A short introduction to scientific Python programming

Hans Petter Langtangen [1, 2]
Leif Rune Hellevik [3]

[1] **Center for Biomedical Computing, Simula Research Laboratory**
[2] **Department of Informatics, University of Oslo**
[3] **Biomechanics Group, Department of Structural Engineering NTNU**

Mar 11, 2020

Contents. This note introduces very basic programming elements, such as

- variables for numbers, lists, and arrays
- while loops and for loops
- functifions
- if tests
- plotting

through examples involving a mathematical formula. A glimpse of vectorization of code is also given.

Table of contents

Variables, loops, lists, and arrays

Getting access to Python

Simple mathematical calculations can be done in plain Python, but for more advanced scientific computing, as we do here, several add-on packages are needed. Getting all software correctly installed used to be quite challenging, but today there are several easy-to-use approaches to get access to Python.

Mac and Windows

We recommend to download and install Anaconda, which is a free distribution of Python that comes with most of the packages you need for advanced scientific computing.

Ubuntu

Debian-based Linux systems, such as Ubuntu, can also use Anaconda, but it is more common to use the `apt-get` command-line tool or the Ubuntu installer to install a set of Debian packages. Here is a list of the packages you need to install for this introduction:

```
Terminal> sudo apt-get install python-pip python-setuptools \
  python-scipy python-sympy python-cython python-matplotlib \
  python-dev python-profiler pydb spyder imagemagick gedit vim \
  emacs python-mode git mercurial lib-avtools gnome-terminal
```

In addition, run

```
Terminal> pip install nose
Terminal> pip install pytest
Terminal> pip install ipython --upgrade
Terminal> pip install tornado --upgrade
Terminal> pip install pyzmq --upgrade
```

Web access

You can also access Python directly through a web browser without having it installed on your local computer. We refer to the document How to access Python for doing scientific computing for more details on such tools and also installations based on Anaconda and Ubuntu.

Mathematical example

We shall use a famous mathematical formula in our forthcoming programming examples. The formula tells how long distance ss an object has moved in a time interval $[0, t]$[0,t] if it starts out with a velocity v_0v0 and undergoes constant acceleration aa:

$$s = v_0 t + \frac{1}{2} a t^2 . \tag{1}$$

(1)s=v0t+12at2.

We ma y view ss as a function of tt: $s(t)$ s(t), and also include the parameters in the notation: $s(t; v_0, a)$ s(t;v0,a).

A program for evaluating a formula

Here is a Python program for computing ss, given $t = 0.5$ t=0.5, $v_0 = 2$ v0=2, and $a = 0.2$ a=0.2:

```
t = 0.5
v0 = 2
a = 0.2
s = v0*t + 0.5*a*t**2
print(s)
print('s=%g' % s)
print('s\t = \t %.3f' % s)
```

The program is pure text and must be placed in a *pure text file* using a *text editor*. Popular text editors are Gedit, Nano, Emacs, and Vim on Linux, TextWrangler on Mac OS X, and Notepad++ on Windows. Save the text to a program file whose name ends in `.py`, say `distance.py`.

The program is run in a terminal window (Command Prompt on Windows, Terminal application on Mac OS X, `xterm` or `gnome-terminal` on Linux):

```
Terminal> python distance.py
1.025
```

The result of the `print` statement is the number `1.025` in the terminal window.

As an alternative to writing programs in a text editor and executing them in a terminal window, you may use the Spyder graphical interface which gives a more Matlab-style environment to work in. Anaconda installations come with Spyder.

The code snippet above first contains four *assignment statements* where we assign numbers or the results of arithmetic expressions to *variables*. The variables have names coinciding with the mathematical notation used in the formula: `t`, `v0`, `a`, and `s`. You may think of variables in this programs just as variables in mathematics.

More technical details. A statement like `t = 0.5` works as follows. First, the right-hand side is interpreted by Python as a real number and a `float` object containing the value 0.5 is created. Then the name `t` is defined as a reference to this object.
In the statement `s = v0*t + 0.5*a*t**2`, Python will first go to the right-hand side and observe that it is an arithmetic expression involving variables with known values (or names referring to existing objects). The arithmetic expression is calculated, resulting in a real number that is saved as a `float` object with value 1.025 in the computer's memory.

Everything in Python is an object of some type. Here, `t`, `a`, and `s` are `float` objects, representing real (floating-point) numbers, while `v0` is an `int` object, representing the integer 2. There are many other types of objects: strings, lists, tuples, dictionaries, arrays, files, ...

Formatted output with text and numbers

For any object `s` in Python, `print s` will (normally) print the contents of `s`. However, sometimes we want to combine the content of an object with some text. Say we want to print `s=1.025` rather than just the number. This is easily accomplished using *printf* syntax:

```
print('s=%g' % s)
```

The output is specified as a string, enclosed in single or double quotes. Inside the string, there can be "slots" for numbers (or other objects), indicated by a percentage sign followed by a specification of kind of data that will be inserted at this place. In the string above, there is one such slot, `%g`, where the `g` implies a real number written as compactly as possible.

It is easy to control the number of decimals using printf syntax. Printing out `s=1.03`, i.e., `s` with two decimals, is done by

```
print('s=%.2f' % s)
```

where the `f` signifies a *decimal number* and the preceding `.2` means 2 decimals. Scientific notation, as in `s=1.03E+00` ($1.03 \cdot 10^0$ 1.03·100), is specified as `%.2E` (2 decimals).

The printf syntax is available in numerous programming languages. Python also offers a related variant, called *format string syntax*:

```
print('s={s:.2f}'.format(s=s))
```

While loops

Suppose we want to make a table with two columns, one with t values and one with the corresponding s values. Now we have to repeat a lot of calculations with the formula (1). This is easily done with a *loop*. There are two types of loops in Python: *while* loops and *for* loops.

Let the t values go from 0 to 2 in increments of 0.1. The following program applies a *while* loop:

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t = 0     # Start value
while t <= 2:
    s = v0*t + 0.5*a*t**2
    print(t, s)
    t = t + dt
```

The result of running this program in a terminal window is

```
Terminal> python while.py
0 0.0
0.1 0.201
0.2 0.404
0.3 0.609
0.4 0.816
0.5 1.025
0.6 1.236
0.7 1.449
0.8 1.664
0.9 1.881
1.0 2.1
1.1 2.321
1.2 2.544
1.3 2.769
1.4 2.996
1.5 3.225
1.6 3.456
1.7 3.689
1.8 3.924
1.9 4.161
```

So, how do we interpret the contents of this program? First we initialize four variables: `v0`, `a`, `dt`, and `t`. Everything after `#` on a line is a *comment*and does not affect what happens in the program, but is meant to be of help for a human reading the program. Then comes the while loop:

```
while condition:
    <intented statement>
    <intented statement>
    <intented statement>
```

Observe the colon at the end of the `while` line. The set of indented statements are repeated as long as the expression or variable `condition` evaluates to `True`. In the present case, the condition is the *boolean expression* `t <= 2`, so as long as the value is less than or equal to 2, `t <= 2` evaluates to `True`, otherwise it evaluates to `False`.

> **Error in the code.** According to the code, the last pass in the while loop should correspond to $t = 2$ t=2, but looking at the output, we see that the last print statement has $t = 1.9$ t=1.9. The next test in the while condition involves $t = 2$ t=2 and the boolean condition is expected to be `2 == 2`. However, it seems that the condition is `False` since the computations for $t = 2$ t=2 are not printed. Why do we experience this behavior?

The Python Online Tutor is a great tool to examine the program flow. Consider this little loop run in the Python Online Tutor (view in Chrome):

The Python Online Tutor executes each statement and displays the contents of variables such that you can track the program flow and the evolution of variables.

So, why is not `a` printed for 1.2? That is, why does `a == 1.2` evaluate to `True` when `a` is (expected to be) 2? We must look at the accuracy of `a` to investigate this question and write it out with 16 decimals in scientific notation (printf specification `%.16E`):

The problem is that `da` is not exactly 0.4, but contains a small round-off error. Doing `a = a + da` then results in a slightly inaccurate `a`. When the exact `a` should reach 1.2, the `a` in the program has a an error and equals `1.2000000000000002`. Obviously the test for exact equality, `a == 1.2`, becomes `False`, and the loop is terminated.

```
a = 1.2
b = 0.4 + 0.4 + 0.4
boolean_condition1 = a == b              # False
print(boolean_condition1)
tol = 1E-14
boolean_condition2 = abs(a - b) < tol    # True
print(boolean_condition2)
```

The Python Online Tutor is ideal for demonstrating program flow and contents of variables in small and simple programs. However, you should use a real debugger instead when searching for errors in real programs.

Lists

The table created in the previous section has two columns of data. We could store all the numbers in each column in a *list* object. A list is just a collection of objects, here numbers, in a given sequence. For example,

```
L = [-1, 1, 8.0]
```

is a list of three numbers. Lists are enclosed in square brackets and may contain any type of objects separated by commas. Here we mix a filename (string), a real number, and an integer:

```
L = ['mydata.txt', 3.14, 10]
```

The different list *elements* can be reached via indexing: `L[0]` is the first element, `L[1]` is the second, and `L[-1]` is the last element. Here is an *interactive Python shell* where we can write Python statements and examine the contents of variables as we perform various operations:

```
L = ['mydata.txt', 3.14, 10]
print(L[0])
print(L[1])
del(L[0])  # delete the first element
print(L)
print(len(L))  # length of L
L.append(-1)  # add -1 at the end of the list
print(L)
```

Let us store the numbers in the previous table in lists, one for each column. We can start with empty lists `[]` and use `append` to add a new element to the lists in each pass of the while loop. Thereafter, we can run a new while loop and print the contents of the lists in a nice, tabular fashion:

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t = 0
t_values = []
s_values = []
while t <= 2.1:
    s = v0*t + 0.5*a*t**2
    t_values.append(t)
    s_values.append(s)
    t = t + dt
print(s_values)  # Just take a look at a created list
print(t_values)

# Print a nicely formatted table
i = 0
while i <= len(t_values)-1:
    print('%.18f  %.4f' % (t_values[i], s_values[i]))
    i += 1   # Compact form for i = i + 1
```

The output looks like

```
[0.0, 0.201, 0.404, 0.6090000000000001, 0.8160000000000001,
 1.025, 1.236, 1.4489999999999998, 1.664, 1.8809999999999998,
 2.0999999999999996, 2.3209999999999997, ...]
0.00  0.0000
0.10  0.2010
0.20  0.4040
0.30  0.6090
0.40  0.8160
0.50  1.0250
0.60  1.2360
0.70  1.4490
...
```

Note that `print s_values` here leads to output with many decimals and small round-off errors. To get complete control of the formatting of real numbers in the table, we use the printf syntax.

Lists come with a lot of functionality. See the Python Tutorial for many more examples.

For loops

A for loop is used for visiting elements in a list, one by one:

```
>>> L = [1, 4, 8, 9]
>>> for e in L:
...     print(e)
...
1
4
8
9
```

The variable `e` is successively set equal to the elements in the list, in the right order. Note the colon at the end of the `for` line. The statements to be executed in each pass of the loop (here only `print e`) must be indented. When `e` is set equal to the last element and all indented statements are executed, the loop is over, and the program flow continues with the next statement that is not indented. Try the following code out in the Python Online Tutor:

A for loop over the valid indices in a list is created by

```
for i in range(len(somelist)):
    # Work with somelist[i]
```

The `range` function returns a list of integers: `range(a, b, s)` returns the integers `a, a+s, a+2*s, ...` up to *but not including* `b`. Just writing `range(b)` implies `a=0` and `s=1`, so `range(len(somelist))` returns `[0, 1, 2]`.

For loops over real numbers. The `for i in range(...)` construction can only run a loop over integers. If you need a loop over real values, you have to either create a list with the values first, or use a formula where each value is generated through an integer counter:

```
# Need loop over 0, 0.1, 0.2, ..., 1
values = []
for i in range(11):
    values.append(i*0.1)

# Shorter version using list comprehension (same as the loop above)
values = [i*0.1 for i in range(11)]

for value in values:
    # work with value
```

We can now rewrite our program that used lists and while loops to use for loops instead:

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t_values = []
s_values = []
n = int(round(2/dt)) + 1  # No of t values
for i in range(n):
    t = i*dt
    s = v0*t + 0.5*a*t**2
    t_values.append(t)
    s_values.append(s)
print(s_values)  # Just take a look at a created list

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print('%.2f  %.4f' % (t, s))

# Alternative implementation
for i in range(len(t_values)):
    print('%.2f  %.4f' % (t_values[i], s_values[i]))
```

Observe that we have here used a slightly different technique for computing the t values inside the first loop: now we set t as $i\Delta t$ iΔt, where Δt Δt (`dt` in the code) is the increment (0.1) between each t value. The computation of `n` , the number of t values, makes use of `round` to make a correct mathematical rounding to the nearest integer (and `int` makes an integer object out of the rounded real number). (In an interval $[a, b]$[a,b] divided into subintervals of equal length Δt Δt, there will be $1 + (b - a)/\Delta t$ 1+(b−a)/Δt points in total.)

Running through multiple lists simultaneously is done with the `zip` construction:

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
```

One may instead create a for loop over all the legal index values instead and index each array,

```
for i in range(len(list1)):
    e1 = list1[i]
    e2 = list2[i]
    ...
```

Arrays

Lists are useful for collecting a set of numbers or other objects in a single variable. Arrays are much like lists, but tailored for collection of numbers. The primary advantage of arrays is that you can use them very efficiently and conveniently in mathematical computations, but the downside is that an array has (in practice) a fixed length and all elements must be of the same type. This is usually no important restriction in scientific computations.

Much of Python's functionality are coded in *modules*. To use such functionality, one must *import* the relevant modules. Arrays, for example, are available in the `numpy` module, which must be imported. Functions or variables in the module must be prefixed by `numpy` . This will be demonstrated below. Here is an example involving basic operations with arrays:

```
L = [1, 4, 10.0]    # List of numbers
import numpy
a = numpy.array(L)  # Make corresponding array
print(a)
print(a[1])
print(a.dtype)       # Data type of an element
b = 2*a + 1
print(b)
```

Note that all elements in the `a` array are of `float` type (because one element in `L` was `float`). Arithmetic expressions such as `2*a+1` work with `a` as array, but not as list. In fact, we can pass arrays to mathematical functions:

```
>>> c = numpy.log(a)
>>> print(c)
[ 0.          1.38629436  2.30258509]
```

The `numpy` module has a lot of very useful utilities. To create $n + 1$ n+1 uniformly distributed coordinates in an interval $[a, b]$[a,b], stored in an array, one can use `linspace` :

```
t = numpy.linspace(a, b, n+1)
```

This construction makes it easy to create arrays for the t and s values in our tables:

```
import numpy
v0 = 2
a = 0.2
dt = 0.1  # Increment
n = int(round(2/dt)) + 1  # No of t values

t_values = numpy.linspace(0, 2, n+1)
s_values = v0*t_values + 0.5*a*t_values**2

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print('%.2f  %.4f' % (t, s))
```

Mathematical functions

Python offers access to all standard mathematical functions such as $\sin x$ sinx, $\cos x$ cosx, $\tan x$ tanx, $\sinh x$ sinhx, $\cosh x$ coshx, $\tanh x$ tanhx, all their inverses (called `asin(x)`, `asinh(x)`, and so forth), e^x ex (`exp(x)`), $\ln x$ lnx (`log(x)`), and $x!$ x! (`factorial(x)`). However, one has to import a module to get access to these functions. For scalars (single numbers) the relevant module is `math`:

```
>>> import math
>>> print(math.sin(math.pi))
1.2246467991473532e-16
```

which shows that the sine function is only approximate (to 16 digits). Many prefer to write mathematical expressions without the `math` prefix:

```
from math import sin, pi
print(sin(pi))

# Or import everything from math
from math import *
print(sin(pi), log(e), tanh(0.5))
```

The `numpy` module contains sine, cosine, and other mathematical functions that work on scalars as well as arrays.

Import of `numpy`. To get Python code that is as similar to Matlab as possible, one would do a "start import" of everything,

```
from numpy import *
x = linspace(0, 1, 101)
y = exp(-x)*sin(pi*x)
```

However, in the Python community it has been a culture to use a prefix `np` as abbreviation for `numpy`:

```
import numpy as np
x = np.linspace(0, 1, 101)
y = np.exp(-x)*np.sin(np.pi*x)
```

Our convention is to use the `np` prefix for typical Matlab functions, but skip the prefix when working with mathematical functions like `exp(x)*sin(pi*x)`to get a one-to-one correspondence between formulas in the program and in the mathematical description of the problem.

```
import numpy as np
from numpy import sin, exp, pi
x = np.linspace(0, 1, 101)
y = exp(-x)*sin(pi*x)
```

Plotting

We can easily make a graph of a function $s(t)$ s(t) using the module `matplotlib`. The technique is to compute an array of t t values and a corresponding array of function values $s(t)$ s(t). Plotting programs will draw straight lines between the points on the curve, so a sufficient number of points are needed to give the impression of a smooth curve. Our $s(t)$ s(t) function is plotted by the following code:

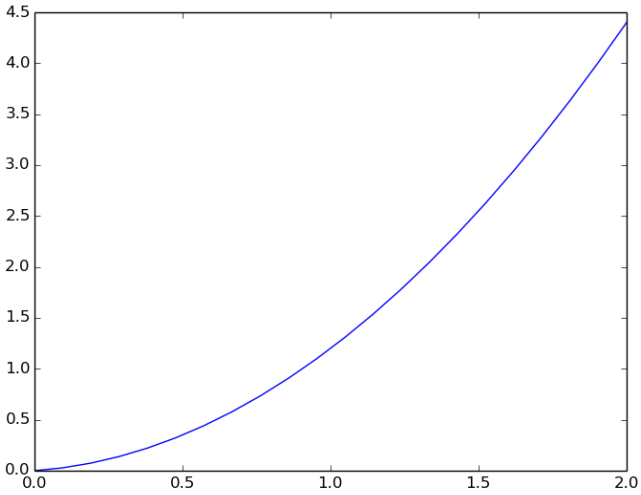
```
import numpy as np
import matplotlib.pyplot as plt

v0 = 0.2
a = 2
n = 21  # No of t values for plotting

t = np.linspace(0, 2, n+1)
s = v0*t + 0.5*a*t**2

plt.plot(t, s)
plt.savefig('myplot.png')
plt.show()
```

The plotfile `myplot.png` looks like



Matlab users may prefer to do

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

such that they can use `linspace` and `plot` without any prefix, just as in Matlab.

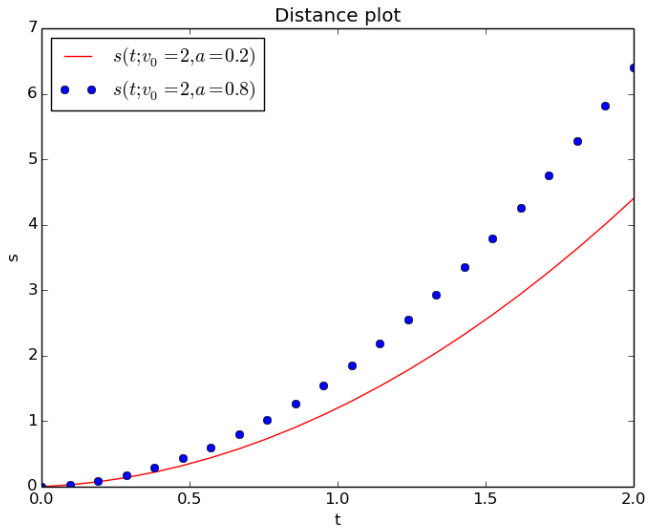
Two curves can easily be plotted, this time also with labels on the axis and a box with legends such that we can distinguish the two curves:

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

v0 = 0.2
a = 2
n = 21  # No of t values for plotting

t = np.linspace(0, 2, n+1)
s = v0*t + 0.5*a*t**2

plt.plot(t, s)
plt.savefig('myplot.png')
plt.show()
```



Functions and branching

Functions

Since $s(t)$ s(t) is a function in mathematics, it is convenient to have it as a function in Python too:

```
def s(t):
    return v0*t + 0.5*a*t**2

v0 = 0.2
a = 4
value = s(3)   # Call the function
```

Note that

- functions start with the keyword `def`
- statements belonging to the function must be indented
- function input is represented by arguments (separated by comma if more than one)
- function output is returned to the calling code

In this program, `v0` and `a` are *global variables*, which must be defined before calling the `s(t)` function, otherwise `v0` and `a` becomes undefined variables in the expression inside the function.
Instead of having `v0` and `a` as global variables, we may let them be function arguments:

```
def s(t, v0, a):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4)   # Call the function

# More readable call
value = s(t=3, v0=0.2, a=4)
```

Function arguments may also be declared to have a default value, allowing us to drop the argument in the call if the default value is appropriate. Such arguments are called *keyword arguments* or *default arguments*. Here is an example:

```
def s(t, v0=1, a=1):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4)         # specify new v0 and a
value = s(3)                 # rely on v0=1 and a=1
value = s(3, a=2)            # rely on v0=1
value = s(3, v0=2)           # rely on a=1
value = s(t=3, v0=2, a=2)    # specify everything
value = s(a=2, t=3, v0=2)    # any sequence allowed
```

Notice.

- Arguments without the argument name are called *positional arguments*.
- Positional arguments must always be listed before the keyword arguments in the function and in any call.
- The sequence of the keyword arguments can be arbitrary.
- If argument names are used for all arguments (as in the last line above) the sequence of the arguments is not important.

Vectorized functions. Applying the function `s(t, v0, a)` to an array `t` can be done in two ways:

```
# Scalar code: work with one element at a time
for i in range(len(t)):
    s_values[i] = s(t_values[i], v0, a)

# Vectorized code: apply s to the entire array
s_values = s(t_values, v0, a)
```

For the last line to work, the function `s` must contain statements that work correctly when `t` is an array argument.

The advantage of vectorized code is that we avoid a loop in Python. Instead, we carry out mathematical operations on entire arrays, e.g., `v0*t` and `a*t**2`. Technically, such (binary) operations are executed as loops in very fast (compiled) C code.

A function can return more than one value, say $s(t)$ s(t) and the velocity $s'(t) = v_0 + at$ s′(t)=v0+at:

```
def movement(t, v0, a):
    s = v0*t + 0.5*a*t**2
    v = v0 + a*t
    return s, v

s_value, v_value = movement(t=0.2, v0=2, a=4)
```

When n n values are returned, we list n n variables on the left-hand side in the call.

Python functions return only one object. Even when we return several values, as in `return s, v`, actually only one object is returned. The `s` and `v` values are packed together in a *tuple* object (which is very similar to a list).

```
>>> def f(x):
...     return x+1, x+2, x+3
...
>>> r = f(3)     # Store all three return values in one object r
>>> print(r)
(4, 5, 6)
>>> type(r)      # What type of object is r?
<type 'tuple'>
>>> print(r[1])
5
```

Tuples are constant lists, so you can index them as lists, but you cannot change the contents (`append` or `del` is illegal).

A more general mathematical formula

The formula (1) arises from the basic differential equations in kinematics:

$$v = \frac{ds}{dt}, \quad s(0) = s_0, \tag{2}$$

$$a = \frac{dv}{dt}, \quad v(0) = v_0\,. \tag{3}$$

(2)v=dsdt,s(0)=s0,(3)a=dvdt,v(0)=v0.

Given any acceleration $a(t)$ a(t), we can solve for $s(t)$ s(t) through integration. First, we integrate to find $v(t)$ v(t):

$$\int_0^t a(t)dt = \int_0^t \frac{dv}{dt}dt,$$

$$\int_0 t a(t)dt = \int_0 t dv dt t,$$

which gives

$$v(t) = v_0 + \int_0^t a(t)dt\,.$$

$$v(t)=v0+\int_0 t a(t)dt.$$

Then we integrate again over $[0, t]$ [0,t] to find $s(t)$ s(t):

$$s(t) = s_0 + v_0 t + \int_0^t (\int_0^t a(t)dt)\,dt\,. \tag{4}$$

$$(4)s(t)=s0+v0t+\int_0 t(\int_0 t a(t)dt)dt.$$

Suppose we have some constant acceleration a_0 a0 in $[0, t_1]$ [0,t1] and no acceleration thereafter. We find

$$s(t) = \begin{cases} s_0 + v_0 t + \frac{1}{2}a_0 t^2, & t \leq t_1 \\ s_0 + v_0 t_1 + \frac{1}{2}a_0 t_1^2 + a_0 t_1 (t - t_1), & t > t_1 \end{cases} \tag{5}$$

$$(5)s(t)=\{s0+v0t+12a0t2,t\leq t1s0+v0t1+12a0t12+a0t1(t-t1),t>t1$$

To implement a function like (5), we need to branch into one type of code (formula) if $t \leq t_1$ t≤t1 and another type of code (formula) if $t > t_1$ t>t1. This is called *branching* and the *if test* is the primary construction to use.

If tests

An if test has the structure

```
if condition:
    <statements when condition is True>
else:
    <statements when condition is False>
```

Here, `condition` is a boolean expression that evaluates to `True` or `False`. For the piecewisely defined function (5) we would use an if test in the implementation:

```
if t <= t1:
    s = v0*t + 0.5*a0*t**2
else:
    s = v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1)
```

The `else` part can be omitted when not needed. Several branches are also possible:

```
if condition1:
    <statements when condition1 is True>
elif condition2:
    <statements when condition1 is False and condition2 is True>
elif condition3:
    <statements when condition1 and conditon 2 are False
     and condition3 is True>
else:
    <statements when condition1/2/3 all are False>
```

A Python function implementing the mathematical function (5) reads

```
def s_func(t, v0, a0, t1):
    if t <= t1:
        s = v0*t + 0.5*a0*t**2
    else:
        s = v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1)
    return s
```

To plot this $s(t)$ s(t), we need to compute points on the curve. Trying to call `s_func` with an array `t` as argument will not work because of the if test. In general, functions with if tests will not automatically work with arrays because a test like `if t <= t1` evaluates to an if applied to a boolean array (`t <= t1` becomes a boolean array, not just a boolean).

One solution is to compute the function values one by one in a loop such that the `s` function is always called with a scalar value for `t` . Appropriate code is

```
n = 201  # No of t values for plotting
t1 = 1.5

t = np.linspace(0, 2, n+1)
s = np.zeros(n+1)
for i in range(len(t)):
    s[i] = s_func(t=t[i], v0=0.2, a0=20, t1=t1)
```

Relevant statements for plotting are now

```
plt.plot(t, s, 'b-')
plt.plot([t1, t1], [0, s_func(t=t1, v0=0.2, a0=20, t1=t1)], 'r--')
plt.xlabel('t')
plt.ylabel('s')
plt.savefig('myplot.png')
plt.show()
```

Vectorization of functions with if tests. To vectorize the computation of the array of $s(t)$ s(t) values, i.e., avoid a loop where `s_func` is called for each `t[i]` element, we must completely rewrite the if test. There are two methods: the `numpy.where` function and array indexing.

Using the `where` function

A vectorized if-else test can be coded as

```
s = np.where(condition, s1, s2)
```

Here, `condition` is an array of boolean values, and `s[i] = s1[i]` if `condition[i]` is `True` , and otherwise `s[i] = s2[i]` .

Our example then becomes

```
s = np.where(t <= t1,
             v0*t + 0.5*a0*t**2,
             v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))
```

Note that `t <= t1` with array `t` and scalar `t1` results in a boolean array `b` where `b[i] = t[i] <= t1` .

Using array indexing

It is possible to index a subset of indices in an array `s` using a boolean array `b` : `s[b]` . This construction picks out all the elements `s[i]` where `b[i]` is `True` . On the right-hand side we can then assign some array expression `expr` of the same length as `s[b]` :

```
s[b] = (expr)[b]
```

Our example can utilize this technique with `b` as `t <= t1` and `t > t1`:

```
s = np.zeros_like(t)  # Make s as zeros, same size & type as t
s[t <= t1] = (v0*t + 0.5*a0*t**2)[t <= t1]
s[t > t1]  = (v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))[t > t1]
```

Array view versus array copying

Arrays are usually large and most array libraries have carefully designed functionality for avoiding unnecessary copying of large of amounts of data. If you do a simple assignment,

```
a = np.linspace(1, 5, 5)
b = a
```

`b` becomes a just *view* of `a`: the variables `b` and `a` point to the same data. In other languages one may say that `b` is a pointer or reference to the array. This means that changing `a` changes `b` and vice versa:

```
array([ 1.,  2.,  3.,  4.,  5.])
>>> b[0] = 5                        # changes a[0] to 5
>>> a
array([ 5.,  2.,  3.,  4.,  5.])
>>> a[1] = 9                        # changes b[1] to 9
>>> b
array([ 5.,  9.,  3.,  4.,  5.])
```

Similarly, `b[1:-1]` gives a view to a subset of `a` and no copy of data. If we want to change `b` without affecting `a`, a copy of the array is required:

```
>>> c = a.copy()       # copy all elements to new array c
>>> c[0] = 6           # a is not changed
>>> a
array([ 1.,  2.,  3.,  4.,  5.])
>>> c
array([ 6.,  2.,  3.,  4.,  5.])
>>> b
array([ 5.,  2.,  3.,  4.,  5.])
```

Note that `b` remains unchanged by the change in `c` since the `b` and `c` variables now refer to different data. Copying of the elements of a sub-array is also done by the `copy()` method: `b = a[1:-1].copy()`.

Understand the difference between view and copy! Mathematical errors and/or inefficient code may easily arise from confusion between array view and array copying. Make sure you understand the implication of any variable assignment to arrays!

```
a = np.array([-1, 4.5, 8, 9])
b = a[1:-2]            # view: changes in a are reflected in b
b = a[1:-2].copy()     # copy: changes in a are not reflected in b
```

Linear systems

Solving linear systems of the form $Ax = b$ Ax=b with a matrix AA and vectors xx and bb is a frequently encountered task. A sample 2×2 2×2 system can be coded as

```
import numpy as np
A = np.array([[1., 2],
              [4,  2]])
b = np.array([1., -2])
x = np.linalg.solve(A, b)
```

Many scientific computing problems involves large matrices with special *sparse* structures. Significant memory and computing time can be saved by utilizing *sparse matrix* data structures and associated algorithms. Python has sparse matrix package scipy.sparse that can be used to construct various types of sparse matrices. The particular function `scipy.sparse.spdiags` can construct a sparse matrix from a few vectors representing the diagonals in the matrix (all the diagonals must have the same length as the dimension of their sparse matrix, consequently some elements of the diagonals are not used: the first kk elements are not used of the kk super-diagonal, and the last kk elements are not used of the $-k$ −k sub-diagonal). Here is an example of constructing a *tridiagonal* matrix:

```
>>> import numpy as np
>>> N = 6
>>> diagonals = np.zeros((3, N))   # 3 diagonals
diagonals[0,:] = np.linspace(-1, -N, N)
diagonals[1,:] = -2
diagonals[2,:] = np.linspace(1, N, N)
>>> import scipy.sparse
>>> A = scipy.sparse.spdiags(diagonals, [-1,0,1], N, N, format='csc')
>>> A.toarray()    # look at corresponding dense matrix
[[-2.  2.  0.  0.  0.  0.]
 [-1. -2.  3.  0.  0.  0.]
 [ 0. -2. -2.  4.  0.  0.]
 [ 0.  0. -3. -2.  5.  0.]
 [ 0.  0.  0. -4. -2.  6.]
 [ 0.  0.  0.  0. -5. -2.]]
```

Solving a linear system with a tridiagonal coefficient matrix requires a special function with a special algorithm to take advantage of the matrix' structure. The next example chooses a solution `x`, computes the corresponding right-hand side $b = Ax$ b=Ax using the sparse matrix vector product associated with the sparse matrix AA, and then solves $Ax = b$ Ax=b:

```
>>> x = np.linspace(-1, 1, N)  # choose solution
>>> b = A.dot(x)               # sparse matrix vector product
>>> import scipy.sparse.linalg
>>> x = scipy.sparse.linalg.spsolve(A, b)
>>> print(x)
[-1.  -0.6 -0.2  0.2  0.6  1. ]
```

Although we can easily check that `x` is correctly computed, we can do another check by converting the linear system to its dense counterpart:

```
>>> A_d = A.toarray()            # corresponding dense matrix
>>> b = np.dot(A_d, x)           # standard matrix vector product
>>> x = np.linalg.solve(A_d, b)  # standard Ax=b algorithm
>>> print(x)
[-1.  -0.6 -0.2  0.2  0.6  1. ]
```

Files

This section outlines basic file handling in Python, including file utilities in the `numpy` package.

File reading

Suppose we create a file with typical input data to our little demo program for evaluating the formula $s(t) = v_0 t + \frac{1}{2}at^2$ s(t)=v0t+12at2:

```
v0 = 2
a = 0.2
dt = 0.1
interval = [0, 2]
```

We want to read the parameters in the file into Python variables and create a table with columns of t t and $S(t)$ S(t)for $t \in [0, 2]$t∈[0,2] with steps of $\Delta t = 0.1$ Δt=0.1 (as specified by `dt` and `interval` in the file).

The code for reading the lines in the file, interpreting them, and assigning values to variables is given next.

```
infile = open('.input.dat', 'r')
for line in infile:
    # Typical line: variable = value
    variable, value = line.split('=')
    variable = variable.strip()  # remove leading/traling blanks
    if variable == 'v0':
        v0 = float(value)
    elif variable == 'a':
        a = float(value)
    elif variable == 'dt':
        dt = float(value)
    elif variable == 'interval':
        interval = eval(value)
infile.close()
```

The name of the file is here `.input.dat` , and it is opened with the parameter `'r'` for reading. The for loop `for line in infile` will read the lines from the file, one by one, and in the current line is available in the string `line` . To split a line into words separated by a character '=', we use `line.split('=')` , resulting in a list of the words. For example,

```
>>> line = 'v0 = 5.3'
>>> variable, value = line.split('=')
>>> variable
'v0 '
>>> value
' 5.3'
```

Note that there are blanks in the strings. Leading and trailing blanks can be removed by `variable.strip()` . We do this before comparing the variable name with `v0` , `a` , etc.

It is important to note that `value` (the text to the right of `=` on each line) is a string variable. We need to convert to a `float` object to get a variable that we can compute with. The assignment `interval = eval(value)` does some magic and deserve an explanation: `eval(s)` interprets the text in the string `s` as Python code. In the present example, `value` is `[0, 2]` and this is interpreted as Python code, i.e., as a list, and `interval` becomes a name for a list object with content `[0, 2]` .

A more modern Python way of opening files is

```
with open('.input.dat', 'r') as infile:
    for line in infile:
        ...
```

Now it is not necessary to close the file as it will be automatically done after the `with` block.

File writing

Suppose we generate t and $s(t)$ s(t) values in two lists, `t_values` and `s_values`, and want to write these as a nicely formatted table to file. The following code does the work:

```
outfile = open('table1.dat', 'w')
outfile.write('# t    s(t)\n')  # write table header
for t, s in zip(t_values, s_values):
    outfile.write('%.2f  %.4f\n' % (t, s))
```

Files intended for writing must be opened with a `'w'` parameter. The key statement is `outfile.write(s)` for writing a string to a file (recall that while `print` automatically adds a newline at the end of the string to be printed, `outfile.write(s)` does not append a newline so `s` must contain the newline).

The `numpy` package contains a convenient function `savetxt` for saving tabular data. The data must be stored in a two-dimensional `numpy` array. The `savetxt` function enables control of the format of the numbers in each column (`fmt`) through the printf syntax, a header can be added (`header`) and the header lines begin with a comment character (`comment`). The code reads

```
import numpy as np
# Make two-dimensional array of [t, s(t)] values in each row
data = np.array([t_values, s_values]).transpose()

# Write data array to file in table format
np.savetxt('table2.dat', data, fmt=['%.2f', '%.4f'],
           header='t   s(t)', comments='# ')
```

The tabular data in the file can be read back into a `numpy` array by the `loadtxt` function:

```
data = np.loadtxt('table2.dat', comments='#')
```

Lines beginning with the comment character are skipped in the reading. The resulting object `data` is a two-dimensional array: `data[i,j]` contains row number `i` and column number `j` in the table, i.e., `data[i,0]` holds the t t value and `data[i,1]` the $s(t)$ s(t) value in the `i` -th row.

Classes

All objects in Python are in fact implemented as classes, but you can program with objects without knowing about classes. Nevertheless, the class concept is a powerful tool and some basic knowledge will make it easier to understand much useful Python information that is floating around.

A very simple class

A class packs together a set of variables and a set of functions. All functions can access all variables. [1] The idea is to encapsulate variables and functions in logical units such that a larger program can be composed by combing such units (classes).

1: In classes, the functions are called *methods* and the variables are called *attributes*.

Here is a trivial class that has one variable `a` and one function `dump` that writes the contents of `a`:

```
class Trivial:
    def __init__(self, a):
        self.a = a

    def dump(self):
        print(self.a)
```

How can we use this class? First, we must make an *instance* (object) of the class:

```
t = Trivial(a=4)
```

The syntax `Trivial(a=4)` implies a call to the `__init__` method (function) with `4` as the value of the argument `a`. Inside `__init__`, we store the value of `a` as an attribute in the class: `self.a`. If there were several methods in the class, all of them could then access `self.a` (as if `a` were some global variable in the class). The `__init__` function is called a *constructor* since it is used to construct an instance (object) of the class.

Having an instance `t` of class `Trivial`, we can call the `dump` method as follows:

```
t.dump()
```

Even though both `__init__` and `dump` have `self` as first argument, this argument *is not used in a call*.

The `self` argument. It takes time and experience to understand the `self` argument in class methods.

1. `self` must always be the first argument.
2. `self` is never used in calls.
3. `self` is used to access attributes and methods inside methods.

We refer to a more comprehensive text on classes for better explanation of `self`.

A class for representing a mathematical function

The Python implementation of the mathematical function

$$s(t; v_0, a) = v_0 t + \frac{1}{2} a t^2$$

s(t;v0,a)=v0t+12at2

can benefit from being implemented as a class. The reason is that s s is a function of one variable, t t, so it should be called as $s(t)$ s(t), but the function also contains two parameters, v_0 v0 and a a. Using a class, we can pack v_0 v0 and a a together with a function computing $s(t)$ s(t) and that can be called with one argument.

The class code

```
class Distance:
    def __init__(self, v0, a):
        self.v0 = v0
        self.a = a

    def __call__(self, t):
        v0, a = self.v0, self.a  # make local variables
        return v0*t + 0.5*a*t**2
```

Dissection

The class has two methods (functions). The name of a method can be freely chosen by the programmer, say `dump` as we used above, but here we have used three special names, starting and ending with double underscores, which allows us to use special attractive syntax in the calls (such methods are actually known as*special methods*).

The constructor `__init__` has one purpose: storing data in class attributes, here `self.v0` and `self.a` . We can then access these data in class methods.

The `__call__` method is used to evaluate $s(t)$ s(t). It has one argument (`self` does not count since it is never used in calls). This special methods allows us to view a class instance as if were a function. Let us illustrate by some code:

```
s = Distance(v0=2, a=0.5)  # create instance
v = s(t=0.2)               # runs s.__call__(t=0.2)
```

The last line has some magic: `s` is a class instance, not a function, but still we can write `s(t=0.2)` or `s(0.2)` as if `s` were a function. This is the purpose of the special method `__call__` : to allow such syntax.
Actually, `s(0.2)` means `s.__call__(0.2)` . The nice result is that `s` looks like a function, it takes one argument `t` , as the mathematical function $s(t)$ s(t), but it also contains values of the two parameters v_0v0 and aa.

In general, for a function $f(x, y, z; p_1, p_2, \ldots, p_n)$ f(x,y,z;p1,p2,...,pn), here with three independent variables and nn parameters $p_1, p_2, \ldots, p_n$ p1,p2,...,pn, we can pack the function and the parameters together in a class:

```
class F:
    def __init__(self, p1, p2, ...):
        self.p1 = p1
        self.p2 = p2
        ...

    def __call__(self, x, y, z):
        # return formula involving x, y, z and self.p1, self.p2 ...
```

The f f function is initialized by

```
f = F(p1=..., p2=..., ...)
```

and later called as `f(x, y, z)` .

Exercises

Exercise 1: Program a formula

a) Make a simplest possible Python program that calculates and prints the value of the formula

$$y = 6x^2 + 3x + 2, \]for\$x = 2\$.$$
y=6x2+3x+2,\]for$x=2$.

Solution.

The complete program reads

```
x = 2
y = 6*x**2 + 3*x + 2
print(y)
```

b) Make a Python function that takes x x as argument and returns yy. Call the function for $x = 2$ x=2 and print the answer.

Solution.

Exercise 2: Combine text and numbers in output

Let $y = x^2$ y=x2. Make a program that writes the text

```
y(2.550)=6.502
```

if $x = 2.55$ x=2.55. The values of xx and yy should be written with three decimals. Run the program for $x = \pi$ x=π too (the value if ππ is available as the variable `pi` in the `math` module).

Solution.

Here is the code:

```
x = 2.55
y = x**2
print('y(%.3f)=%.3f' % (x, y))
```

Changing $x = 2.55$ x=2.55 to $x = \pi$ x=π,

```
from math import pi
x = pi
y = x**2
print('y(%.3f)=%.3f' % (x, y))
```

gives the output `y(3.142)=9.870` .

Exercise 3: Program a while loop

Define a sequence of numbers,

$$x_n = n^2 + 1,$$
xn=n2+1,

for integers $n = 0, 1, 2, \ldots, N$ n=0,1,2,…,N. Write a program that prints out x_n xn for $n = 0, 1, \ldots, 20$ n=0,1,…,20 using a while loop.

Solution.

Complete program:

```
n = 0
while n <= 20:
    x_n = n**2 + 1
    print('x%d=%d' % (n, x_n))
    n = n + 1
```

Exercise 4: Create a list with a while loop

Store all the x_n xn values computed in Exercise 3: Program a while loop in a list (using a while loop). Print the entire list (as one object).

Solution.

Code:

```
n = 0
x = []  # the x_n values
while n <= 20:
    x.append(n**2 + 1)
    n = n + 1
print(x)
```

Exercise 5: Program a for loop

Do Exercise 4: Create a list with a while loop, but use a for loop.

Solution.

Code:

```
x = []
for n in range(21):
    x.append(n**2 + 1)
print(x)
```

One can also make the code shorter using a list comprehension:

```
x = [n**2 + 1 for n in range(21)]
print(x)
```

Exercise 6: Write a Python function

Write a function `x(n)` for computing an element in the sequence $x_n = n^2 + 1$ xn=n2+1. Call the function for $n = 4$ n=4and write out the result.

Solution.

Code:

```
def x(n):
    return n^2 + 1

print(x(4))
```

Exercise 7: Return three values from a Python function

Write a Python function that evaluates the mathematical functions $f(x) = \cos(2x)$ f(x)=cos(2x), $f'(x) = -2\sin(2x)$ f′(x)=−2sin(2x), and $f''(x) = -4\cos(2x)$ f″(x)=−4cos(2x). Return these three values. Write out the results of these values for $x = \pi$ x=π.

Solution.

Code:

```
from math import sin, cos, pi

def deriv2(x):
    return cos(2*x), -2*sin(2*x), -4*cos(2*x)

f, df, d2f = deriv2(x=pi)
print(f, df, d2f)
```

Running the program gives

```
Terminal> python deriv2.py
1.0 4.89858719659e-16 -4.0
```

as expected.

Exercise 8: Plot a function

Make a program that plots the function $g(y) = e^{-y}\sin(4y)$ g(y)=e−ysin(4y) for $y \in [0, 4]$y∈[0,4] using a red solid line. Use 500 intervals for evaluating points in $[0, 4]$[0,4]. Store all coordinates and values in arrays. Set labels on the axis and use a title "Damped sine wave".

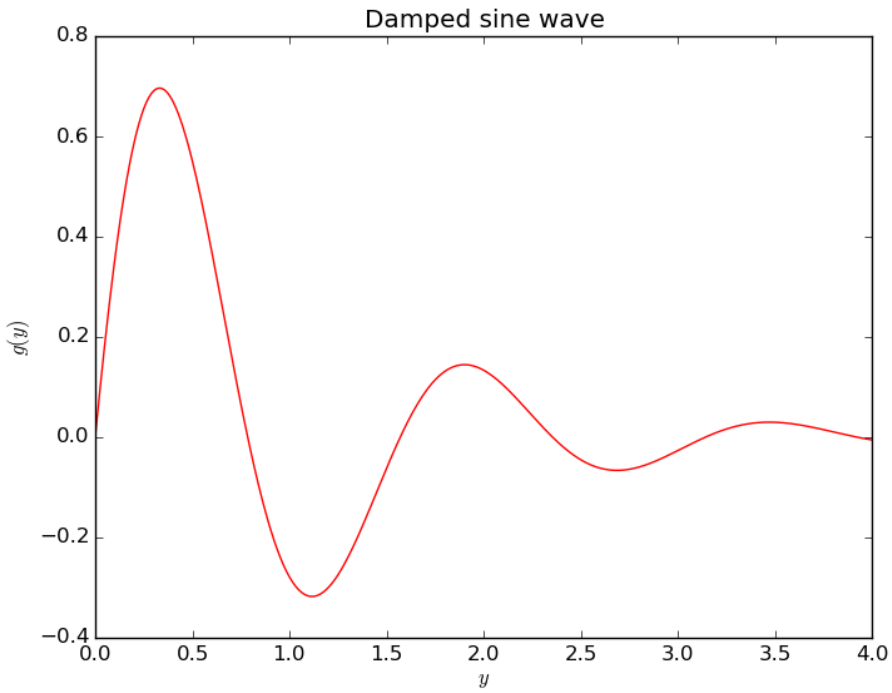
Solution.

Appropriate code is

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import exp, sin  # avoid np. prefix in g(y) formula

def g(y):
    return exp(-y)*sin(4*y)

y = np.linspace(0, 4, 501)
values = g(y)
plt.figure()
plt.plot(y, values, 'r-')
plt.xlabel('$y$'); plt.ylabel('$g(y)$')
plt.title('Damped sine wave')
plt.savefig('tmp.png'); plt.savefig('tmp.pdf')
plt.show()
```



Exercise 9: Plot two functions

As Exercise 9: Plot two functions, but add a black dashed curve for the function $h(y) = e^{-\frac{3}{2}y} \sin(4y)$ h(y)=e−32ysin(4y). Include a legend for each curve (with names `g` and `h`).

Solution.

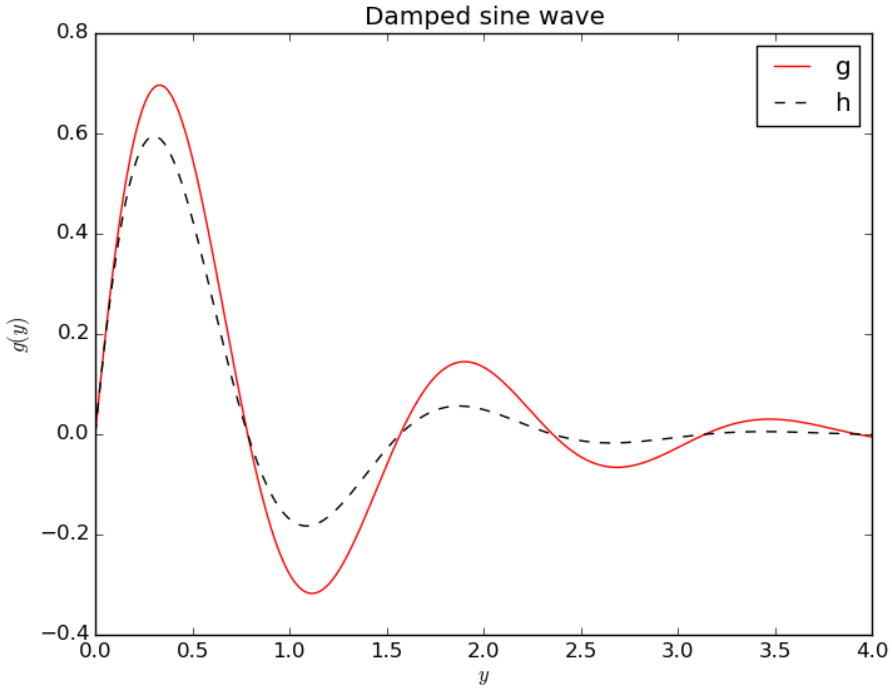
Here is the program:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import exp, sin  # avoid np. prefix in g(y) and h(y)

def g(y):
    return exp(-y)*sin(4*y)

def h(y):
    return exp(-(3./2)*y)*sin(4*y)

y = np.linspace(0, 4, 501)
plt.figure()
plt.plot(y, g(y), 'r-', y, h(y), 'k--')
plt.xlabel('$y$'); plt.ylabel('$g(y)$')
plt.title('Damped sine wave')
plt.legend(['g', 'h'])
plt.savefig('tmp.png'); plt.savefig('tmp.pdf')
plt.show()
```



Exercise 10: Measure the efficiency of vectorization

IPython an enhanced interactive shell for doing computing with Python. IPython has some user-friendly functionality for quick testing of the efficiency of different Python constructions. Start IPython by writing `ipython` in a terminal window. The interactive session below demonstrates how we can use the timer feature `%timeit` to measure the CPU time required by computing $\sin(x)$ sin(x), where x x is an array of 1M elements, using scalar computing with a loop (function `sin_func`) and vectorized computing using the `sin` function from `numpy` .

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

In [2]: n = 1000000

In [3]: x = np.linspace(0, 1, n+1)

In [4]: def sin_func(x):
   ...:     r = np.zeros_like(x)  # result
   ...:     for i in range(len(x)):
   ...:         r[i] = np.sin(x[i])
   ...:     return r
   ...:

In [5]: %timeit y = sin_func(x)
1 loops, best of 3: 2.68 s per loop

In [6]: %timeit y = np.sin(x)
10 loops, best of 3: 40.1 ms per loop
```

Here, `%timeit` ran our function once, but the vectorized function 10 times. The most relevant CPU times measured are listed, and we realize that the vectorized code is $2.68/(40.1/1000) \approx 67$ 2.68/(40.1/1000)≈67 times faster than the loop-based scalar code.

Use the recipe above to investigate the speed up of the vectorized computation of the $s(t)$ s(t) function in the section Functions.