```
N = 5
x = 0
for i in range(1, N+1):
    x += 2*i
print x
```

Executing this code will print the number 30 to the screen. Note in particular how the *accumulation variable* x is initialized to zero. The value of x then gets updated with each iteration of the loop, and not until the loop is finished will x have the correct value. This way of building up the value is very common in programming, so make sure you understand it by simulating the code segment above by hand. It is a technique used with loops in any programming language.

2.4 While Loops

Python also has another standard loop construction, the *while loop*, doing iterations with a loop index very much like the `for` loop. To illustrate what such a loop may look like, we consider another modification of `ball_plot.py` in Sect. 1.4. We will now change it so that it finds the time of flight for the ball. Assume the ball is thrown with a slightly lower initial velocity, say 4.5 ms^{-1}, while everything else is kept unchanged. Since we still look at the first second of the flight, the heights at the end of the flight become negative. However, this only means that the ball has fallen below its initial starting position, i.e., the height where it left the hand, so there is no problem with that. In our array y we will then have a series of heights which towards the end of y become negative. Let us, in a program named `ball_time.py` find the time when heights start to get negative, i.e., when the ball crosses $y = 0$. The program could look like this

```
from numpy import linspace

v0 = 4.5                    # Initial velocity
g = 9.81                    # Acceleration of gravity
t = linspace(0, 1, 1000)    # 1000 points in time interval
y = v0*t - 0.5*g*t**2       # Generate all heights

# Find where the ball hits y=0
i = 0
while y[i] >= 0:
    i += 1

# Now, y[i-1]>0 and y[i]<0 so let's take the middle point
# in time as the approximation for when the ball hits h=0
print "y=0 at", 0.5*(t[i-1] + t[i])

# We plot the path again just for comparison
import matplotlib.pyplot as plt
plt.plot(t, y)
plt.plot(t, 0*t, 'g--')
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()
```

If you type and run this program you should get

```
y=0 at 0.917417417417
```

The new thing here is the `while` loop only. The loop (note colon and indentation) will run as long as the boolean expression `y[i] > 0` evaluates to `True`. Note that the programmer introduced a variable (the loop index) by the name `i`, initialized it (`i = 0`) before the loop, and updated it (`i += 1`) in the loop. So for each iteration, `i` is *explicitly* increased by 1, allowing a check of successive elements in the array `y`.

Compared to a `for` loop, the programmer does not have to specify the number of iterations when coding a `while` loop. It simply runs until the boolean expression becomes `False`. Thus, a loop index (as we have in a `for` loop) is not required. Furthermore, if a loop index is used in a `while` loop, it is not increased automatically; it must be done explicitly by the programmer. Of course, just as in `for` loops and `if` blocks, there might be (arbitrarily) many code lines in a `while` loop. Any `for` loop may also be implemented as a `while` loop, but `while` loops are more general so not all of them can be expressed as a `for` loop.

A problem to be aware of, is what is usually referred to as an *infinite loop*. In those unintentional (erroneous) cases, the boolean expression of the `while` test never evaluates to `False`, and the program can not escape the loop. This is one of the most frequent errors you will experience as a beginning programmer. If you accidentally enter an infinite loop and the program just hangs forever, press `Ctrl+c` to stop the program.

2.5 Lists and Tuples – Alternatives to Arrays

We have seen that a group of numbers may be stored in an array that we may treat as a whole, or element by element. In Python, there is another way of organizing data that actually is much used, at least in non-numerical contexts, and that is a construction called *list*.

A list is quite similar to an array in many ways, but there are pros and cons to consider. For example, the number of elements in a list is allowed to change, whereas arrays have a fixed length that must be known at the time of memory allocation. Elements in a list can be of different type, i.e you may mix integers, floats and strings, whereas elements in an array must be of the same type. In general, lists provide more flexibility than do arrays. On the other hand, arrays give faster computations than lists, making arrays the prime choice unless the flexibility of lists is needed. Arrays also require less memory use and there is a lot of ready-made code for various mathematical operations. Vectorization requires arrays to be used.

The `range()` function that we used above in our `for` loop actually returns a list. If you for example write `range(5)` at the prompt in `ipython`, you get `[0, 1, 2, 3, 4]` in return, i.e., a list with 5 numbers. In a for loop, the line `for i in range[5]` makes `i` take on each of the numbers $0, 1, 2, 3, 4$ in turn, as we saw above. Writing, e.g., `x = range(5)`, gives a list by the name `x`, containing those five numbers. These numbers may now be accessed (e.g., as `x[2]`, which contains the number 2) and used in computations just as we saw for array elements. As with arrays, indices run from 0 to $n - 1$, when `n` is the number of elements in a list. You may convert a list to an array by `x = array(L)`.

A list may also be created by simply writing, e.g.,

```
x = ['hello', 4, 3.14, 6]
```

giving a list where x[0] contains the string hello, x[1] contains the integer 4, etc. We may add and/or delete elements anywhere in the list as shown in the following example.

```
x = ['hello', 4, 3.14, 6]
x.insert(0, -2) # x then becomes [-2, 'hello', 4, 3.14, 6]
del x[3]        # x then becomes [-2, 'hello', 4, 6]
x.append(3.14)  # x then becomes [-2, 'hello', 4, 6, 3.14]
```

Note the ways of writing the different operations here. Using append() will always increase the list at the end. If you like, you may create an empty list as x = [] before you enter a loop which appends element by element. If you need to know the length of the list, you get the number of elements from len(x), which in our case is 5, after appending 3.14 above. This function is handy if you want to traverse all list elements by index, since range(len(x)) gives you all legal indices. Note that there are many more operations on lists possible than shown here.

Previously, we saw how a for loop may run over array elements. When we want to do the same with a list in Python, we may do it as this little example shows,

```
x = ['hello', 4, 3.14, 6]
for e in x:
    print 'x element: ', e
print 'This was all the elements in the list x'
```

This is how it usually is done in Python, and we see that e runs over the elements of x directly, avoiding the need for indexing. Be aware, however, that when loops are written like this, you can not change any element in x by "changing" e. That is, writing e += 2 will not change anything in x, since e can only be used to read (as opposed to overwrite) the list elements.

There is a special construct in Python that allows you to run through all elements of a list, do the same operation on each, and store the new elements in another list. It is referred to as *list comprehension* and may be demonstrated as follows.

```
List_1 = [1, 2, 3, 4]
List_2 = [e*10 for e in List_1]
```

This will produce a new list by the name List_2, containing the elements 10, 20, 30 and 40, in that order. Notice the syntax within the brackets for List_2, for e in List_1 signals that e is to successively be each of the list elements in List_1, and for each e, create the next element in List_2 by doing e*10. More generally, the syntax may be written as

```
List_2 = [E(e) for e in List_1]
```

where E(e) means some expression involving e.

In some cases, it is required to run through 2 (or more) lists at the same time. Python has a handy function called `zip` for this purpose. An example of how to use `zip` is provided in the code `file_handling.py` below.

We should also briefly mention about *tuples*, which are very much like lists, the main difference being that tuples cannot be changed. To a freshman, it may seem strange that such "constant lists" could ever be preferable over lists. However, the property of being constant is a good safeguard against unintentional changes. Also, it is quicker for Python to handle data in a tuple than in a list, which contributes to faster code. With the data from above, we may create a tuple and print the content by writing

```
x = ('hello', 4, 3.14, 6)
for e in x:
    print 'x element: ', e
print 'This was all the elements in the tuple x'
```

Trying `insert` or `append` for the tuple gives an error message (because it cannot be changed), stating that the tuple object has no such attribute.

2.6 Reading from and Writing to Files

Input data for a program often come from files and the results of the computations are often written to file. To illustrate basic file handling, we consider an example where we read x and y coordinates from two columns in a file, apply a function f to the y coordinates, and write the results to a new two-column data file. The first line of the input file is a heading that we can just skip:

```
# x and y coordinates
1.0  3.44
2.0  4.8
3.5  6.61
4.0  5.0
```

The relevant Python lines for reading the numbers and writing out a similar file are given in the file `file_handling.py`

```
filename = 'tmp.dat'
infile = open(filename, 'r')  # Open file for reading
line = infile.readline()      # Read first line
# Read x and y coordinates from the file and store in lists
x = []
y = []
for line in infile:
    words = line.split()      # Split line into words
    x.append(float(words[0]))
    y.append(float(words[1]))
infile.close()
```

```
# Transform y coordinates
from math import log

def f(y):
    return log(y)

for i in range(len(y)):
    y[i] = f(y[i])

# Write out x and y to a two-column file
filename = 'tmp_out.dat'
outfile = open(filename, 'w')  # Open file for writing
outfile.write('# x and y coordinates\n')
for xi, yi in zip(x, y):
    outfile.write('%10.5f %10.5f\n' % (xi, yi))
outfile.close()
```

Such a file with a comment line and numbers in tabular format is very common so numpy has functionality to ease reading and writing. Here is the same example using the loadtxt and savetxt functions in numpy for tabular data (file file_handling_numpy.py):

```
filename = 'tmp.dat'
import numpy
data = numpy.loadtxt(filename, comments='#')
x = data[:,0]
y = data[:,1]
data[:,1] = numpy.log(y)  # insert transformed y back in array
filename = 'tmp_out.dat'
filename = 'tmp_out.dat'
outfile = open(filename, 'w')  # open file for writing
outfile.write('# x and y coordinates\n')
numpy.savetxt(outfile, data, fmt='%10.5f')
```

2.7 Exercises

Exercise 2.1: Errors with colon, indent, etc.
Write the program ball_function.py as given in the text and confirm that the program runs correctly. Then save a copy of the program and use that program during the following error testing.

You are supposed to introduce errors in the code, one by one. For each error introduced, save and run the program, and comment how well Python's response corresponds to the actual error. When you are finished with one error, re-set the program to correct behavior (and check that it works!) before moving on to the next error.

a) Change the first line from `def y(t):` to `def y(t)`, i.e., remove the colon.
b) Remove the indent in front of the statement `v0 = 5` inside the function y, i.e., shift the text four spaces to the left.

c) Now let the statement `v0 = 5` inside the function `y` have an indent of three spaces (while the remaining two lines of the function have four).
d) Remove the left parenthesis in the first statement `def y(t):`
e) Change the first line of the function definition from `def y(t):` to `def y():`, i.e., remove the parameter `t`.
f) Change the first occurrence of the statement `print y(time)` to `print y()`.

Filename: `errors_colon_indent_etc.py`.

Exercise 2.2: Compare integers a and b
Explain briefly, in your own words, what the following program does.

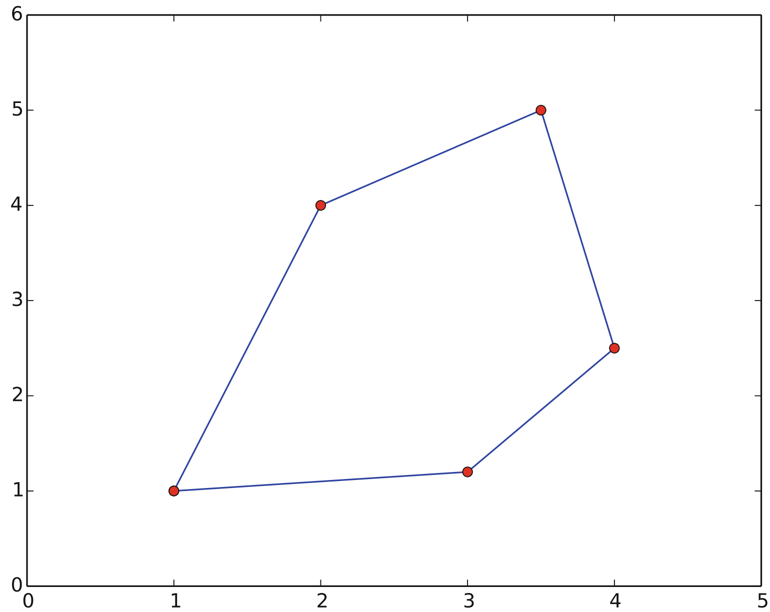
```
a = input('Give an integer a: ')
b = input('Give an integer b: ')

if a < b:
    print "a is the smallest of the two numbers"
elif a == b:
    print "a and b are equal"
else:
    print "a is the largest of the two numbers"
```

Proceed by writing the program, and then run it a few times with different values for `a` and `b` to confirm that it works as intended. In particular, choose combinations for `a` and `b` so that all three branches of the `if` construction get tested.
Filename: `compare_a_and_b.py`.

Exercise 2.3: Functions for circumference and area of a circle
Write a program that takes a circle radius `r` as input from the user and then computes the circumference `C` and area `A` of the circle. Implement the computations of `C` and `A` as two separate functions that each takes `r` as input parameter. Print `C` and `A` to the screen along with an appropriate text. Run the program with $r = 1$ and confirm that you get the right answer.
Filename: `functions_circumference_area.py`.

Exercise 2.4: Function for area of a rectangle
Write a program that computes the area $A = bc$ of a rectangle. The values of b and c should be user input to the program. Also, write the area computation as a function that takes b and c as input parameters and returns the computed area. Let the program print the result to screen along with an appropriate text. Run the program with $b = 2$ and $c = 3$ to confirm correct program behavior.
Filename: `function_area_rectangle.py`.

Exercise 2.5: Area of a polygon
One of the most important mathematical problems through all times has been to find the area of a polygon, especially because real estate areas often had the shape of polygons, and it was necessary to pay tax for the area. We have a polygon as depicted below.



The vertices ("corners") of the polygon have coordinates (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), numbered either in a clockwise or counter clockwise fashion. The area A of the polygon can amazingly be computed by just knowing the boundary coordinates:

$$A = \frac{1}{2} |(x_1 y_2 + x_2 y_3 + \cdots + x_{n-1} y_n + x_n y_1)$$
$$-(y_1 x_2 + y_2 x_3 + \cdots + y_{n-1} x_n + y_n x_1)| \ .$$

Write a function `polyarea(x, y)` that takes two coordinate arrays with the vertices as arguments and returns the area. Assume that `x` and `y` are either lists or arrays.

Test the function on a triangle, a quadrilateral, and a pentagon where you can calculate the area by alternative methods for comparison.

Hint Since Python lists and arrays has 0 as their first index, it is wise to rewrite the mathematical formula in terms of vertex coordinates numbered as $x_0, x_1, \ldots, x_{n-1}$ and $y_0, y_1, \ldots, y_{n-1}$.
Filename: `polyarea.py`.

Exercise 2.6: Average of integers
Write a program that gets an integer $N > 1$ from the user and computes the average of all integers $i = 1, \ldots, N$. The computation should be done in a function that takes N as input parameter. Print the result to the screen with an appropriate text. Run the program with $N = 5$ and confirm that you get the correct answer.
Filename: `average_1_to_N.py`.

Exercise 2.7: While loop with errors
Assume some program has been written for the task of adding all integers $i = 1, 2, \ldots, 10$:

```
some_number = 0
i = 1
while i < 11
    some_number += 1
print some_number
```

a) Identify the errors in the program by just reading the code and simulating the program by hand.
b) Write a new version of the program with errors corrected. Run this program and confirm that it gives the correct output.

Filename: `while_loop_errors.py`.

Exercise 2.8: Area of rectangle versus circle
Consider one circle and one rectangle. The circle has a radius $r = 10.6$. The rectangle has sides a and b, but only a is known from the outset. Let $a = 1.3$ and write a program that uses a `while` loop to find the largest possible integer b that gives a rectangle area smaller than, but as close as possible to, the area of the circle. Run the program and confirm that it gives the right answer (which is $b = 271$).
Filename: `area_rectangle_vs_circle.py`.

Exercise 2.9: Find crossing points of two graphs
Consider two functions $f(x) = x$ and $g(x) = x^2$ on the interval $[-4, 4]$.

Write a program that, by trial and error, finds approximately for which values of x the two graphs cross, i.e., $f(x) = g(x)$. Do this by considering N equally distributed points on the interval, at each point checking whether $|f(x)-g(x)| < \epsilon$, where ϵ is some small number. Let N and ϵ be user input to the program and let the result be printed to screen. Run your program with $N = 400$ and $\epsilon = 0.01$. Explain the output from the program. Finally, try also other values of N, keeping the value of ϵ fixed. Explain your observations.
Filename: `crossing_2_graphs.py`.

Exercise 2.10: Sort array with numbers
The import statement `from random import *` will give access to a function `uniform` that may be used to draw (pseudo-)random numbers from a uniform distribution between two numbers a (inclusive) and b (inclusive). For example, writing `x = uniform(0,10)` makes `x` a float value larger than, or equal to, 0, and smaller than, or equal to, 10.

Write a script that generates an array of 6 random numbers between 0 and 10. The program should then sort the array so that numbers appear in increasing order. Let the program make a formatted print of the array to the screen both before and after sorting. The printouts should appear on the screen so that comparison is made easy. Confirm that the array has been sorted correctly.
Filename: `sort_numbers.py`.

Exercise 2.11: Compute π
Up through history, great minds have developed different computational schemes for the number π. We will here consider two such schemes, one by Leibniz ($1646-1716$), and one by Euler ($1707-1783$).

The scheme by Leibniz may be written

$$\pi = 8\sum_{k=0}^{\infty} \frac{1}{(4k+1)(4k+3)},$$

while one form of the Euler scheme may appear as

$$\pi = \sqrt{6\sum_{k=1}^{\infty} \frac{1}{k^2}}.$$

If only the first N terms of each sum are used as an approximation to π, each modified scheme will have computed π with some error.

Write a program that takes N as input from the user, and plots the error development with both schemes as the number of iterations approaches N. Your program should also print out the final error achieved with both schemes, i.e. when the number of terms is `N`. Run the program with $N = 100$ and explain briefly what the graphs show.
Filename: `compute_pi.py`.

Exercise 2.12: Compute combinations of sets
Consider an ID number consisting of two letters and three digits, e.g., RE198. How many different numbers can we have, and how can a program generate all these combinations?

If a collection of n things can have m_1 variations of the first thing, m_2 of the second and so on, the total number of variations of the collection equals $m_1 m_2 \cdots m_n$. In particular, the ID number exemplified above can have $26{\cdot}26{\cdot}10{\cdot}10{\cdot}10 = 676,000$ variations. To generate all the combinations, we must have five nested for loops. The first two run over all letters A, B, and so on to Z, while the next three run over all digits $0, 1, \ldots, 9$.

To convince yourself about this result, start out with an ID number on the form A3 where the first part can vary among A, B, and C, and the digit can be among 1, 2, or 3. We must start with A and combine it with 1, 2, and 3, then continue with B, combined with 1, 2, and 3, and finally combine C with 1, 2, and 3. A double for loop does the work.

a) In a deck of cards, each card is a combination of a rank and a suit. There are 13 ranks: ace (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), queen (Q), king (K), and four suits: clubs (C), diamonds (D), hearts (H), and spades (S). A typical card may be D3. Write statements that generate a deck of cards, i.e., all the combinations `CA`, `C2`, `C3`, and so on to `SK`.
b) A vehicle registration number is on the form `DE562`, where the letters vary from A to Z and the digits from 0 to 9. Write statements that compute all the possible registration numbers and stores them in a list.

c) Generate all the combinations of throwing two dice (the number of eyes can vary from 1 to 6). Count how many combinations where the sum of the eyes equals 7.

Filename: `combine_sets.py`.

Exercise 2.13: Frequency of random numbers
Write a program that takes a positive integer N as input and then draws N random integers in the interval $[1, 6]$ (both ends inclusive). In the program, count how many of the numbers, M, that equal 6 and write out the fraction M/N. Also, print all the random numbers to the screen so that you can check for yourself that the counting is correct. Run the program with a small value for `N` (e.g., `N = 10`) to confirm that it works as intended.

Hint Use `random.randint(1,6)` to draw a random integer between 1 and 6.
Filename: `count_random_numbers.py`.

Remarks For large N, this program computes the probability M/N of getting six eyes when throwing a die.

Exercise 2.14: Game 21
Consider some game where each participant draws a series of random integers evenly distributed from 0 and 10, with the aim of getting the sum as close as possible to 21, but *not larger* than 21. You are out of the game if the sum passes 21. After each draw, you are told the number and your total sum, and are asked whether you want another draw or not. The one coming closest to 21 is the winner.

Implement this game in a program.

Hint Use `random.randint(0,10)` to draw random integers in $[0, 10]$.
Filename: `game_21.py`.

Exercise 2.15: Linear interpolation
Some measurements y_i, $i = 0, 1, \ldots, N$ (given below), of a quantity y have been collected regularly, once every minute, at times $t_i = i$, $i = 0, 1, \ldots, N$. We want to find the value y *in between* the measurements, e.g., at $t = 3.2$ min. Computing such y values is called *interpolation*.

Let your program use *linear interpolation* to compute y between two consecutive measurements:

1. Find i such that $t_i \leq t \leq t_{i+1}$.
2. Find a mathematical expression for the straight line that goes through the points (i, y_i) and $(i + 1, y_{i+1})$.
3. Compute the y value by inserting the user's time value in the expression for the straight line.

a) Implement the linear interpolation technique in a function that takes an array with the y_i measurements as input, together with some time t, and returns the interpolated y value at time t.

b) Write another function with a loop where the user is asked for a time on the interval $[0, N]$ and the corresponding (interpolated) y value is written to the screen. The loop is terminated when the user gives a negative time.
c) Use the following measurements: $4.4, 2.0, 11.0, 21.5, 7.5$, corresponding to times $0, 1, \ldots, 4$ (min), and compute interpolated values at $t = 2.5$ and $t = 3.1$ min. Perform separate hand calculations to check that the output from the program is correct.

Filename: `linear_interpolation.py`.

Exercise 2.16: Test straight line requirement
Assume the straight line function $f(x) = 4x + 1$. Write a script that tests the "point-slope" form for this line as follows. Within a chosen interval on the x-axis (for example, for x between 0 and 10), randomly pick 100 points on the line and check if the following requirement is fulfilled for each point:

$$\frac{f(x_i) - f(c)}{x_i - c} = a, \qquad i = 1, 2, \ldots, 100,$$

where a is the slope of the line and c defines a fixed point $(c, f(c))$ on the line. Let $c = 2$ here.
Filename: `test_straight_line.py`.

Exercise 2.17: Fit straight line to data
Assume some measurements $y_i, i = 1, 2, \ldots, 5$ have been collected, once every second. Your task is to write a program that fits a straight line to those data.

a) Make a function that computes the error between the straight line $f(x) = ax+b$ and the measurements:

$$e = \sum_{i=1}^{5} (ax_i + b - y_i)^2 .$$

b) Make a function with a loop where you give a and b, the corresponding value of e is written to the screen, and a plot of the straight line $f(x) = ax + b$ together with the discrete measurements is shown.

Hint To make the plotting from the loop to work, you may have to insert `from matplotlib.pylab import *` at the top of the script and also add `show()` after the plot command in the loop.

c) Given the measurements $0.5, 2.0, 1.0, 1.5, 7.5$, at times $0, 1, 2, 3, 4$, use the function in b) to interactively search for a and b such that e is minimized.

Filename: `fit_straight_line.py`.

Remarks Fitting a straight line to measured data points is a very common task. The manual search procedure in c) can be automated by using a mathematical method called the *method of least squares*.

Exercise 2.18: Fit sines to straight line
A lot of technology, especially most types of digital audio devices for processing sound, is based on representing a signal of time as a sum of sine functions. Say the signal is some function $f(t)$ on the interval $[-\pi, \pi]$ (a more general interval $[a, b]$ can easily be treated, but leads to slightly more complicated formulas). Instead of working with $f(t)$ directly, we approximate f by the sum

$$S_N(t) = \sum_{n=1}^{N} b_n \sin(nt), \tag{2.1}$$

where the coefficients b_n must be adjusted such that $S_N(t)$ is a good approximation to $f(t)$. We shall in this exercise adjust b_n by a trial-and-error process.

a) Make a function `sinesum(t, b)` that returns $S_N(t)$, given the coefficients b_n in an array `b` and time coordinates in an array `t`. Note that if `t` is an array, the return value is also an array.
b) Write a function `test_sinesum()` that calls `sinesum(t, b)` in a) and determines if the function computes a test case correctly. As test case, let `t` be an array with values $-\pi/2$ and $\pi/4$, choose $N = 2$, and $b_1 = 4$ and $b_2 = -3$. Compute $S_N(t)$ by hand to get reference values.
c) Make a function `plot_compare(f, N, M)` that plots the original function $f(t)$ together with the sum of sines $S_N(t)$, so that the quality of the approximation $S_N(t)$ can be examined visually. The argument `f` is a Python function implementing $f(t)$, `N` is the number of terms in the sum $S_N(t)$, and `M` is the number of uniformly distributed t coordinates used to plot f and S_N.
d) Write a function `error(b, f, M)` that returns a mathematical measure of the error in $S_N(t)$ as an approximation to $f(t)$:

$$E = \sqrt{\sum_i (f(t_i) - S_N(t_i))^2},$$

where the t_i values are M uniformly distributed coordinates on $[-\pi, \pi]$. The array `b` holds the coefficients in S_N and `f` is a Python function implementing the mathematical function $f(t)$.
e) Make a function `trial(f, N)` for interactively giving b_n values and getting a plot on the screen where the resulting $S_N(t)$ is plotted together with $f(t)$. The error in the approximation should also be computed as indicated in d). The argument `f` is a Python function for $f(t)$ and `N` is the number of terms N in the sum $S_N(t)$. The `trial` function can run a loop where the user is asked for the b_n values in each pass of the loop and the corresponding plot is shown. You must find a way to terminate the loop when the experiments are over. Use `M=500` in the calls to `plot_compare` and `error`.

Hint To make this part of your program work, you may have to insert `from matplotlib.pylab import *` at the top and also add `show()` after the plot command in the loop.

f) Choose $f(t)$ to be a straight line $f(t) = \frac{1}{\pi}t$ on $[-\pi, \pi]$. Call `trial(f, 3)` and try to find through experimentation some values b_1, b_2, and b_3 such that the sum of sines $S_N(t)$ is a good approximation to the straight line.
g) Now we shall try to automate the procedure in f). Write a function that has three nested loops over values of b_1, b_2, and b_3. Let each loop cover the interval $[-1, 1]$ in steps of 0.1. For each combination of b_1, b_2, and b_3, the error in the approximation S_N should be computed. Use this to find, and print, the smallest error and the corresponding values of b_1, b_2, and b_3. Let the program also plot f and the approximation S_N corresponding to the smallest error.

Filename: `fit_sines.py`.

Remarks

1. The function $S_N(x)$ is a special case of what is called a *Fourier series*. At the beginning of the 19th century, Joseph Fourier (1768-1830) showed that any function can be approximated analytically by a sum of cosines and sines. The approximation improves as the number of terms (N) is increased. Fourier series are very important throughout science and engineering today.
   (a) Finding the coefficients b_n is solved much more accurately in Exercise 3.12, by a procedure that also requires much less human and computer work!
   (b) In real applications, $f(t)$ is not known as a continuous function, but function values of $f(t)$ are provided. For example, in digital sound applications, music in a CD-quality WAV file is a signal with 44,100 samples of the corresponding analog signal $f(t)$ per second.

Exercise 2.19: Count occurrences of a string in a string
In the analysis of genes one encounters many problem settings involving searching for certain combinations of letters in a long string. For example, we may have a string like

```
gene = 'AGTCAATGGAATAGGCCAAGCGAATATTTGGGCTACCA'
```

We may traverse this string, letter by letter, by the for loop `for letter in gene`. The length of the string is given by `len(gene)`, so an alternative traversal over an index `i` is `for i in range(len(gene))`. Letter number `i` is reached through `gene[i]`, and a substring from index `i` up to, but not including `j`, is created by `gene[i:j]`.

a) Write a function `freq(letter, text)` that returns the frequency of the letter `letter` in the string `text`, i.e., the number of occurrences of `letter` divided by the length of `text`. Call the function to determine the frequency of `C` and `G` in the `gene` string above. Compute the frequency by hand too.
b) Write a function `pairs(letter, text)` that counts how many times a pair of the letter `letter` (e.g., `GG`) occurs within the string `text`. Use the function to determine how many times the pair `AA` appears in the string `gene` above. Perform a manual counting too to check the answer.

c) Write a function `mystruct(text)` that counts the number of a certain structure in the string `text`. The structure is defined as `G` followed by `A` or `T` until a double `GG`. Perform a manual search for the structure too to control the computations by `mystruct`.

Filename: `count_substrings.py`.

Remarks You are supposed to solve the tasks using simple programming with loops and variables. While a) and b) are quite straightforward, c) quickly involves demanding logic. However, there are powerful tools available in Python that can solve the tasks efficiently in very compact code: a) `text.count(letter)/float(len(text))`; b) `text.count(letter*2)`; c) `len(re.findall('G[AT]+?GG', text))`. That is, there is rich functionality for analysis of text in Python and this is particularly useful in analysis of gene sequences.