np.diag)`

[95]:
```
help(np.diag)
```

```
Help on function diag in module numpy:

diag(v, k=0)
    Extract a diagonal or construct a diagonal array.

    See the more detailed documentation for ``numpy.diagonal`` if you use this
    function to extract a diagonal and wish to write to the resulting array;
    whether it returns a copy or a view depends on what version of numpy you
    are using.

    Parameters
    ----------
    v : array_like
        If `v` is a 2-D array, return a copy of its `k`-th diagonal.
        If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th
        diagonal.
    k : int, optional
        Diagonal in question. The default is 0. Use `k>0` for diagonals
        above the main diagonal, and `k<0` for diagonals below the main
        diagonal.

    Returns
    -------
    out : ndarray
        The extracted diagonal or constructed diagonal array.

    See Also
    --------
```

```
diagonal : Return specified diagonals.
diagflat : Create a 2-D array with the flattened input as a diagonal.
trace : Sum along diagonals.
triu : Upper triangle of an array.
tril : Lower triangle of an array.

Examples
--------
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(x, k=1)
array([1, 5])
>>> np.diag(x, k=-1)
array([3, 7])

>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

3.0.5 Exercise 4

Let us consider the vectors $x = (1, 4, 7, 2, 1, 2)$ and $y = (0, 9, 1, 4, 3, 0)$. Compute (without using any loops for points a and b): 1. the product, component by component, between two vectors x and y (tip: use the operator `*`) 2. the scalar product between the same vectors x and y (tip: use the operator `@`) 3. a vector whose elements are defined by: $v_1 = x_1\, y_n, \quad v_2 = x_2\, y_{n-1}, \quad \ldots, \quad v_{n-1} = x_{n-1}\, y_2, \quad v_n = x_n\, y_1$.

Solution

[96]:
```
x=np.array([1,4,7,2,1,2])
y=np.array([0,9,1,4,3,0])

v1=x*y
v2=x@y
v3=x*np.flipud(y)

print('v1 = '+str(v1))
print('v2 = '+str(v2))
print('v3 = '+str(v3))
```

```
v1 = [ 0 36  7  8  3  0]
v2 = 54
v3 = [ 0 12 28  2  9  0]
```

3.0.6 Exercise 5

Plot the functions

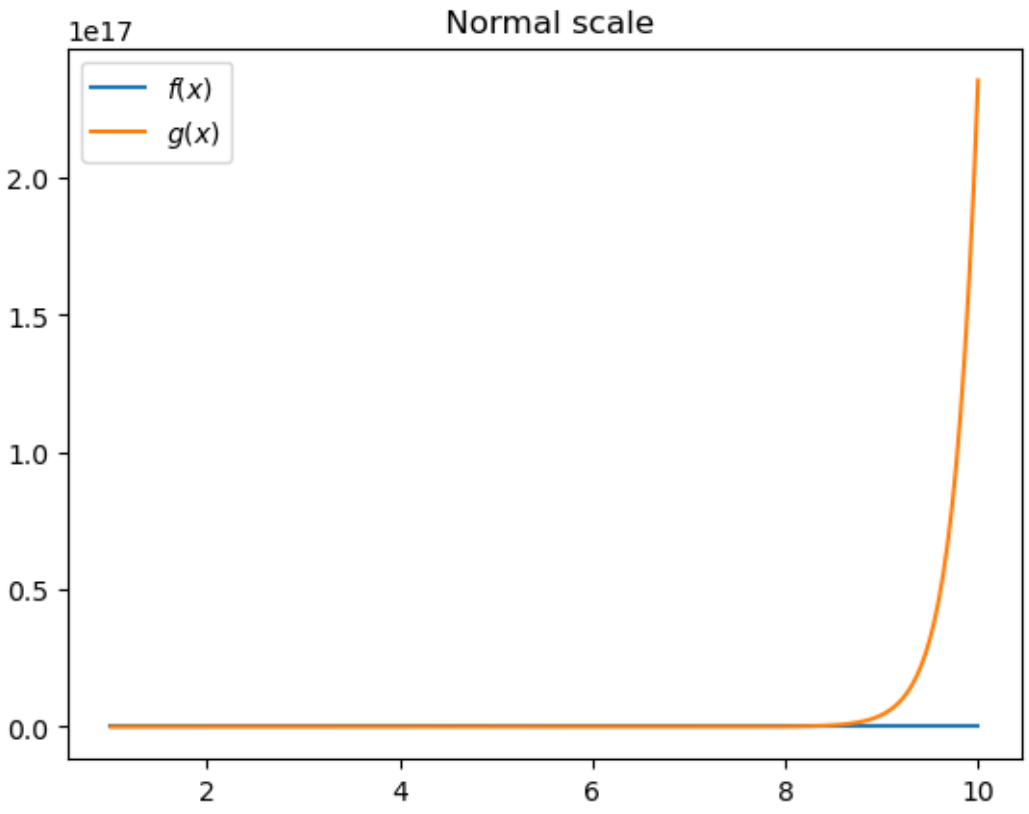
$$f(x) = x^3, \quad x \in [1, 10]$$

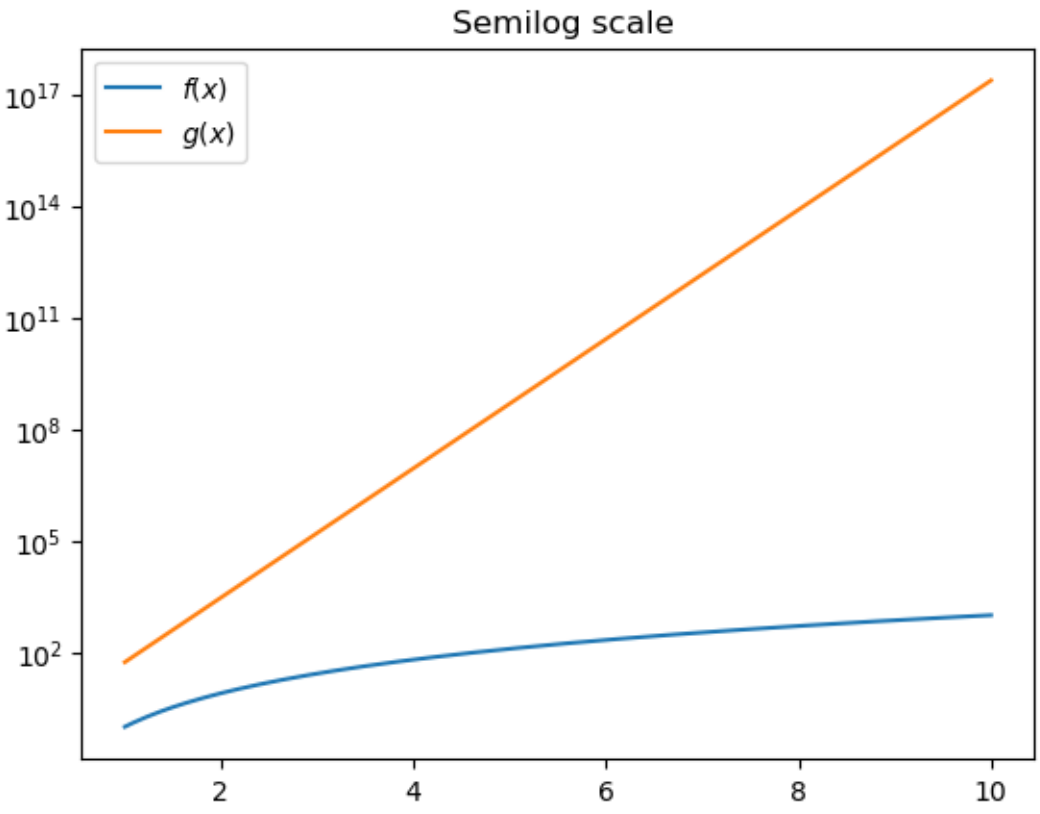
$$g(x) = \exp(4x), \quad x \in [1, 10]$$

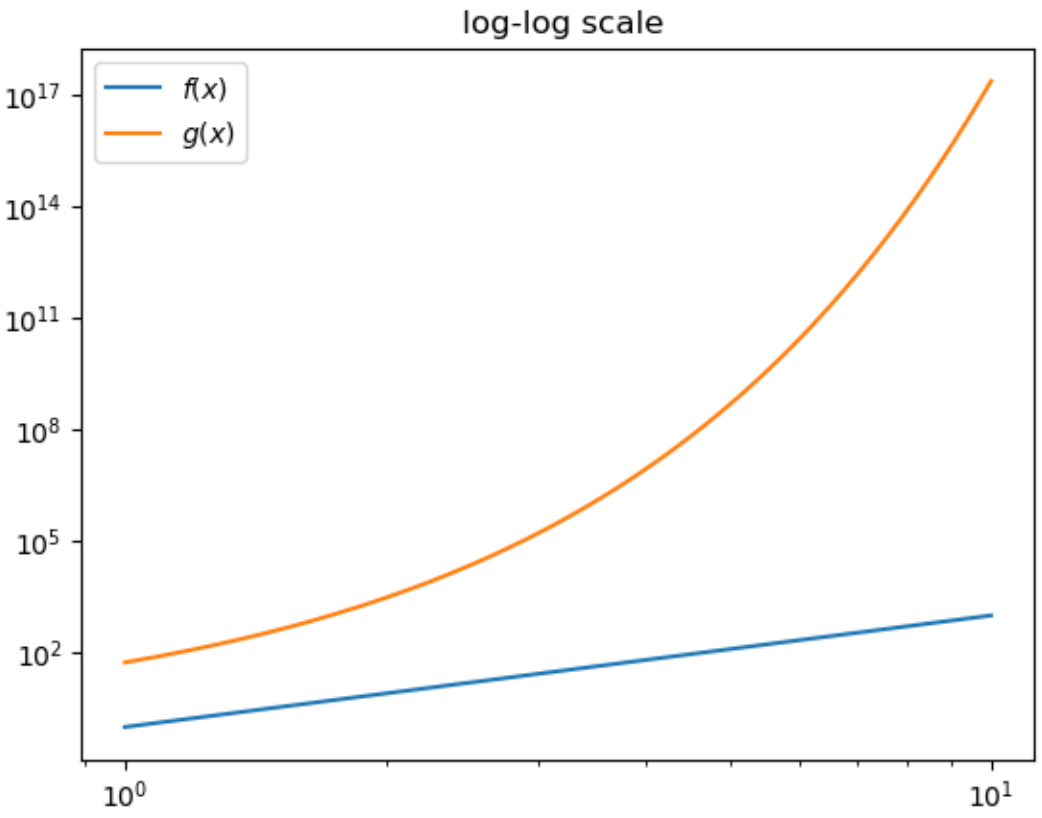
in linear scale (i.e. using the function `plt.plot()` and logarithmic scale (i.e. using the function `plt.semilogy()` and `plt.loglog()`. We will use the definition interval to plot these graphs considering 200 equally distributed points.

Solution

[97]:
```
f=lambda x: x**3
g=lambda x: np.exp(4*x)
x=np.linspace(1,10,200)

#plot ins plot plot
plt.title(' Normal scale')
plt.plot(x,f(x),label=r'$f(x)$')
plt.plot(x,g(x),label=r'$g(x)$')
plt.legend()
plt.show()

#plot in semilogy
plt.title(' Semilog scale')
plt.semilogy(x,f(x),label=r'$f(x)$')
plt.semilogy(x,g(x),label=r'$g(x)$')
plt.legend()
plt.show()


#plot in log-log
plt.title(' log-log scale')
plt.loglog(x,f(x),label=r'$f(x)$')
plt.loglog(x,g(x),label=r'$g(x)$')
plt.legend()
plt.show()
```


Normal scale
1e17
f(x)
g(x)
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10


Semilog scale
f(x)
g(x)
10^17
10^14
10^11
10^8
10^5
10^2
2
4
6
8
10



3.0.7 Exercise 6

1. Given

$$f(x) = \frac{x^2}{2}\sin(x), \quad x \in [1, 20]$$

plot the function f using 10, 20 and 100 equidistant points in the given interval. Plot the three graphs on the same figure with three different colors. Which one gives the best representation of f?

2. Do the same for the functions:

$$g(x) = \frac{x^3}{6}\cos(\sin(x))\exp(-x) + \left(\frac{1}{1+x}\right)^2, \quad x \in [1, 20]$$

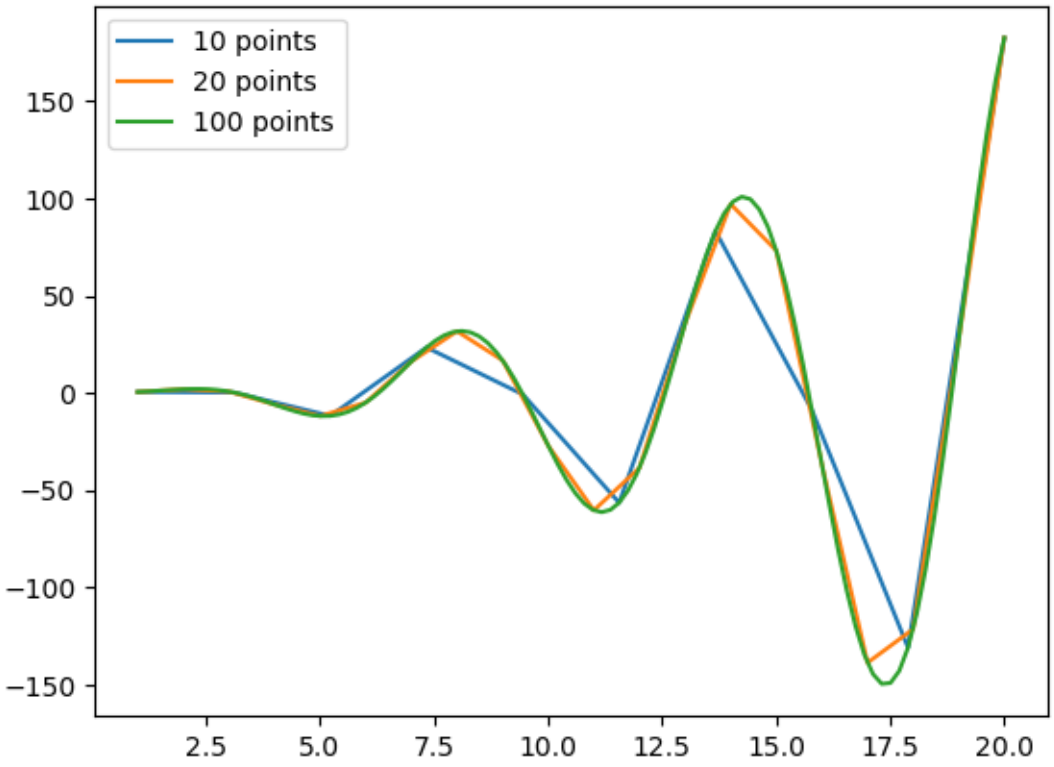
$$h(x) = x(1-x) + \frac{\sin(x)\cos(x)}{x^3}, \quad x \in [1, 20].$$

solution

[98]:
```
f= lambda x: 0.5*x**2*np.sin(x)
x=np.linspace(1,20,10)
y=np.linspace(1,20,20)
z=np.linspace(1,20,100)
```

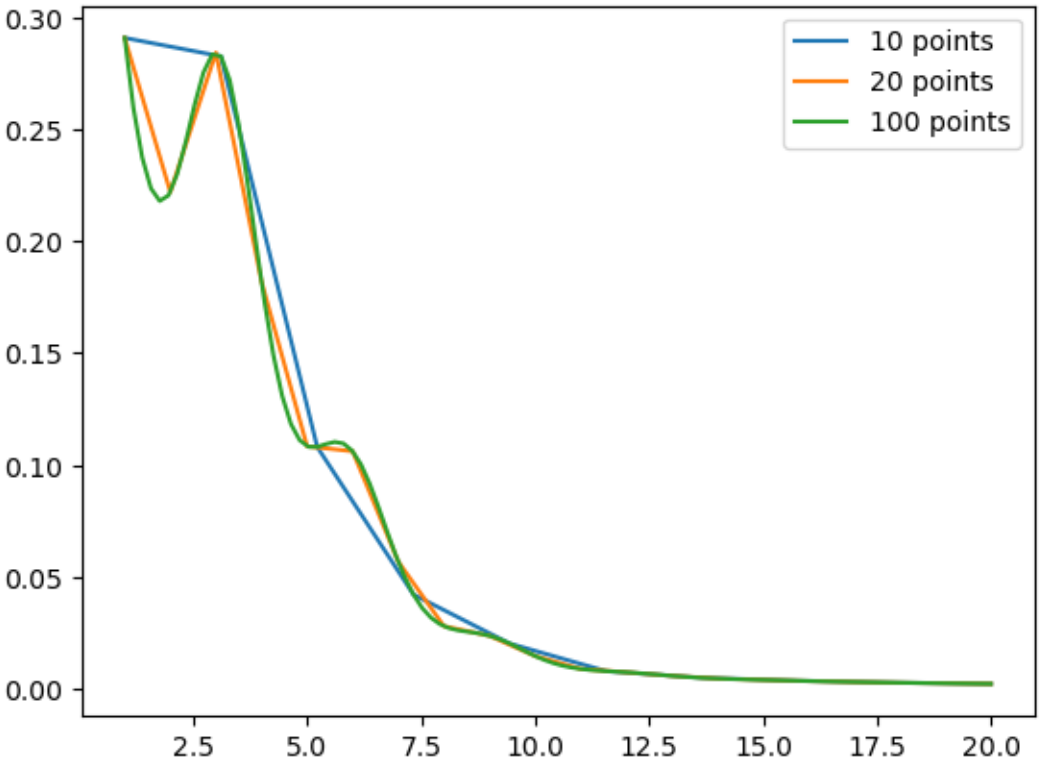
```
plt.plot(x,f(x), label='10 points')
plt.plot(y,f(y), label='20 points')
plt.plot(z,f(z), label='100 points')
plt.legend()
```

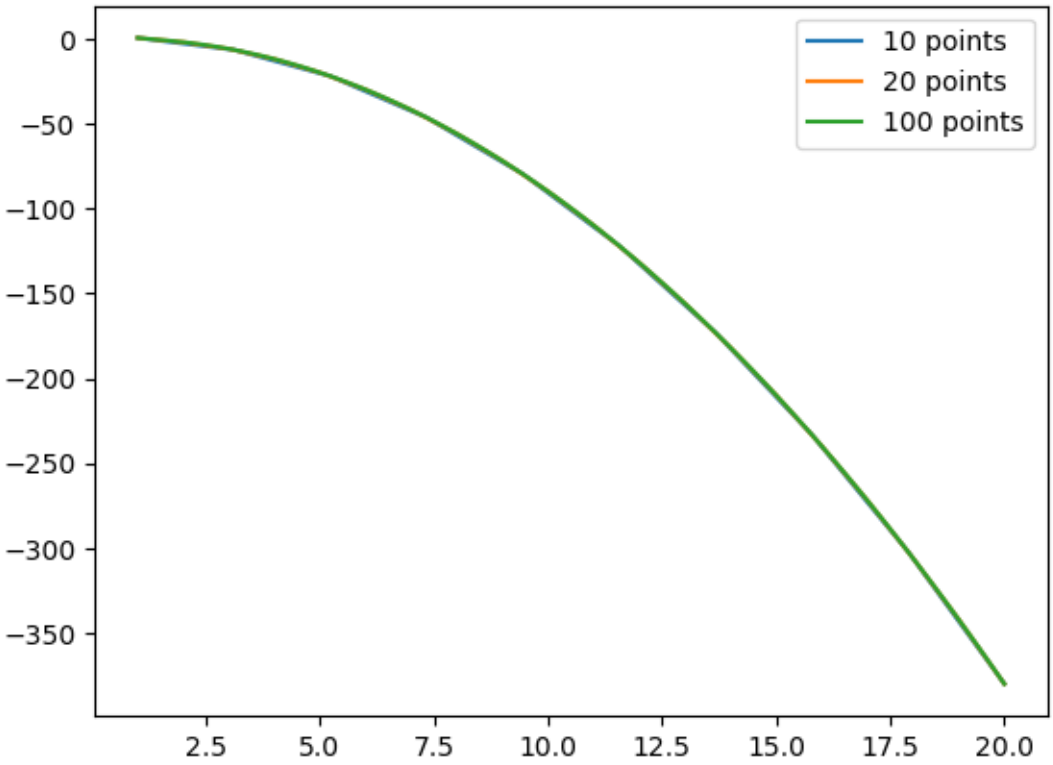
[98]: <matplotlib.legend.Legend at 0x7f88ee8a0e90>



[99]:
```
g= lambda x: x**3/6*np.cos(np.sin(x))*np.exp(-x)+(1/(1+x))**2
x=np.linspace(1,20,10)
y=np.linspace(1,20,20)
z=np.linspace(1,20,100)
plt.plot(x,g(x), label='10 points')
plt.plot(y,g(y), label='20 points')
plt.plot(z,g(z), label='100 points')
plt.legend()
plt.show()

h=lambda x: x*(1-x)+(np.sin(x)*np.cos(x))/(x**3)
x=np.linspace(1,20,10)
y=np.linspace(1,20,20)
z=np.linspace(1,20,100)
plt.plot(x,h(x), label='10 points')
```

```
plt.plot(y,h(y), label='20 points')
plt.plot(z,h(z), label='100 points')
plt.legend()
plt.show()
```




10 points
20 points
100 points
0
−50
−100
−150
−200
−250
−300
−350
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

[ ]: