```
Terminal
ball_yc.py
At t=0.0417064 s and 0.977662 s, the height is 0.2 m.
```

Recall from Section 1.5.3 that we just write the program name. A real execution demands prefixing the program name by `python` in a terminal window, or by `run` if you run the program from an interactive IPython session. We refer to Appendix H.2 for more complete information on running Python programs in different ways.

Sometimes just the output from a program is shown, and this output appears as plain computer text:

```
h = 0.2
order=0, error=0.221403
order=1, error=0.0214028
order=2, error=0.00140276
order=3, error=6.94248e-05
order=4, error=2.75816e-06
```

Files containing data are shown in a similar way in this book:

```
date   Oslo   London   Berlin   Paris   Rome   Helsinki
01.05  18     21.2     20.2     13.7    15.8   15
01.06  21     13.2     14.9     18      24     20
01.07  13     14       16       25      26.2   14.5
```

Style guide for Python code. This book presents Python code that is (mostly) in accordance with the official Style Guide for Python Code[5], known in the Python community as *PEP8*. Some exceptions to the rules are made to make code snippets shorter: multiple imports on one line and less blank lines.

1.9 Exercises

What does it mean to solve an exercise? The solution to most of the exercises in this book is a Python program. To produce the solution, you first need understand the problem and what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements. Equally important is the verification (testing) of the program. A complete solution to a programming exercises therefore consists of two parts: 1) the program text and 2) a demonstration that the program works correctly. Some simple programs, like the ones in the first two exercises below, have so obviously correct output that the verification can just be to run the program and record the output.

In cases where the correctness of the output is not obvious, it is necessary to prove or bring evidence that the result is correct. This can be done through comparisons with calculations done separately on a

[5] `http://www.python.org/dev/peps/pep-0008/`

calculator, or one can apply the program to a special simple test case with known results. The requirement is to provide evidence to the claim that the program is without programming errors.

The sample run of the program to check its correctness can be inserted at the end of the program as a triple-quoted string. Alternatively, the output lines can be inserted as comments, but using a multi-line string requires less typing. (Technically, a string object is created, but not assigned to any name or used for anything in the program beyond providing useful information for the reader of the code.) One can do

Terminal

```
Terminal> python myprogram.py > result
```

and use a text editor to insert the file `result` inside the triple-quoted multi-line string. Here is an example on a run of a Fahrenheit to Celsius conversion program inserted at the end as a triple-quoted string:

```
F = 69.8                # Fahrenheit degrees
C = (5.0/9)*(F - 32)    # Corresponding Celsius degrees
print C

'''
Sample run (correct result is 21):
python f2c.py
21.0
'''
```

Exercise 1.1: Compute 1+1

The first exercise concerns some very basic mathematics and programming: assign the result of 1+1 to a variable and print the value of that variable. Filename: `1plus1.py`.

Exercise 1.2: Write a Hello World program

Almost all books about programming languages start with a very simple program that prints the text `Hello, World!` to the screen. Make such a program in Python. Filename: `hello_world.py`.

Exercise 1.3: Derive and compute a formula

Can a newborn baby in Norway expect to live for one billion (10^9) seconds? Write a Python program for doing arithmetics to answer the question. Filename: `seconds2years.py`.

Exercise 1.4: Convert from meters to British length units

Make a program where you set a length given in meters and then compute and write out the corresponding length measured in inches, in feet, in yards, and in miles. Use that one inch is 2.54 cm, one foot is 12 inches, one yard is 3 feet, and one British mile is 1760 yards. For verification, a length of 640 meters corresponds to 25196.85 inches, 2099.74 feet, 699.91 yards, or 0.3977 miles. Filename: `length_conversion.py`.

Exercise 1.5: Compute the mass of various substances

The density of a substance is defined as $\varrho = m/V$, where m is the mass of a volume V. Compute and print out the mass of one liter of each of the following substances whose densities in g/cm^3 are found in the file `src/files/densities.dat`[6]: iron, air, gasoline, ice, the human body, silver, and platinum. Filename: `1liter.py`.

Exercise 1.6: Compute the growth of money in a bank

Let p be a bank's interest rate in percent per year. An initial amount A has then grown to

$$A\left(1+\frac{p}{100}\right)^n$$

after n years. Make a program for computing how much money 1000 euros have grown to after three years with 5 percent interest rate. Filename: `interest_rate.py`.

Exercise 1.7: Find error(s) in a program

Suppose somebody has written a simple one-line program for computing sin(1):

```
x=1; print 'sin(%g)=%g' % (x, sin(x))
```

Create this program and try to run it. What is the problem?

Exercise 1.8: Type in program text

Type the following program in your editor and execute it. If your program does not work, check that you have copied the code correctly.

[6] `http://tinyurl.com/pwyasaa/files/densities.dat`

```
from math import pi

h = 5.0   # height
b = 2.0   # base
r = 1.5   # radius

area_parallelogram = h*b
print 'The area of the parallelogram is %.3f' % area_parallelogram

area_square = b**2
print 'The area of the square is %g' % area_square

area_circle = pi*r**2
print 'The area of the circle is %.3f' % area_circle

volume_cone = 1.0/3*pi*r**2*h
print 'The volume of the cone is %.3f' % volume_cone
```

Filename: `formulas_shapes.py`.

Exercise 1.9: Type in programs and debug them

Type these short programs in your editor and execute them. When they do not work, identify and correct the erroneous statements.

a) Does $\sin^2(x) + \cos^2(x) = 1$?

```
from math import sin, cos
x = pi/4
1_val = math.sin^2(x) + math.cos^2(x)
print 1_VAL
```

b) Compute s in meters when $s = v_0t + \frac{1}{2}at^2$, with $v_0 = 3$ m/s, $t = 1$ s, $a = 2$ m/s^2.

```
v0 = 3 m/s
t = 1 s
a = 2 m/s**2
s = v0.t + 0,5.a.t**2
print s
```

c) Verify these equations:

$$(a + b)^2 = a^2 + 2ab + b^2$$

$$(a - b)^2 = a^2 - 2ab + b^2$$

```
a = 3,3   b = 5,3
a2 = a**2
b2 = b**2

eq1_sum = a2 + 2ab + b2
eq2_sum = a2 - 2ab + b2

eq1_pow = (a + b)**2
```

```
eq2_pow = (a - b)**2

print 'First equation:  %g = %g', % (eq1_sum, eq1_pow)
print 'Second equation: %h = %h', % (eq2_pow, eq2_pow)
```

Filename: `sin2_plus_cos2.py`.

Exercise 1.10: Evaluate a Gaussian function

The bell-shaped Gaussian function,

$$f(x) = \frac{1}{\sqrt{2\pi}\, s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right], \tag{1.7}$$

is one of the most widely used functions in science and technology. The parameters m and $s > 0$ are prescribed real numbers. Make a program for evaluating this function when $m = 0$, $s = 2$, and $x = 1$. Verify the program's result by comparing with hand calculations on a calculator. Filename: `gaussian1.py`.

Remarks. The function (1.7) is named after Carl Friedrich Gauss[7], 1777-1855, who was a German mathematician and scientist, now considered as one of the greatest scientists of all time. He contributed to many fields, including number theory, statistics, mathematical analysis, differential geometry, geodesy, electrostatics, astronomy, and optics. Gauss introduced the function (1.7) when he analyzed probabilities related to astronomical data.

Exercise 1.11: Compute the air resistance on a football

The drag force, due to air resistance, on an object can be expressed as

$$F_d = \frac{1}{2} C_D \varrho A V^2, \tag{1.8}$$

where ϱ is the density of the air, V is the velocity of the object, A is the cross-sectional area (normal to the velocity direction), and C_D is the drag coefficient, which depends heavily on the shape of the object and the roughness of the surface.

The gravity force on an object with mass m is $F_g = mg$, where $g = 9.81 \mathrm{m\,s}^{-2}$.

We can use the formulas for F_d and F_g to study the importance of air resistance versus gravity when kicking a football. The density of air is $\varrho = 1.2$ kg m^{-3}. We have $A = \pi a^2$ for any ball with radius a. For a football, $a = 11$ cm, the mass is 0.43 kg, and C_D can be taken as 0.2.

[7] `http://en.wikipedia.org/wiki/Carl_Gauss`

Make a program that computes the drag force and the gravity force on a football. Write out the forces with one decimal in units of Newton ($\mathrm{N} = \mathrm{kg\,m/s^2}$). Also print the ratio of the drag force and the gravity force. Define C_D, ϱ, A, V, m, g, F_d, and F_g as variables, and put a comment with the corresponding unit. Use the program to calculate the forces on the ball for a hard kick, $V = 120$ km/h and for a soft kick, $V = 10$ km/h (it is easy to mix inconsistent units, so make sure you compute with V expressed in m/s). Filename: `kick.py`.

Exercise 1.12: How to cook the perfect egg

As an egg cooks, the proteins first denature and then coagulate. When the temperature exceeds a critical point, reactions begin and proceed faster as the temperature increases. In the egg white, the proteins start to coagulate for temperatures above 63 C, while in the yolk the proteins start to coagulate for temperatures above 70 C. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 C, but the yolk should not be heated above 70 C. For a hard boiled egg, the center of the yolk should be allowed to reach 70 C.

The following formula expresses the time t it takes (in seconds) for the center of the yolk to reach the temperature T_y (in Celsius degrees):

$$t = \frac{M^{2/3} c \rho^{1/3}}{K\pi^2 (4\pi/3)^{2/3}} \ln\left[0.76 \frac{T_o - T_w}{T_y - T_w}\right] . \tag{1.9}$$

Here, M, ρ, c, and K are properties of the egg: M is the mass, ρ is the density, c is the specific heat capacity, and K is thermal conductivity. Relevant values are $M = 47$ g for a small egg and $M = 67$ g for a large egg, $\rho = 1.038\ \mathrm{g\,cm^{-3}}$, $c = 3.7\ \mathrm{J\,g^{-1}\,K^{-1}}$, and $K = 5.4 \cdot 10^{-3}\ \mathrm{W\,cm^{-1}\,K^{-1}}$. Furthermore, T_w is the temperature (in C degrees) of the boiling water, and T_o is the original temperature (in C degrees) of the egg before being put in the water. Implement the formula in a program, set $T_w = 100$ C and $T_y = 70$ C, and compute t for a large egg taken from the fridge ($T_o = 4$ C) and from room temperature ($T_o = 20$ C). Filename: `egg.py`.

Exercise 1.13: Derive the trajectory of a ball

The purpose of this exercise is to explain how Equation (1.6) for the trajectory of a ball arises from basic physics. There is no programming in this exercise, just physics and mathematics.

The motion of the ball is governed by Newton's second law:

$$F_x = ma_x \tag{1.10}$$
$$F_y = ma_y \tag{1.11}$$

where F_x and F_y are the sum of forces in the x and y directions, respectively, a_x and a_y are the accelerations of the ball in the x and y directions, and m is the mass of the ball. Let $(x(t), y(t))$ be the position of the ball, i.e., the horizontal and vertical coordinate of the ball at time t. There are well-known relations between acceleration, velocity, and position: the acceleration is the time derivative of the velocity, and the velocity is the time derivative of the position. Therefore we have that

$$a_x = \frac{d^2x}{dt^2}, \tag{1.12}$$
$$a_y = \frac{d^2y}{dt^2}\,. \tag{1.13}$$

If we assume that gravity is the only important force on the ball, $F_x = 0$ and $F_y = -mg$.

Integrate the two components of Newton's second law twice. Use the initial conditions on velocity and position,

$$\frac{d}{dt}x(0) = v_0 \cos\theta, \tag{1.14}$$
$$\frac{d}{dt}y(0) = v_0 \sin\theta, \tag{1.15}$$
$$x(0) = 0, \tag{1.16}$$
$$y(0) = y_0, \tag{1.17}$$

to determine the four integration constants. Write up the final expressions for $x(t)$ and $y(t)$. Show that if $\theta = \pi/2$, i.e., the motion is purely vertical, we get the formula (1.1) for the y position. Also show that if we eliminate t, we end up with the relation (1.6) between the x and y coordinates of the ball. You may read more about this type of motion in a physics book, e.g., [16]. Filename: `trajectory.*`.

Exercise 1.14: Find errors in the coding of formulas

Some versions of our program for calculating the formula (1.3) are listed below. Find the versions that will not work correctly and explain why in each case.

```
C = 21;    F =  9/5*C + 32;         print F
C = 21.0;  F =  (9/5)*C + 32;       print F
C = 21.0;  F =  9*C/5 + 32;         print F
C = 21.0;  F =  9.*(C/5.0) + 32;    print F
C = 21.0;  F =  9.0*C/5.0 + 32;     print F
```

```
C = 21;     F =  9*C/5 + 32;        print F
C = 21.0;   F =  (1/5)*9*C + 32;    print F
C = 21;     F =  (1./5)*9*C + 32;   print F
```

Exercise 1.15: Explain why a program does not work

Find out why the following program does not work:

```
C = A + B
A = 3
B = 2
print C
```

Exercise 1.16: Find errors in Python statements

Try the following statements in an interactive Python shell. Explain why some statements fail and correct the errors.

```
1a = 2
a1 = b
x = 2
y = X + 4  # is it 6?
from Math import tan
print tan(pi)
pi = "3.14159’
print tan(pi)
c = 4**3**2**3
_ = ((c-78564)/c + 32))
discount = 12%
AMOUNT = 120.-
amount = 120$
address = hpl@simula.no
and = duck
class = ’INF1100, gr 2"
continue_ = x > 0
rev = fox = True
Norwegian = [’a human language’]
true = fox is rev in Norwegian
```

Hint. It is wise to test the values of the expressions on the right-hand side, and the validity of the variable names, separately before you put the left- and right-hand sides together in statements. The last two statements work, but explaining why goes beyond what is treated in this chapter.

Exercise 1.17: Find errors in the coding of a formula

Given a quadratic equation,

$$ax^2 + bx + c = 0,$$

the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,. \qquad (1.18)$$

What are the problems with the following program?

```
a = 2; b = 1; c = 2
from math import sqrt
q = b*b - 4*a*c
q_sr = sqrt(q)
x1 = (-b + q_sr)/2*a
x2 = (-b - q_sr)/2*a
print x1, x2
```

Correct the program so that it solves the given equation. Filename: `find_errors_roots.py`.

Quick references, which list almost to all Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet[6] [6].

The website `http://www.python.org/doc/` contains a list of useful Python introductions and reference manuals.

2.7 Exercises

Exercise 2.1: Make a Fahrenheit-Celsius conversion table

Write a Python program that prints out a table with Fahrenheit degrees $0, 10, 20, \ldots, 100$ in the first column and the corresponding Celsius degrees in the second column.

Hint. Modify the `c2f_table_while.py` program from Section 2.1.2. Filename: `f2c_table_while.py`.

Exercise 2.2: Generate an approximate Fahrenheit-Celsius conversion table

Many people use an approximate formula for quickly converting Fahrenheit (F) to Celsius (C) degrees:

$$C \approx \hat{C} = (F - 30)/2 \tag{2.2}$$

Modify the program from Exercise 2.1 so that it prints three columns: F, C, and the approximate value $\hat{C}$. Filename: `f2c_approx_table.py`.

Exercise 2.3: Work with a list

Set a variable `primes` to a list containing the numbers 2, 3, 5, 7, 11, and 13. Write out each list element in a `for` loop. Assign 17 to a variable `p` and add `p` to the end of the list. Print out the entire new list. Filename: `primes.py`.

Exercise 2.4: Generate odd numbers

Write a program that generates all odd numbers from 1 to `n`. Set `n` in the beginning of the program and use a `while` loop to compute the numbers. (Make sure that if `n` is an even number, the largest generated odd number is `n-1`.) Filename: `odd.py`.

[6] `http://rgruet.free.fr/PQR27/PQR2.7.html`

Exercise 2.5: Sum of first n integers

Write a program that computes the sum of the integers from 1 up to and including n. Compare the result with the famous formula $n(n+1)/2$.
Filename: `sum_int.py`.

Exercise 2.6: Generate equally spaced coordinates

We want to generate $n+1$ equally spaced x coordinates in $[a,b]$. Store the coordinates in a list.

a) Start with an empty list, use a `for` loop and append each coordinate to the list.

Hint. With n intervals, corresponding to $n+1$ points, in $[a,b]$, each interval has length $h=(b-a)/n$. The coordinates can then be generated by the formula $x_i = a + ih$, $i = 0, \ldots, n+1$.

b) Use a list comprehension (see Section 2.3.5).
Filename: `coor.py`.

Exercise 2.7: Make a table of values from a formula

The purpose of this exercise is to write code that prints a nicely formatted table of t and $y(t)$ values, where

$$y(t) = v_0 t - \frac{1}{2} g t^2 \,.$$

Use $n+1$ uniformly spaced t values throughout the interval $[0, 2v_0/g]$.

a) Use a `for` loop to produce the table.

b) Add code with a `while` loop to produce the table.

Hint. Because of potential round-off errors, you may need to adjust the upper limit of the `while` loop to ensure that the last point ($t = 2v_0/g$, $y = 0$) is included.
Filename: `ball_table1.py`.

Exercise 2.8: Store values from a formula in lists

This exercise aims to produce the same table of numbers as in Exercise 2.7, but with different code. First, store the t and y values in two lists `t` and `y`. Thereafter, write out a nicely formatted table by traversing the two lists with a `for` loop.

Hint. In the `for` loop, use either `zip` to traverse the two lists in parallel, or use an index and the `range` construction.
Filename: `ball_table2.py`.

Exercise 2.9: Simulate operations on lists by hand

You are given the following program:

```
a = [1, 3, 5, 7, 11]
b = [13, 17]
c = a + b
print c
b[0] = -1
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d[-2:]
for e1 in a:
    for e2 in b:
        print e1 + e2
```

Go through each statement and explain what is printed by the program.

Exercise 2.10: Compute a mathematical sum

The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{k}$:

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

This program does not work correctly. What are the three errors? (If you try to run the program, nothing will happen on the screen. Type `Ctrl+c`, i.e., hold down the Control (`Ctrl`) key and then type the `c` key, to stop the program.) Write a correct program.

Hint. There are two basic ways to find errors in a program:

1. read the program carefully and think about the consequences of each statement,
2. print out intermediate results and compare with hand calculations.

First, try method 1 and find as many errors as you can. Thereafter, try method 2 for $M = 3$ and compare the evolution of `s` with your own hand calculations.
Filename: `sum_while.py`.

Exercise 2.11: Replace a while loop by a for loop

Rewrite the corrected version of the program in Exercise 2.10 using a `for` loop over `k` values instead of a `while` loop. Filename: `sum_for.py`.

Exercise 2.12: Simulate a program by hand

Consider the following program for computing with interest rates:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

a) Use a pocket calculator or an interactive Python shell and work through the program calculations by hand. Write down the value of `amount` and `years` in each pass of the loop.

b) Set `p = 5` instead. Why will the loop now run forever? (Apply Ctrl+c, see Exercise 2.10, to stop a program with a loop that runs forever.) Make the program robust against such errors.

c) Make use of the operator `+=` wherever possible in the program.

d) Explain with words what type of mathematical problem that is solved by this program.
Filename: `interest_rate_loop.py`.

Exercise 2.13: Explore Python documentation

Suppose you want to compute with the inverse sine function: $\sin^{-1} x$. How do you do that in a Python program?

Hint. The `math` module has an inverse sine function. Find the correct name of the function by looking up the module content in the online Python Standard Library[7] document or use `pydoc`, see Section 2.6.3.
Filename: `inverse_sine.py`.

Exercise 2.14: Index a nested lists

We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

a) Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`.

b) We can visit all elements of `q` using this nested `for` loop:

[7] `http://docs.python.org/2/library/`

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`?
Filename: `index_nested_list.py`.

Exercise 2.15: Store data in lists

Modify the program from Exercise 2.2 so that all the F, C, and $\hat{C}$ values are stored in separate lists `F`, `C`, and `C_approx`, respectively. Then make a nested list `conversion` so that `conversion[i]` holds a row in the table: `[F[i], C[i], C_approx[i]]`. Finally, let the program traverse the `conversion` list and write out the same table as in Exercise 2.2.
Filename: `f2c_approx_lists.py`.

Exercise 2.16: Store data in a nested list

a) Compute two lists of t and y values as explained in Exercise 2.8. Store the two lists in a new nested list `ty1` such that `ty1[0]` and `ty1[1]` correspond to the two lists. Write out a table with t and y values in two columns by looping over the data in the `ty1` list. Each number should be written with two decimals.

b) Make a list `ty2` which holds each row in the table of t and y values (`ty1` is a list of table columns while `ty2` is a list of table rows, as explained in Section 2.4). Loop over the `ty2` list and write out the t and y values with two decimals each.
Filename: `ball_table3.py`.

Exercise 2.17: Values of boolean expressions

Explain the outcome of each of the following boolean expressions:

```
C = 41
C == 40
C != 40 and C < 41
C != 40 or  C < 41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Note

It makes sense to compare `True` and `False` to the integers 0 and 1, but not other integers (e.g., `True == 12` is `False` although the *integer* 12 evaluates to `True` in a boolean context, as in `bool(12)` or `if 12`).

Exercise 2.18: Explore round-off errors from a large number of inverse operations

Maybe you have tried to hit the square root key on a calculator multiple times and then squared the number again an equal number of times. These set of inverse mathematical operations should of course bring you back to the starting value for the computations, but this does not always happen. To avoid tedious pressing of calculator keys, we can let a computer automate the process. Here is an appropriate program:

```
from math import sqrt
for n in range(1, 60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print '%d times sqrt and **2: %.16f' % (n, r)
```

Explain with words what the program does. Then run the program. Round-off errors are here completely destroying the calculations when `n` is large enough! Investigate the case when we come back to 1 instead of 2 by fixing an `n` value where this happens and printing out `r` in both `for` loops over `i`. Can you now explain why we come back to 1 and not 2? Filename: `repeated_sqrt.py`.

Exercise 2.19: Explore what zero can be on a computer

Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print '...............', eps
    eps = eps/2.0
print 'final eps:', eps
```

Explain with words what the code is doing, line by line. Then examine the output. How can it be that the "equation" $1 \neq 1 + \text{eps}$ is not true? Or in other words, that a number of approximately size 10^{-16} (the final `eps` value when the loop terminates) gives the same result as if `eps` were zero? Filename: `machine_zero.py`.

Remarks. The nonzero `eps` value computed above is called *machine epsilon* or *machine zero* and is an important parameter to know, especially when certain mathematical techniques are applied to control round-off errors.

Exercise 2.20: Compare two real numbers with a tolerance

Run the following program:

```
a = 1/947.0*947
b = 1
if a != b:
    print 'Wrong result!'
```

The lesson learned from this program is that one should never compare two floating-point objects directly using `a == b` or `a != b`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if `abs(a - b) < tol`, where `tol` is a very small number. Modify the test according to this idea. Filename: `compare_floats.py`.

Exercise 2.21: Interpret a code

The function `time` in the module `time` returns the number of seconds since a particular date (called the Epoch, which is January 1, 1970, on many types of computers). Python programs can therefore use `time.time()` to mimic a stop watch. Another function, `time.sleep(n)` causes the program to pause for `n` seconds and is handy for inserting a pause. Use this information to explain what the following code does:

```
import time
t0 = time.time()
while time.time() - t0 < 10:
    print '....I like while loops!'
    time.sleep(2)
print 'Oh, no - the loop is over.'
```

How many times is the `print` statement inside the loop executed? Now, copy the code segment and change the `<` sign in the loop condition to a `>` sign. Explain what happens now. Filename: `time_while.py`.

Exercise 2.22: Explore problems with inaccurate indentation

Type in the following program in a file and check carefully that you have exactly the same spaces:

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
        print C, F
C = C + dC
```

Run the program. What is the first problem? Correct that error. What is the next problem? What is the cause of that problem? (See Exercise 2.10 for how to stop a hanging program.) Filename: `indentation.py`.

Remarks. The lesson learned from this exercise is that one has to be very careful with indentation in Python programs! Other computer languages usually enclose blocks belonging to loops in curly braces, parentheses, or begin-end marks. Python's convention with using solely indentation contributes to visually attractive, easy-to-read code, at the cost of requiring a pedantic attitude to blanks from the programmer.

Exercise 2.23: Explore punctuation in Python programs

Some of the following assignments work and some do not. Explain in each case why the assignment works/fails and, if it works, what kind of object `x` refers to and what the value is if we do a `print x`.

```
x = 1
x = 1.
x = 1;
x = 1!
x = 1?
x = 1:
x = 1,
```

Hint. Explore the statements in an interactive Python shell.
Filename: `punctuation.*`.

Exercise 2.24: Investigate a for loop over a changing list

Study the following interactive session and explain in detail what happens in each pass of the loop, and use this explanation to understand the output.

```
>>> numbers = range(10)
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for n in numbers:
...     i = len(numbers)/2
...     del numbers[i]
...     print 'n=%d, del %d' % (n,i), numbers
...
n=0, del 5 [0, 1, 2, 3, 4, 6, 7, 8, 9]
n=1, del 4 [0, 1, 2, 3, 6, 7, 8, 9]
n=2, del 4 [0, 1, 2, 3, 7, 8, 9]
n=3, del 3 [0, 1, 2, 7, 8, 9]
n=8, del 3 [0, 1, 2, 8, 9]
```

Warning

The message in this exercise is to *never modify a list that we are looping over.* Modification is indeed technically possible, as shown above, but you really need to know what you are doing. Otherwise you will experience very strange program behavior.

```
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of f
    from a to b using Simpson's rule with n intervals.
    """
    if a > b:
        print 'Error: a=%g > b=%g' % (a, b)
        return None

    # check that n is even:
    if n % 2 != 0:
        print 'Error: n=%d is not an even integer!' % n
        n = n+1  # make n even

    h = (b - a)/float(n)
    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

The complete code is found in the file `Simpson.py`.

A very good exercise is to simulate the program flow by hand, starting with the call to the `application` function. The Online Python Tutor or a debugger (see Section F.1) are convenient tools for controlling that your thinking is correct.

3.5 Exercises

Exercise 3.1: Write a Fahrenheit-Celsius conversion function

The formula for converting Fahrenheit degrees to Celsius reads

$$C = \frac{5}{9}(F - 32)\,. \tag{3.7}$$

Write a function `C(F)` that implements this formula. To verify the implementation, you can use `F(C)` from Section 3.1.1 and test that `C(F(c)` equals `c`.

Hint. Do not test `C(F(c)) == c` exactly, but use a tolerance for the difference.
Filename: `f2c.py`.

Exercise 3.2: Evaluate a sum and write a test function

a) Write a Python function `sum_1k(M)` that returns the sum $s = \sum_{k=1}^{M} \frac{1}{k}$.

b) Compute s for $M = 3$ by hand and write another function `test_sum_1k()` that calls `sum_1k(3)` and checks that the answer is correct.

Hint. We recommend that `test_sum_1k` follows the conventions of the pytest and nose testing frameworks as explained in Sections 3.3.3 and 3.4.2 (see also Section H.6). It means setting a boolean variable `success` to `True` if the test passes, otherwise `False`. The next step is to do `assert success`, which will abort the program with an error message if `success` is `False` and the test failed. To provide an informative error message, you can add your own message string `msg`: `assert success, msg`.
Filename: `sum_func.py`.

Exercise 3.3: Write a function for solving $ax^2 + bx + c = 0$

a) Given a quadratic equation $ax^2+bx+c = 0$, write a function `roots(a, b, c)` that returns the two roots of the equation. The returned roots should be `float` objects when the roots are real, otherwise the function returns `complex` objects.

Hint. Use `sqrt` from the `numpy.lib.scimath` library, see Chapter 1.6.3.

b) Construct two test cases with known solutions, one with real roots and the other with complex roots, Implement the two test cases in two test functions `test_roots_float` and `test_roots_complex`, where you call the `roots` function and check the type and value of the returned objects.
Filename: `roots_quadratic.py`.

Exercise 3.4: Implement the sum function

The standard Python function called `sum` takes a list as argument and computes the sum of the elements in the list:

```
>>> sum([1,3,5,-5])
4
>>> sum([[1,2], [4,3], [8,1]])
[1, 2, 4, 3, 8, 1]
>>> sum(['Hello, ', 'World!'])
'Hello, World!'
```

Implement your own version of `sum`. Filename: `mysum.py`.

Exercise 3.5: Compute a polynomial via a product

Given $n+1$ roots $r_0, r_1, \ldots, r_n$ of a polynomial $p(x)$ of degree $n+1$, $p(x)$ can be computed by

$$p(x) = \prod_{i=0}^{n}(x - r_i) = (x - r_0)(x - r_1)\cdots(x - r_{n-1})(x - r_n)\,. \qquad (3.8)$$

Write a function `poly(x, roots)` that takes x and a list `roots` of the roots as arguments and returns $p(x)$. Construct a test case for verifying the implementation. Filename: `polyprod.py`.

Exercise 3.6: Integrate a function by the Trapezoidal rule

a) An approximation to the integral of a function $f(x)$ over an interval $[a, b]$ can be found by first approximating $f(x)$ by the straight line that goes through the end points $(a, f(a))$ and $(b, f(b))$, and then finding the area under the straight line, which is the area of a trapezoid. The resulting formula becomes

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a) + f(b))\,. \qquad (3.9)$$

Write a function `trapezint1(f, a, b)` that returns this approximation to the integral. The argument `f` is a Python implementation `f(x)` of the mathematical function $f(x)$.

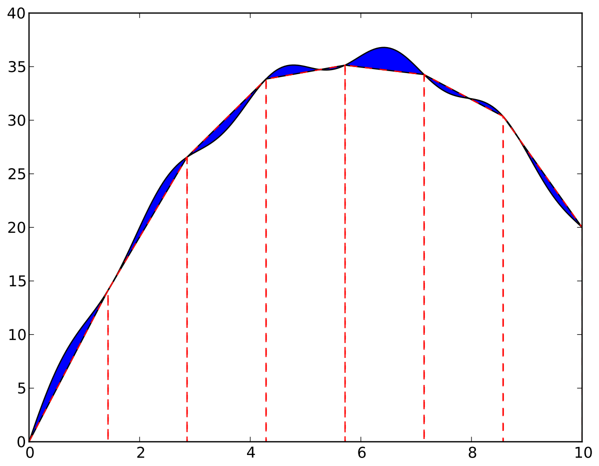
b) Use the approximation (3.9) to compute the following integrals: $\int_0^\pi \cos x\,dx$, $\int_0^\pi \sin x\,dx$, and $\int_0^{\pi/2} \sin x\,dx$, In each case, write out the error, i.e., the difference between the exact integral and the approximation (3.9). Make rough sketches of the trapezoid for each integral in order to understand how the method behaves in the different cases.

c) We can easily improve the formula (3.9) by approximating the area under the function $f(x)$ by two equal-sized trapezoids. Derive a formula for this approximation and implement it in a function `trapezint2(f, a, b)`. Run the examples from b) and see how much better the new formula is. Make sketches of the two trapezoids in each case.

d) A further improvement of the approximate integration method from c) is to divide the area under the $f(x)$ curve into n equal-sized trapezoids. Based on this idea, derive the following formula for approximating the integral:

$$\int_a^b f(x)dx \approx \sum_{0=1}^{n-1} \frac{1}{2}h\left(f(x_i) + f(x_{i+1})\right), \qquad (3.10)$$

where h is the width of the trapezoids, $h = (b - a)/n$, and $x_i = a + ih$, $i = 0, \ldots, n$, are the coordinates of the sides of the trapezoids. The figure below visualizes the idea of the Trapezoidal rule.



Implement (3.10) in a Python function `trapezint(f, a, b, n)`. Run the examples from b) with $n = 10$.

e) Write a test function `test_trapezint()` for verifying the implementation of the function `trapezint` in d).

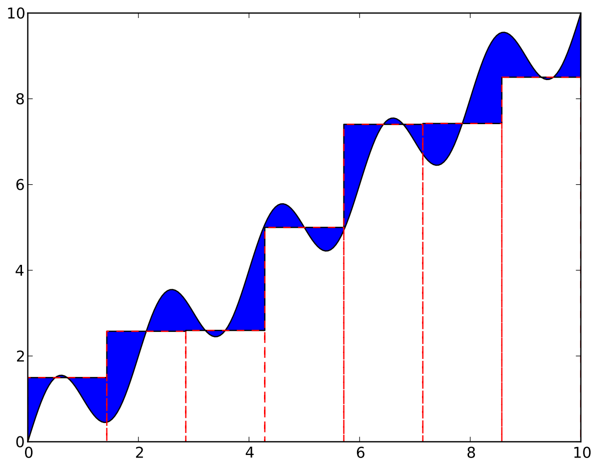
Hint. Obviously, the Trapezoidal method integrates linear functions exactly for any n. Another more surprising result is that the method is also exact for, e.g., $\int_0^{2\pi} \cos x \, dx$ for any n.
Filename: `trapezint.py`.

Remarks. Formula (3.10) is not the most common way of expressing the Trapezoidal integration rule. The reason is that $f(x_{i+1})$ is evaluated twice, first in term i and then as $f(x_i)$ in term $i + 1$. The formula can be further developed to avoid unnecessary evaluations of $f(x_{i+1})$, which results in the standard form

$$\int_a^b f(x)dx \approx \frac{1}{2}h(f(a) + f(b)) + h\sum_{i=1}^{n-1} f(x_i)\,. \qquad (3.11)$$

Exercise 3.7: Derive the general Midpoint integration rule

The idea of the Midpoint rule for integration is to divide the area under the curve $f(x)$ into n equal-sized rectangles (instead of trapezoids as in Exercise 3.6). The height of the rectangle is determined by the value of f at the midpoint of the rectangle. The figure below illustrates the idea.



Compute the area of each rectangle, sum them up, and arrive at the formula for the Midpoint rule:

$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-1} f(a + ih + \frac{1}{2}h), \tag{3.12}$$

where $h = (b - a)/n$ is the width of each rectangle. Implement this formula in a Python function `midpointint(f, a, b, n)` and test the function on the examples listed in Exercise 3.6b. How do the errors in the Midpoint rule compare with those of the Trapezoidal rule for $n = 1$ and $n = 10$? Filename: `midpointint.py`.

Exercise 3.8: Make an adaptive Trapezoidal rule

A problem with the Trapezoidal integration rule (3.10) in Exercise 3.6 is to decide how many trapezoids (n) to use in order to achieve a desired accuracy. Let E be the error in the Trapezoidal method, i.e., the difference between the exact integral and that produced by (3.10). We would like to prescribe a (small) tolerance ϵ and find an n such that $E \leq \epsilon$.

Since the exact value $\int_a^b f(x)dx$ is not available (that is why we use a numerical method!), it is challenging to compute E. Nevertheless, it has been shown by mathematicians that

$$E \leq \frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} |f''(x)| \ . \tag{3.13}$$

The maximum of $|f''(x)|$ can be computed (approximately) by evaluating $f''(x)$ at a large number of points in $[a, b]$, taking the absolute value $|f''(x)|$, and finding the maximum value of these. The double derivative can be computed by a finite difference formula:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}\,.$$

With the computed estimate of $\max |f''(x)|$ we can find h from setting the right-hand side in (3.13) equal to the desired tolerance:

$$\frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} |f''(x)| = \epsilon\,.$$

Solving with respect to h gives

$$h = \sqrt{12\epsilon}\left((b-a)\max_{x\in[a,b]} |f''(x)|\right)^{-1/2}. \tag{3.14}$$

With $n = (b-a)/h$ we have the n that corresponds to the desired accuracy ϵ.

a) Make a Python function `adaptive_trapezint(f, a, b, eps=1E-5)` for computing the integral $\int_a^b f(x)dx$ with an error less than or equal to ϵ (`eps`).

Hint. Compute the n corresponding to ϵ as explained above and call `trapezint(f, a, b, n)` from Exercise 3.6.

b) Apply the function to compute the integrals from Exercise 3.6b. Write out the exact error and the estimated n for each case.
Filename: `adaptive_trapezint.py`.

Remarks. A numerical method that applies an expression for the error to adapt the choice of the discretization parameter to a desired error tolerance, is known as an *adaptive* numerical method. The advantage of an adaptive method is that one can control the approximation error, and there is no need for the user to determine an appropriate number of intervals n.

Exercise 3.9: Explain why a program works

Explain how and thereby why the following program works:

```
def add(A, B):
    C = A + B
    return C

A = 3
B = 2
print add(A, B)
```

Exercise 3.10: Simulate a program by hand

Simulate the following program by hand to explain what is printed.

```
def a(x):
    q = 2
    x = 3*x
    return q + x

def b(x):
    global q
    q += x
    return q + x

q = 0
x = 3
print a(x), b(x), a(x), b(x)
```

Hint. If you encounter problems with understanding function calls and local versus global variables, paste the code into the Online Python Tutor[8] and step through the code to get a good explanation of what happens.

Exercise 3.11: Compute the area of an arbitrary triangle

An arbitrary triangle can be described by the coordinates of its three vertices: (x_1, y_1), (x_2, y_2), (x_3, y_3), numbered in a counterclockwise direction. The area of the triangle is given by the formula

$$A = \frac{1}{2} \left| x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1 \right| . \tag{3.15}$$

Write a function `area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a nested list of the vertex coordinates. For example, computing the area of the triangle with vertex coordinates $(0, 0)$, $(1, 0)$, and $(0, 2)$ is done by

```
triangle1 = area([[0,0], [1,0], [0,2]])
# or
v1 = (0,0);  v2 = (1,0);  v3 = (0,2)
vertices = [v1, v2, v3]
triangle1 = area(vertices)

print 'Area of triangle is %.2f' % triangle1
```

Test the `area` function on a triangle with known area. Filename: `area_triangle.py`.

Exercise 3.12: Compute the length of a path

Some object is moving along a path in the plane. At $n + 1$ points of time we have recorded the corresponding (x, y) positions of the object: (x_0, y_0), (x_1, y_2), ..., (x_n, y_n). The total length L of the path from (x_0, y_0)

[8] `http://www.pythontutor.com/visualize.html`

to (x_n, y_n) is the sum of all the individual line segments ((x_{i-1}, y_{i-1}) to (x_i, y_i), $i = 1, \ldots, n$):

$$L = \sum_{i=1}^{n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}\,. \tag{3.16}$$

a) Make a Python function `pathlength(x, y)` for computing L according to the formula. The arguments `x` and `y` hold all the $x_0, \ldots, x_n$ and $y_0, \ldots, y_n$ coordinates, respectively.

b) Write a test function `test_pathlength()` where you check that `pathlength` returns the correct length in a test problem.
Filename: `pathlength.py`.

Exercise 3.13: Approximate π

The value of π equals the circumference of a circle with radius 1/2. Suppose we approximate the circumference by a polygon through $n+1$ points on the circle. The length of this polygon can be found using the `pathlength` function from Exercise 3.12. Compute $n+1$ points (x_i, y_i) along a circle with radius 1/2 according to the formulas

$$x_i = \frac{1}{2}\cos(2\pi i/n), \quad y_i = \frac{1}{2}\sin(2\pi i/n), \quad i = 0, \ldots, n\,.$$

Call the `pathlength` function and write out the error in the approximation of π for $n = 2^k$, $k = 2, 3, \ldots, 10$. Filename: `pi_approx.py`.

Exercise 3.14: Write functions

Three functions, `hw1`, `hw2`, and `hw3`, work as follows:

```
>>> print hw1()
Hello, World!
>>> hw2()
Hello, World!
>>> print hw3('Hello, ', 'World!')
Hello, World!
>>> print hw3('Python ', 'function')
Python function
```

Write the three functions. Filename: `hw_func.py`.

Exercise 3.15: Approximate a function by a sum of sines

We consider the piecewise constant function

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t < T \end{cases} \tag{3.17}$$

Sketch this function on a piece of paper. One can approximate $f(t)$ by the sum

$$S(t;n) = \frac{4}{\pi}\sum_{i=1}^{n} \frac{1}{2i-1}\sin\left(\frac{2(2i-1)\pi t}{T}\right). \tag{3.18}$$

It can be shown that $S(t;n) \to f(t)$ as $n \to \infty$.

a) Write a Python function `S(t, n, T)` for returning the value of $S(t;n)$.

b) Write a Python function `f(t, T)` for computing $f(t)$.

c) Write out tabular information showing how the error $f(t) - S(t;n)$ varies with n and t for the cases where $n = 1, 3, 5, 10, 30, 100$ and $t = \alpha T$, with $T = 2\pi$, and $\alpha = 0.01, 0.25, 0.49$. Use the table to comment on how the quality of the approximation depends on α and n.
Filename: `sinesum1.py`.

Remarks. A sum of sine and/or cosine functions, as in (3.18), is called a *Fourier series*. Approximating a function by a Fourier series is a very important technique in science and technology. Exercise 5.39 asks for visualization of how well $S(t;n)$ approximates $f(t)$ for some values of n.

Exercise 3.16: Implement a Gaussian function

Make a Python function `gauss(x, m=0, s=1)` for computing the Gaussian function

$$f(x) = \frac{1}{\sqrt{2\pi}\, s}\exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right].$$

Write out a nicely formatted table of x and $f(x)$ values for n uniformly spaced x values in $[m - 5s, m + 5s]$. (Choose m, s, and n as you like.)
Filename: `gaussian2.py`.

Exercise 3.17: Wrap a formula in a function

Implement the formula (1.9) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters ρ, K, c, and T_w can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T_o = 25$ C). Filename: `egg_func.py`.

Exercise 3.18: Write a function for numerical differentiation

The formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \qquad (3.19)$$

can be used to find an approximate derivative of a mathematical function $f(x)$ if h is small.

a) Write a function `diff(f, x, h=1E-5)` that returns the approximation (3.19) of the derivative of a mathematical function represented by a Python function `f(x)`.

b) Write a function `test_diff()` that verifies the implementation of the function `diff`. As test case, one can use the fact that (3.19) is exact for quadratic functions. Follow the conventions of the pytest and nose testing frameworks, as outlined in Exercise 3.2 and Sections 3.3.3, 3.4.2, and H.6.

c) Apply (3.19) to differentiate

- $f(x) = e^x$ at $x = 0$,
- $f(x) = e^{-2x^2}$ at $x = 0$,
- $f(x) = \cos x$ at $x = 2\pi$,
- $f(x) = \ln x$ at $x = 1$.

Use $h = 0.01$. In each case, write out the error, i.e., the difference between the exact derivative and the result of (3.19). Collect these four examples in a function `application()`.
Filename: `centered_diff.py`.

Exercise 3.19: Implement the factorial function

The factorial of n is written as $n!$ and defined as

$$n! = n(n-1)(n-2)\cdots 2\cdot 1, \qquad (3.20)$$

with the special cases

$$1! = 1, \quad 0! = 1\,. \qquad (3.21)$$

For example, $4! = 4\cdot3\cdot2\cdot1 = 24$, and $2! = 2\cdot1 = 2$. Write a Python function `fact(n)` that returns $n!$. (Do not simply call the ready-made function `math.factorial(n)` - that is considered cheating in this context!)

Hint. Return 1 immediately if x is 1 or 0, otherwise use a loop to compute $n!$.
Filename: `fact.py`.

Exercise 3.20: Compute velocity and acceleration from 1D position data

Suppose we have recorded GPS coordinates $x_0, \ldots, x_n$ at times $t_0, \ldots, t_n$ while running or driving along a straight road. We want to compute the velocity v_i and acceleration a_i from these position coordinates. Using finite difference approximations, one can establish the formulas

$$v_i \approx \frac{x_{i+1} - x_{i-1}}{t_{i+1} - t_{i-1})}, \tag{3.22}$$

$$a_i \approx 2(t_{i+1} - t_{i-1})^{-1} \left(\frac{x_{i+1} - x_i}{t_{i+1} - t_i} - \frac{x_i - x_{i-1}}{t_i - t_{i-1}} \right), \tag{3.23}$$

for $i = 1, \ldots, n-1$.

a) Write a Python function `kinematics(x, i, dt=1E-6)` for computing v_i and a_i, given the array `x` of position coordinates $x_0, \ldots, x_n$.

b) Write a Python function `test_kinematics()` for testing the implementation in the case of constant velocity V. Set $t_0 = 0$, $t_1 = 0.5$, $t_2 = 1.5$, and $t_3 = 2.2$, and $x_i = Vt_i$.
Filename: `kinematics1.py`.

Exercise 3.21: Find the max and min values of a function

The maximum and minimum values of a mathematical function $f(x)$ on $[a, b]$ can be found by computing f at a large number (n) of points and selecting the maximum and minimum values at these points. Write a Python function `maxmin(f, a, b, n=1000)` that returns the maximum and minimum value of a function `f(x)`. Also write a test function for verifying the implementation for $f(x) = \cos x$, $x \in [-\pi/2, 2\pi]$.

Hint. The x points where the mathematical function is to be evaluated can be uniformly distributed: $x_i = a + ih$, $i = 0, \ldots, n-1$, $h = (b-a)/(n-1)$. The Python functions `max(y)` and `min(y)` return the maximum and minimum values in the list `y`, respectively.
Filename: `maxmin_f.py`.

Exercise 3.22: Find the max and min elements in a list

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. Write your own `max` and `min` functions.

Hint. Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, set `max_elem` equal to that element. Use a similar technique to compute the minimum element.
Filename: `maxmin_list.py`.

Exercise 3.23: Implement the Heaviside function

The following *step function* is known as the *Heaviside function* and is widely used in mathematics:

$$H(x) = \begin{cases} 0, \, x < 0 \\ 1, \, x \geq 0 \end{cases} \tag{3.24}$$

a) Implement $H(x)$ in a Python function `H(x)`.

b) Make a Python function `test_H()` for testing the implementation of `H(x)`. Compute $H(-10)$, $H(-10^{-15})$, $H(0)$, $H(10^{-15})$, $H(10)$ and test that the answers are correct.
Filename: `Heaviside.py`.

Exercise 3.24: Implement a smoothed Heaviside function

The Heaviside function (3.24) listed in Exercise 3.23 is discontinuous. It is in many numerical applications advantageous to work with a smooth version of the Heaviside function where the function itself and its first derivative are continuous. One such smoothed Heaviside function is

$$H_\epsilon(x) = \begin{cases} 0, & x < -\epsilon, \\ \frac{1}{2} + \frac{x}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi x}{\epsilon}\right), & -\epsilon \leq x \leq \epsilon \\ 1, & x > \epsilon \end{cases} \tag{3.25}$$

a) Implement $H_\epsilon(x)$ in a Python function `H_eps(x, eps=0.01)`.

b) Make a Python function `test_H_eps()` for testing the implementation of `H_eps`. Check the values of some $x < -\epsilon$, $x = -\epsilon$, $x = 0$, $x = \epsilon$, and some $x > \epsilon$.
Filename: `smoothed_Heaviside.py`.

Exercise 3.25: Implement an indicator function

In many applications there is need for an indicator function, which is 1 over some interval and 0 elsewhere. More precisely, we define

$$I(x; L, R) = \begin{cases} 1, \, x \in [L, R], \\ 0, \, \text{elsewhere} \end{cases} \tag{3.26}$$

a) Make two Python implementations of such an indicator function, one with a direct test if $x \in [L, R]$ and one that expresses the indicator function in terms of Heaviside functions (3.24):

$$I(x; L, R) = H(x - L)H(R - x)\,. \tag{3.27}$$

b) Make a test function for verifying the implementation of the functions in a). Check that correct values are returned for some $x < L$, $x = L$, $x = (L + R)/2$, $x = R$, and some $x > R$.
Filename: `indicator_func.py`.

Exercise 3.26: Implement a piecewise constant function

Piecewise constant functions have a lot of important applications when modeling physical phenomena by mathematics. A piecewise constant function can be defined as

$$f(x) = \begin{cases} v_0,\ x \in [x_0, x_1), \\ v_1,\ x \in [x_1, x_2), \\ \vdots \\ v_i \quad x \in [x_i, x_{i+1}), \\ \vdots \\ v_n \quad x \in [x_n, x_{n+1}] \end{cases} \tag{3.28}$$

That is, we have a union of non-overlapping intervals covering the domain $[x_0, x_{n+1}]$, and $f(x)$ is constant in each interval. One example is the function that is -1 on $[0, 1]$, 0 on $[1, 1.5]$, and 4 on $[1.5, 2]$, where we with the notation in (3.28) have $x_0 = 0, x_1 = 1, x_2 = 1.5, x_3 = 2$ and $v_0 = -1, v_1 = 0, v_3 = 4$.

a) Make a Python function `piecewise(x, data)` for evaluating a piecewise constant mathematical function as in (3.28) at the point `x`. The `data` object is a list of pairs (v_i, x_i) for $i = 0, \ldots, n$. For example, `data` is `[(0, -1), (1, 0), (1.5, 4)]` in the example listed above. Since x_{n+1} is not a part of the `data` object, we have no means for detecting whether `x` is to the right of the last interval $[x_n, x_{n+1}]$, i.e., we must assume that the user of the `piecewise` function sends in an $x \leq x_{n+1}$.

b) Design suitable test cases for the function `piecewise` and implement them in a test function `test_piecewise()`.
Filename: `piecewise_constant1.py`.

Exercise 3.27: Apply indicator functions

Implement piecewise constant functions, as defined in Exercise 3.26, by observing that

$$f(x) = \sum_{i=0}^{n} v_i I(x; x_i, x_{i+1}), \tag{3.29}$$

where $I(x; x_i, x_{i+1})$ is the indicator function from Exercise 3.25. Filename: `piecewise_constant2.py`.

Exercise 3.28: Test your understanding of branching

Consider the following code:

```
def where1(x, y):
    if x > 0:
        print 'quadrant I or IV'
    if y > 0:
        print 'quadrant I or II'

def where2(x, y):
    if x > 0:
        print 'quadrant I or IV'
    elif y > 0:
        print 'quadrant II'

for x, y in (-1, 1), (1, 1):
    where1(x,y)
    where2(x,y)
```

What is printed?

Exercise 3.29: Simulate nested loops by hand

Go through the code below by hand, statement by statement, and calculate the numbers that will be printed.

```
n = 3
for i in range(-1, n):
    if i != 0:
        print i

for i in range(1, 13, 2*n):
    for j in range(n):
        print i, j

for i in range(1, n+1):
    for j in range(i):
        if j:
            print i, j

for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                print i, j, k
```

You may use a debugger, see Section F.1, or the Online Python Tutor[9], see Section 3.1.2, to control what happens when you step through the code.

Exercise 3.30: Rewrite a mathematical function

We consider the $L(x;n)$ sum as defined in Section 3.1.8 and the corresponding function `L3(x, epsilon)` function from Section 3.1.10. The sum $L(x;n)$ can be written as

$$L(x;n) = \sum_{i=1}^{n} c_i, \quad c_i = \frac{1}{i}\left(\frac{x}{1+x}\right)^i .$$

a) Derive a relation between c_i and c_{i-1},

$$c_i = ac_{i-1},$$

where a is an expression involving i and x.

b) The relation $c_i = ac_{i-1}$ means that we can start with `term` as c_1, and then in each pass of the loop implementing the sum $\sum_i c_i$ we can compute the next term c_i in the sum as

```
term = a*term
```

Write a new version of the `L3` function, called `L3_ci(x, epsilon)`, that makes use of this alternative computation of the terms in the sum.

c) Write a Python function `test_L3_ci()` that verifies the implementation of `L3_ci` by comparing with the original `L3` function.
Filename: `L3_recursive.py`.

Exercise 3.31: Make a table for approximations of $\cos x$

The function $\cos(x)$ can be approximated by the sum

$$C(x;n) = \sum_{j=0}^{n} c_j, \tag{3.30}$$

where

$$c_j = -c_{j-1}\frac{x^2}{2j(2j-1)}, \quad j = 1, 2, \ldots, n,$$

and $c_0 = 1$.

a) Make a Python function for computing $C(x;n)$.

[9] `http://www.pythontutor.com/`

Hint. Represent c_j by a variable `term`, make updates `term = -term*...` inside a `for` loop, and accumulate the `term` variable in a variable for the sum.

b) Make a function for writing out a table of the errors in the approximation $C(x;n)$ of $\cos(x)$ for some x and n values given as arguments to the function. Let the x values run downward in the rows and the n values to the right in the columns. For example, a table for $x = 4\pi, 6\pi, 8\pi, 10\pi$ and $n = 5, 25, 50, 100, 200$ can look like

```
      x          5          25         50         100         200
  12.5664   1.61e+04   1.87e-11   1.74e-12   1.74e-12   1.74e-12
  18.8496   1.22e+06   2.28e-02   7.12e-11   7.12e-11   7.12e-11
  25.1327   2.41e+07   6.58e+04  -4.87e-07  -4.87e-07  -4.87e-07
  31.4159   2.36e+08   6.52e+09   1.65e-04   1.65e-04   1.65e-04
```

Observe how the error increases with x and decreases with n.
Filename: `cos_sum.py`.

Exercise 3.32: Use None in keyword arguments

Consider the functions `L2(x, n)` and `L3(x, epsilon)` from Sections 3.1.8 and 3.1.10, whose program code is found in the file `lnsum.py`.

Make a more flexible function `L4` where we can either specify a tolerance `epsilon` *or* a number of terms `n` in the sum. Moreover, we can also choose whether we want the sum to be returned or the sum and the number of terms:

```
value, n = L4(x, epsilon=1E-8, return_n=True)
value = L4(x, n=100)
```

Hint. The starting point for all this flexibility is to have some keyword arguments initialized to an "undefined" value, called `None`, which can be recognized inside the function:

```
def L3(x, n=None, epsilon=None, return_n=False):
    if n is not None:
        ...
    if epsilon is not None:
        ...
```

One can also apply `if n != None`, but the `is` operator is most common.

Print error messages for incompatible values when `n` *and* `epsilon` are `None` or both are given by the user.
Filename: `L4.py`.

Exercise 3.33: Write a sort function for a list of 4-tuples

Below is a list of the nearest stars and some of their properties. The list elements are 4-tuples containing the name of the star, the distance from the sun in light years, the apparent brightness, and the luminosity. The apparent brightness is how bright the stars look in our sky compared to the brightness of Sirius A. The luminosity, or the true brightness, is how bright the stars would look if all were at the same distance compared to the Sun. The list data are found in the file `stars.txt`[10], which looks as follows:

```
data = [
('Alpha Centauri A',    4.3,  0.26,      1.56),
('Alpha Centauri B',    4.3,  0.077,     0.45),
('Alpha Centauri C',    4.2,  0.00001,   0.00006),
("Barnard's Star",      6.0,  0.00004,   0.0005),
('Wolf 359',            7.7,  0.000001,  0.00002),
('BD +36 degrees 2147', 8.2,  0.0003,    0.006),
('Luyten 726-8 A',      8.4,  0.000003,  0.00006),
('Luyten 726-8 B',      8.4,  0.000002,  0.00004),
('Sirius A',            8.6,  1.00,      23.6),
('Sirius B',            8.6,  0.001,     0.003),
('Ross 154',            9.4,  0.00002,   0.0005),
]
```

The purpose of this exercise is to sort this list with respect to distance, apparent brightness, and luminosity. Write a program that initializes the `data` list as above and writes out three sorted tables: star name versus distance, star name versus apparent brightness, and star name versus luminosity.

Hint. To sort a list `data`, one can call `sorted(data)`, as in

```
for item in sorted(data):
    ...
```

However, in the present case each element is a 4-tuple, and the default sorting of such 4-tuples results in a list with the stars appearing in alphabetic order. This is not what you want. Instead, we need to sort with respect to the 2nd, 3rd, or 4th element of each 4-tuple. If such a tailored sort mechanism is necessary, we can provide our own sort function as a second argument to `sorted`: `sorted(data, mysort)`. A sort function `mysort` must take two arguments, say `a` and `b`, and return -1 if `a` should become before `b` in the sorted sequence, 1 if `b` should become before `a`, and 0 if they are equal. In the present case, `a` and `b` are 4-tuples, so we need to make the comparison between the right elements in `a` and `b`. For example, to sort with respect to luminosity we can write

```
def mysort(a, b):
    if a[3] < b[3]:
        return -1
    elif a[3] > b[3]:
        return 1
    else:
        return 0
```

Filename: `sorted_stars_data.py`.

[10] http://tinyurl.com/pwyasaa/funcif/stars.txt

Exercise 3.34: Find prime numbers

The *Sieve of Eratosthenes* is an algorithm for finding all prime numbers less than or equal to a number N. Read about this algorithm on Wikipedia and implement it in a Python program. Filename: `find_primes.py`.

Exercise 3.35: Find pairs of characters

Write a function `count_pairs(dna, pair)` that returns the number of occurrences of a pair of characters (`pair`) in a DNA string (`dna`). For example, calling the function with `dna` as `'ACTGCTATCCATT'` and `pair` as `'AT'` will return 2. Filename: `count_pairs.py`.

Exercise 3.36: Count substrings

This is an extension of Exercise 3.35: count how many times a certain string appears in another string. For example, the function returns 3 when called with the DNA string `'ACGTTACGGAACG'` and the substring `'ACG'`.

Hint. For each match of the first character of the substring in the main string, check if the next `n` characters in the main string matches the substring, where `n` is the length of the substring. Use slices like `s[3:9]` to pick out a substring of `s`.
Filename: `count_substr.py`.

Exercise 3.37: Resolve a problem with a function

Consider the following interactive session:

```
>>> def f(x):
...     if 0 <= x <= 2:
...         return x**2
...     elif 2 < x <= 4:
...         return 4
...     elif x < 0:
...         return 0
...
>>> f(2)
4
>>> f(5)
>>> f(10)
```

Why do we not get any output when calling `f(5)` and `f(10)`?

Hint. Save the `f` value in a variable `r` and do `print r`.

Exercise 3.38: Determine the types of some objects

Consider the following calls to the makelist function from Section 3.1.6:

```
l1 = makelist(0, 100, 1)
l2 = makelist(0, 100, 1.0)
l3 = makelist(-1, 1, 0.1)
l4 = makelist(10, 20, 20)
l5 = makelist([1,2], [3,4], [5])
l6 = makelist((1,-1,1), ('myfile.dat', 'yourfile.dat'))
l7 = makelist('myfile.dat', 'yourfile.dat', 'herfile.dat')
```

Simulate each call by hand to determine what type of objects that become elements in the returned list and what the contents of `value` is after one pass in the loop.

Hint. Note that some of the calls will lead to infinite loops if you really perform the above `makelist` calls on a computer.

You can go to the Online Python Tutor[11], paste in the `makelist` function and the session above, and step through the program to see what actually happens.

Remarks. This exercise demonstrates that we can write a function and have in mind certain types of arguments, here typically `int` and `float` objects. However, the function can be used with other (originally unintended) arguments, such as lists and strings in the present case, leading to strange and irrelevant behavior (the problem here lies in the boolean expression `value <= stop` which is meaningless for some of the arguments).

Exercise 3.39: Find an error in a program

Consider the following program for computing

$$f(x) = e^{rx}\sin(mx) + e^{sx}\sin(nx)\,.$$

```
def f(x, m, n, r, s):
    return expsin(x, r, m) + expsin(x, s, n)

x = 2.5
print f(x, 0.1, 0.2, 1, 1)

from math import exp, sin

def expsin(x, p, q):
    return exp(p*x)*sin(q*x)
```

Running this code results in

```
NameError: global name 'expsin' is not defined
```

What is the problem? Simulate the program flow by hand, use the debugger to step from line to line, or use the Online Python Tutor. Correct the program.

[11] http://www.pythontutor.com/

Potential problems with the software. Let us solve

- $x = \tanh x$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$,
- $x^5 = \tanh(x^5)$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$.

Both equations have one root $x = 0$.

```
Terminal

bisection.py "x-tanh(x)" -10 10
Found root x=-5.96046e-07 in 25 iterations

bisection.py "x**5-tanh(x**5)" -10 10
Found root x=-0.0266892 in 25 iterations
```

These results look strange. In both cases we halve the start interval $[-10, 10]$ 25 times, but in the second case we end up with a much less accurate root although the value of ϵ is the same. A closer inspection of what goes on in the bisection algorithm reveals that the inaccuracy is caused by round-off errors. As $a, b, m \to 0$, raising a small number to the fifth power in the expression for $f(x)$ yields a much smaller result. Subtracting a very small number $\tanh x^5$ from another very small number x^5 may result in a small number with wrong sign, and the sign of f is essential in the bisection algorithm. We encourage the reader to graphically inspect this behavior by running these two examples with the `bisection_plot.py` program using a smaller interval $[-1, 1]$ to better see what is going on. The command-line arguments for the `bisection_plot.py` program are `'x-tanh(x)' -1 1` and `'x**5-tanh(x**5)' -1 1`. The very flat area, in the latter case, where $f(x) \approx 0$ for $x \in [-1/2, 1/2]$ illustrates well that it is difficult to locate an exact root.

Distributing the bisection module to others. The Python standard for installing software is to run a `setup.py` program,

```
Terminal

Terminal> sudo python setup.py install
```

to install the system. The relevant `setup.py` for the `bisection` module arises from substituting the name `interest` by `bisection` in the `setup.py` file listed in Section 4.9.8. You can then distribute `bisection.py` and `setup.py` together.

4.11 Exercises

Exercise 4.1: Make an interactive program

Make a program that asks the user for a temperature in Fahrenheit degrees and reads the number; computes the corresponding temperature in Celsius degrees; and prints out the temperature in the Celsius scale. Filename: `f2c_qa.py`.

Exercise 4.2: Read a number from the command line

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from the command line. Filename: `f2c_cml.py`.

Exercise 4.3: Read a number from a file

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from a file with the following content:

```
Temperature data
----------------

Fahrenheit degrees: 67.2
```

Hint. Create a sample file manually. In the program, skip the first three lines, split the fourth line into words and grab the third word.
Filename: `f2c_file_read.py`.

Exercise 4.4: Read and write several numbers from and to file

This is a variant of Exercise 4.3 where we have several Fahrenheit degrees in a file and want to read all of them into a list and convert the numbers to Celsius degrees. Thereafter, we want to write out a file with two columns, the left with the Fahrenheit degrees and the right with the Celsius degrees.

An example on the input file format looks like

```
Temperature data
----------------

Fahrenheit degrees: 67.2
Fahrenheit degrees: 66.0
Fahrenheit degrees: 78.9
Fahrenheit degrees: 102.1
Fahrenheit degrees: 32.0
Fahrenheit degrees: 87.8
```

A sample file is `Fdeg.dat`[5]. Filename: `f2c_file_read_write.py`.

Exercise 4.5: Use exceptions to handle wrong input

Extend the program from Exercise 4.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Filename: `f2c_cml_exc.py`.

[5] `http://tinyurl.com/pwyasaa/input/Fdeg.dat`

Exercise 4.6: Read input from the keyboard

Make a program that asks for input from the user, applies `eval` to this input, and prints out the type of the resulting object and its value. Test the program by providing five types of input: an integer, a real number, a complex number, a list, and a tuple. Filename: `objects_qa.py`.

Exercise 4.7: Read input from the command line

a) Let a program store the result of applying the `eval` function to the first command-line argument. Print out the resulting object and its type.

b) Run the program with different input: an integer, a real number, a list, and a tuple.

Hint. On Unix systems you need to surround the tuple expressions in quotes on the command line to avoid error message from the Unix shell.

c) Try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy?
Filename: `objects_cml.py`.

Exercise 4.8: Try MSWord or LibreOffice to write a program

The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

a) Type the following one-line program in either MSWord or LibreOffice:

```
print "Hello, World!"
```

Both Word and LibreOffice are so “smart” that they automatically edit “print” to “Print” since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

b) Save the program as a `.docx` (Word) or `.odt` (LibreOffice) file. Now try to run this file as a Python program. What kind of error message do you get? Can you explain why?

c) Save the program as a `.txt` file in Word or LibreOffice and run the file as a Python program. What happened now? Try to find out what the problem is.

Exercise 4.9: Prompt the user for input to a formula

Consider the simplest program for evaluating the formula $y(t) = v_0 t - \frac{1}{2}gt^2$:

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Modify this code so that the program asks the user questions `t=?` and `v0=?`, and then gets `t` and `v0` from the user's input through the keyboard. Filename: `ball_qa.py`.

Exercise 4.10: Read parameters in a formula from the command line

Modify the program listed in Exercise 4.9 such that `v0` and `t` are read from the command line. Filename: `ball_cml.py`.

Exercise 4.11: Use exceptions to handle wrong input

The program from Exercise 4.10 reads input from the command line. Extend that program with exception handling such that missing command-line arguments are detected. In the `except IndexError` block, use the `raw_input` function to ask the user for missing input data. Filename: `ball_cml_qa.py`.

Exercise 4.12: Test validity of input data

Test if the `t` value read in the program from Exercise 4.10 lies between 0 and $2v_0/g$. If not, print a message and abort the execution. Filename: `ball_cml_tcheck.py`.

Exercise 4.13: Raise an exception in case of wrong input

Instead of printing an error message and aborting the program explicitly, raise a `ValueError` exception in the `if` test on legal `t` values in the program from Exercise 4.12. Notify the user about the legal interval for t in the exception message. Filename: `ball_cml_ValueError.py`.

Exercise 4.14: Evaluate a formula for data in a file

We consider the formula $y(t) = v_0 t - 0.5gt^2$ and want to evaluate y for a range of t values found in a file with format

```
v0: 3.00
t:
0.15592  0.28075   0.36807889 0.35 0.57681501876
0.21342619  0.0519085  0.042  0.27  0.50620017 0.528
0.2094294  0.1117  0.53012  0.3729850  0.39325246
0.21385894  0.3464815 0.57982969 0.10262264
0.29584013  0.17383923
```

More precisely, the first two lines are always present, while the next lines contain an arbitrary number of t values on each line, separated by one or more spaces.

a) Write a function that reads the input file and returns v_0 and a list with the t values.

b) Write a function that creates a file with two nicely formatted columns containing the t values to the left and the corresponding y values to the right. Let the t values appear in increasing order (note that the input file does not necessarily have the t values sorted).

c) Make a test function that generates an input file, calls the function for reading the file, and checks that the returned data objects are correct. Filename: `ball_file_read_write.py`.

Exercise 4.15: Compute the distance it takes to stop a car

A car driver, driving at velocity v_0, suddenly puts on the brake. What braking distance d is needed to stop the car? One can derive, using Newton's second law of motion or a corresponding energy equation, that

$$d = \frac{1}{2}\frac{v_0^2}{\mu g}\,. \tag{4.7}$$

Make a program for computing d in (4.7) when the initial car velocity v_0 and the friction coefficient μ are given on the command line. Run the program for two cases: $v_0 = 120$ and $v_0 = 50$ km/h, both with $\mu = 0.3$ (μ is dimensionless).

Hint. Remember to convert the velocity from km/h to m/s before inserting the value in the formula.
Filename: `stopping_length.py`.

Exercise 4.16: Look up calendar functionality

The purpose of this exercise is to make a program that takes a date, consisting of year (4 digits), month (2 digits), and day (1-31) on the command line and prints the corresponding name of the weekday (Monday, Tuesday, etc.). Python has a module `calendar`, which makes it easy to solve the exercise, but the task is to find out how to use this module. Filename: `weekday.py`.

Exercise 4.17: Use the StringFunction tool

Make the program `user_formula.py` from Section 4.3.2 shorter by using the convenient `StringFunction` tool from Section 4.3.3. Filename: `user_formula2.py`.

Exercise 4.18: Why we test for specific exception types

The simplest way of writing a `try-except` block is to test for *any* exception, for example,

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command-line argument'
    sys.exit(1)
```

Write the above statements in a program and test the program. What is the problem?

The fact that a user can forget to supply a command-line argument when running the program was the original reason for using a `try` block. Find out what kind of exception that is relevant for this error and test for this specific exception and re-run the program. What is the problem now? Correct the program. Filename: `unnamed_exception.py`.

Exercise 4.19: Make a complete module

a) Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: `C2F`, `F2C`, `C2K`, `K2C`, `F2K`, and `K2F`.

b) Collect these functions in a module `convert_temp`.

c) Import the module in an interactive Python shell and demonstrate some sample calls on temperature conversions.

d) Insert the session from c) in a triple quoted string at the top of the module file as a doc string for demonstrating the usage.

e) Write a function `test_conversion()` that verifies the implementation. Call this function from the test block if the first command-line argument is `verify`.

Hint. Check that `C2F(F2C(f))` is `f`, `K2C(C2K(c))` is `c`, and `K2F(F2K(f))` is `f` - with tolerance. Follow the conventions for test functions outlined in Sections 4.9.4 and 4.10.2 with a boolean variable that is `False` if a test failed, and `True` if all test are passed, and then an `assert` statement to abort the program when any test fails.

f) Add a user interface to the module such that the user can write a temperature as the first command-line argument and the corresponding temperature scale as the second command-line argument, and then get the temperature in the two other scales as output. For example, `21.3 C` on the command line results in the output `70.3 F 294.4 K`. Encapsulate the user interface in a function, which is called from the test block.
Filename: `convert_temp.py`.

Exercise 4.20: Make a module

Collect the `f` and `S` functions in the program from Exercise 3.15 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program from Exercise 3.15) in a separate function `table(n_values, alpha_values, T)`. Make a test block in the module to read T and a series of n and α values from the command line and make a corresponding call to `table`. Filename: `sinesum2.py`.

Exercise 4.21: Read options and values from the command line

Let the input to the program in Exercise 4.20 be option-value pairs with the options `-n`, `-alpha`, and `-T`. Provide sensible default values in the module file.

Hint. Apply the `argparse` module to read the command-line arguments. Do not copy code from the `sinesum2` module, but make a new file for reading option-value pairs from the command and import the `table` function from the `sinesum2` module.
Filename: `sinesum3.py`.

Exercise 4.22: Check if mathematical identities hold

Because of round-off errors, it could happen that a mathematical rule like $(ab)^3 = a^3b^3$ does not hold exactly on a computer. The idea of testing this potential problem is to check such identities for a large number of random numbers. We can make random numbers using the `random` module in Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Here, `a` and `b` will be random numbers, which are always larger than or equal to `A` and smaller than `B`.

a) Make a function `power3_identity(A=-100, B=100, n=1000)` that tests the identity `(a*b)**3 == a**3*b**3` a large number of times, `n`. Return the fraction of failures.

Hint. Inside the loop over `n`, draw random numbers `a` and `b` as described above and count the number of times the test is `True`.

b) We shall now parameterize the expressions to be tested. Make a function

```
equal(expr1, expr2, A=-100, B=100, n=500)
```

where `expr1` and `expr2` are strings containing the two mathematical expressions to be tested. More precisely, the function draws random numbers `a` and `b` between `A` and `B` and tests if `eval(expr1) == eval(expr2)`. Return the fraction of failures.

Test the function on the identities $(ab)^3 = a^3b^3$, $e^{a+b} = e^ae^b$, and $\ln a^b = b\ln a$.

Hint. Make the `equal` function robust enough to handle illegal a and b values in the mathematical expressions (e.g., $a \le 0$ in $\ln a$).

c) We want to test the validity of the following set of identities on a computer:

- $a - b$ and $-(b - a)$
- a/b and $1/(b/a)$
- $(ab)^4$ and a^4b^4
- $(a + b)^2$ and $a^2 + 2ab + b^2$
- $(a + b)(a - b)$ and $a^2 - b^2$
- e^{a+b} and e^ae^b
- $\ln a^b$ and $b\ln a$
- $\ln ab$ and $\ln a + \ln b$
- ab and $e^{\ln a+\ln b}$
- $1/(1/a + 1/b)$ and $ab/(a + b)$
- $a(\sin^2 b + \cos^2 b)$ and a
- $\sinh(a + b)$ and $(e^ae^b - e^{-a}e^{-b})/2$
- $\tan(a + b)$ and $\sin(a + b)/\cos(a + b)$
- $\sin(a + b)$ and $\sin a\cos b + \sin b\cos a$

Store all the expressions in a list of 2-tuples, where each 2-tuple contains two mathematically equivalent expressions as strings, which can be sent to the `equal` function. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate. Write this table to a file. Try out `A=1` and `B=2` as well as `A=1` and `B=100`. Does the failure rate seem to depend on the magnitude of the numbers a and b? Filename: `math_identities_failures.py`.

Exercise 4.23: Compute probabilities with the binomial distribution

Consider an uncertain event where there are two outcomes only, typically success or failure. Flipping a coin is an example: the outcome is uncertain and of two types, either head (can be considered as success) or tail (failure). Throwing a die can be another example, if (e.g.) getting a six is considered success and all other outcomes represent failure. Such experiments are called *Bernoulli trials*.

Let the probability of success be p and that of failure $1-p$. If we perform n experiments, where the outcome of each experiment does not depend on the outcome of previous experiments, the probability of getting success x times, and consequently failure $n-x$ times, is given by

$$B(x,n,p) = \frac{n!}{x!(n-x)!}p^x(1-p)^{n-x}\,. \tag{4.8}$$

This formula (4.8) is called the binomial distribution. The expression $x!$ is the factorial of x: $x! = x(x-1)(x-2)\cdots 1$ and `math.factorial` can do this computation.

a) Implement (4.8) in a function `binomial(x, n, p)`.

b) What is the probability of getting two heads when flipping a coin five times? This probability corresponds to $n=5$ events, where the success of an event means getting head, which has probability $p=1/2$, and we look for $x=2$ successes.

c) What is the probability of getting four ones in a row when throwing a die? This probability corresponds to $n=4$ events, success is getting one and has probability $p=1/6$, and we look for $x=4$ successful events.

d) Suppose cross country skiers typically experience one ski break in one out of 120 competitions. Hence, the probability of breaking a ski can be set to $p=1/120$. What is the probability b that a skier will experience a ski break during five competitions in a world championship?

Hint. This question is a bit more demanding than the other two. We are looking for the probability of 1, 2, 3, 4 or 5 ski breaks, so it is simpler to ask for the probability c of *not* breaking a ski, and then compute $b=1-c$. Define *success* as breaking a ski. We then look for $x=0$ successes out of $n=5$ trials, with $p=1/120$ for each trial. Compute b.
Filename: `Bernoulli_trials.py`.

Exercise 4.24: Compute probabilities with the Poisson distribution

Suppose that over a period of t_m time units, a particular uncertain event happens (on average) νt_m times. The probability that there will be x such events in a time period t is approximately given by the formula

$$P(x,t,\nu) = \frac{(\nu t)^x}{x!} e^{-\nu t}\,. \tag{4.9}$$

This formula is known as the Poisson distribution. (It can be shown that (4.9) arises from (4.8) when the probability p of experiencing the event in a small time interval t/n is $p = \nu t/n$ and we let $n \to \infty$.) An important assumption is that all events are independent of each other and that the probability of experiencing an event does not change significantly over time. This is known as a *Poisson process* in probability theory.

a) Implement (4.9) in a function `Poisson(x, t, nu)`, and make a program that reads x, t, and ν from the command line and writes out the probability $P(x,t,\nu)$. Use this program to solve the problems below.

b) Suppose you are waiting for a taxi in a certain street at night. On average, 5 taxis pass this street every hour at this time of the night. What is the probability of not getting a taxi after having waited 30 minutes? Since we have 5 events in a time period of $t_m = 1$ hour, $\nu t_m = \nu = 5$. The sought probability is then $P(0, 1/2, 5)$. Compute this number. What is the probability of having to wait two hours for a taxi? If 8 people need two taxis, that is the probability that two taxis arrive in a period of 20 minutes?

c) In a certain location, 10 earthquakes have been recorded during the last 50 years. What is the probability of experiencing exactly three earthquakes over a period of 10 years in this area? What is the probability that a visitor for one week does not experience any earthquake? With 10 events over 50 years we have $\nu t_m = \nu \cdot 50$ years $= 10$ events, which implies $\nu = 1/5$ event per year. The answer to the first question of having $x = 3$ events in a period of $t = 10$ years is given directly by (4.9). The second question asks for $x = 0$ events in a time period of 1 week, i.e., $t = 1/52$ years, so the answer is $P(0, 1/52, 1/5)$.

d) Suppose that you count the number of misprints in the first versions of the reports you write and that this number shows an average of six misprints per page. What is the probability that a reader of a first draft of one of your reports reads six pages without hitting a misprint? Assuming that the Poisson distribution can be applied to this problem, we have "time" t_m as 1 page and $\nu \cdot 1 = 6$, i.e., $\nu = 6$ events (misprints) per page. The probability of no events in a "period" of six pages is $P(0, 6, 6)$.
Filename: `Poisson_processes.py`.

confined to a thin layer close to the surface, while a small b leads to temperature variations also deep down in the ground. You are encouraged to run the program with $b = 2$ and $b = 20$ to experience the major difference, or just view the ready-made animations[7].

We can understand the results from a physical perspective. Think of increasing ω, which means reducing the oscillation period so we get a more rapid temperature variation. To preserve the value of b we must increase k by the same factor. Since a large k means that heat quickly spreads down in the ground, and a small k implies the opposite, we see that more rapid variations at the surface requires a larger k to more quickly conduct the variations down in the ground. Similarly, slow temperature variations on the surface can penetrate deep in the ground even if the ground's ability to conduct (k) is low.

5.9 Exercises

Exercise 5.1: Fill lists with function values

Define

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \, . \tag{5.15}$$

Fill lists `xlist` and `hlist` with x and $h(x)$ values for 41 uniformly spaced x coordinates in $[-4, 4]$.

Hint. You may adapt the example in Section 5.2.1.
Filename: `fill_lists.py`.

Exercise 5.2: Fill arrays; loop version

The aim is to fill two arrays `x` and `y` with x and $h(x)$ values, respectively, where $h(x)$ is defined in (5.15). Let the x values be as in Exercise 5.1. Create empty `x` and `y` arrays and compute each element in `x` and `y` with a `for` loop. Filename: `fill_arrays_loop.py`.

Exercise 5.3: Fill arrays; vectorized version

Vectorize the code in Exercise 5.2 by creating the x values using the `linspace` function from the `numpy` package and by evaluating $h(x)$ for an array argument. Filename: `fill_arrays_vectorized.py`.

[7] `http://hplgit.github.io/scipro-primer/video/heatwave.html`

Exercise 5.4: Plot a function

Make a plot of the function in Exercise 5.1 for $x \in [-4, 4]$. Filename: `plot_Gaussian.py`.

Exercise 5.5: Apply a function to a vector

Given a vector $v = (2, 3, -1)$ and a function $f(x) = x^3 + xe^x + 1$, apply f to each element in v. Then calculate by hand $f(v)$ as the NumPy expression `v**3 + v*exp(v) + 1` using vector computing rules. Demonstrate that the two results are equal.

Exercise 5.6: Simulate by hand a vectorized expression

Suppose `x` and `t` are two arrays of the same length, entering a vectorized expression

```
y = cos(sin(x)) + exp(1/t)
```

If `x` holds two elements, 0 and 2, and `t` holds the elements 1 and 1.5, calculate by hand (using a calculator) the `y` array. Thereafter, write a program that mimics the series of computations you did by hand (typically a sequence of operations of the kind we listed in Section 5.1.3 - use explicit loops, but at the end you can use Numerical Python functionality to check the results). Filename: `simulate_vector_computing.py`.

Exercise 5.7: Demonstrate array slicing

Create an array `w` with values $0, 0.1, 0.2, \ldots, 3$. Write out `w[:]`, `w[:-2]`, `w[::5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed. Filename: `slicing.py`.

Exercise 5.8: Replace list operations by array computing

The data analysis problem in Section 2.6.2 is solved by list operations. Convert the list to a two-dimensional array and perform the computations using array operations (i.e., no explicit loops, but you need a loop to make the printout). Filename: `sun_data_vec.py`.

Exercise 5.9: Plot a formula

Make a plot of the function $y(t) = v_0 t - \frac{1}{2} g t^2$ for $v_0 = 10$, $g = 9.81$, and $t \in [0, 2v_0/g]$. Set the axes labels as `time (s)` and `height (m)`. Filename: `plot_ball1.py`.

Exercise 5.10: Plot a formula for several parameters

Make a program that reads a set of v_0 values from the command line and plots the corresponding curves $y(t) = v_0 t - \frac{1}{2}gt^2$ in the same figure, with $t \in [0, 2v_0/g]$ for each curve. Set $g = 9.81$.

Hint. You need a different vector of t coordinates for each curve. Filename: `plot_ball2.py`.

Exercise 5.11: Specify the extent of the axes in a plot

Extend the program from Exercises 5.10 such that the minimum and maximum t and y values are computed, and use the extreme values to specify the extent of the axes. Add some space above the highest curve to make the plot look better. Filename: `plot_ball3.py`.

Exercise 5.12: Plot exact and inexact Fahrenheit-Celsius conversion formulas

A simple rule to quickly compute the Celsius temperature from the Fahrenheit degrees is to subtract 30 and then divide by 2: $C = (F-30)/2$. Compare this curve against the exact curve $C = (F - 32)5/9$ in a plot. Let F vary between -20 and 120. Filename: `f2c_shortcut_plot.py`.

Exercise 5.13: Plot the trajectory of a ball

The formula for the trajectory of a ball is given by

$$f(x) = x \tan\theta - \frac{1}{2v_0^2}\frac{gx^2}{\cos^2\theta} + y_0, \tag{5.16}$$

where x is a coordinate along the ground, g is the acceleration of gravity, v_0 is the size of the initial velocity, which makes an angle θ with the x axis, and $(0, y_0)$ is the initial position of the ball.

In a program, first read the input data y_0, θ, and v_0 from the command line. Then plot the trajectory $y = f(x)$ for $y \geq 0$. Filename: `plot_trajectory.py`.

Exercise 5.14: Plot data in a two-column file

The file `src/plot/xy.dat`[8] contains two columns of numbers, corresponding to x and y coordinates on a curve. The start of the file looks as this:

[8] `http://tinyurl.com/pwyasaa/plot/xy.dat`

```
-1.0000       -0.0000
-0.9933       -0.0087
-0.9867       -0.0179
-0.9800       -0.0274
-0.9733       -0.0374
```

Make a program that reads the first column into a list `x` and the second column into a list `y`. Plot the curve. Print out the mean y value as well as the maximum and minimum y values.

Hint. Read the file line by line, split each line into words, convert to `float`, and append to `x` and `y`. The computations with `y` are simpler if the list is converted to an array.
Filename: `read_2columns.py`.

Remarks. The function `loadtxt` in `numpy` can read files with tabular data (any number of columns) and return the data in a two-dimensional array:

```
import numpy as np
data = np.loadtxt('xy.dat', dtype=np.float)  # read table of floats
x = data[:,0]  # column with index 0
y = data[:,1]  # column with index 1
```

The present exercise asks you to implement a simplified version of `loadtxt`, but for later loading of a file with tabular data into an array you will certainly use `loadtxt`.

Exercise 5.15: Write function data to file

We want to dump x and $f(x)$ values to a file, where the x values appear in the first column and the $f(x)$ values appear in the second. Choose n equally spaced x values in the interval $[a, b]$. Provide f, a, b, n, and the filename as input data on the command line.

Hint. Use the `StringFunction` tool (see Sections 4.3.3 and 5.5.1) to turn the textual expression for f into a Python function. (Note that the program from Exercise 5.14 can be used to read the file generated in the present exercise into arrays again for visualization of the curve $y = f(x)$.)
Filename: `write_cml_function.py`.

Exercise 5.16: Plot data from a file

The files `density_water.dat` and `density_air.dat` files in the folder `src/plot`[9] contain data about the density of water and air (respectively) for different temperatures. The data files have some comment lines starting with `#` and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding

[9] `http://tinyurl.com/pwyasaa/plot`

density in the second column. The goal of this exercise is to read the data in such a file and plot the density versus the temperature as distinct (small) circles for each data point. Let the program take the name of the data file as command-line argument. Apply the program to both files. Filename: `read_density_data.py`.

Exercise 5.17: Fit a polynomial to data points

The purpose of this exercise is to find a simple mathematical formula for how the density of water or air depends on the temperature. The idea is to load density and temperature data from file as explained in Exercise 5.16 and then apply some NumPy utilities that can find a polynomial that approximates the density as a function of the temperature.

NumPy has a function `polyfit(x, y, deg)` for finding a "best fit" of a polynomial of degree `deg` to a set of data points given by the array arguments `x` and `y`. The `polyfit` function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term. For example, given points in `x` and `y`, `polyfit(x, y, 1)` returns the coefficients `a, b` in a polynomial `a*x + b` that fits the data in the best way. (More precisely, a line $y = ax + b$ is a "best fit" to the data points (x_i, y_i), $i = 0, \ldots, n-1$ if a and b are chosen to make the sum of squared errors $R = \sum_{j=0}^{n-1}(y_j - (ax_j + b))^2$ as small as possible. This approach is known as *least squares approximation* to data and proves to be extremely useful throughout science and technology.)

NumPy also has a utility `poly1d`, which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print p             # prints the polynomial expression
y_fitted = p(x)     # computes the polynomial at the x points
# use red circles for data points and a blue line for the polynomial
plot(x, y, 'ro', x, y_fitted, 'b-',
     legend=('data', 'fitted polynomial of degree %d' % deg))
```

a) Write a function `fit(x, y, deg)` that creates a plot of data in `x` and `y` arrays along with polynomial approximations of degrees collected in the list `deg` as explained above.

b) We want to call `fit` to make a plot of the density of water versus temperature and another plot of the density of air versus temperature. In both calls, use `deg=[1,2]` such that we can compare linear and quadratic approximations to the data.

c) From a visual inspection of the plots, can you suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature?
Filename: `fit_density_data.py`.

Exercise 5.18: Fit a polynomial to experimental data

Suppose we have measured the oscillation period T of a simple pendulum with a mass m at the end of a massless rod of length L. We have varied L and recorded the corresponding T value. The measurements are found in a file `src/plot/pendulum.dat`[10]. The first column in the file contains L values and the second column has the corresponding T values.

a) Plot L versus T using circles for the data points.

b) We shall assume that L as a function of T is a polynomial. Use the NumPy utilities `polyfit` and `poly1d`, as explained in Exercise 5.17, to fit polynomials of degree 1, 2, and 3 to the L and T data. Visualize the polynomial curves together with the experimental data. Which polynomial fits the measured data best?
Filename: `fit_pendulum_data.py`.

Exercise 5.19: Read acceleration data and find velocities

A file `src/plot/acc.dat`[11] contains measurements $a_0, a_1, \ldots, a_{n-1}$ of the acceleration of an object moving along a straight line. The measurement a_k is taken at time point $t_k = k\Delta t$, where Δt is the time spacing between the measurements. The purpose of the exercise is to load the acceleration data into a program and compute the velocity $v(t)$ of the object at some time t.

In general, the acceleration $a(t)$ is related to the velocity $v(t)$ through $v'(t) = a(t)$. This means that

$$v(t) = v(0) + \int_0^t a(\tau)d\tau\,. \tag{5.17}$$

If $a(t)$ is only known at some discrete, equally spaced points in time, $a_0, \ldots, a_{n-1}$ (which is the case in this exercise), we must compute the integral in (5.17) numerically, for example by the Trapezoidal rule:

$$v(t_k) \approx \Delta t\left(\frac{1}{2}a_0 + \frac{1}{2}a_k + \sum_{i=1}^{k-1} a_i\right), \quad 1 \le k \le n-1\,. \tag{5.18}$$

[10] `http://tinyurl.com/pwyasaa/plot/pendulum.dat`
[11] `http://tinyurl.com/pwyasaa/plot/acc.dat`

We assume $v(0) = 0$ so that also $v_0 = 0$.

Read the values $a_0, \ldots, a_{n-1}$ from file into an array, plot the acceleration versus time, and use (5.18) to compute one $v(t_k)$ value, where Δt and $k \geq 1$ are specified on the command line. Filename: `acc2vel_v1.py`.

Exercise 5.20: Read acceleration data and plot velocities

The task in this exercise is the same as in Exercise 5.19, except that we now want to compute $v(t_k)$ for all time points $t_k = k\Delta t$ and plot the velocity versus time. Now only Δt is given on the command line, and the $a_0, \ldots, a_{n-1}$ values must be read from file as in Exercise 5.19.

Hint. Repeated use of (5.18) for all k values is very inefficient. A more efficient formula arises if we add the area of a new trapezoid to the previous integral (see also Section A.1.7):

$$v(t_k) = v(t_{k-1}) + \int\limits_{t_{k-1}}^{t_k} a(\tau)d\tau \approx v(t_{k-1}) + \Delta t \frac{1}{2}(a_{k-1} + a_k), \qquad (5.19)$$

for $k = 1, 2, \ldots, n-1$, while $v_0 = 0$. Use this formula to fill an array `v` with velocity values.
Filename: `acc2vel.py`.

Exercise 5.21: Plot a trip's path and velocity from GPS coordinates

A GPS device measures your position at every s seconds. Imagine that the positions corresponding to a specific trip are stored as (x, y) coordinates in a file `src/plot/pos.dat`[12] with an x and y number on each line, except for the first line, which contains the value of s.

a) Plot the two-dimensional curve of corresponding to the data in the file.

Hint. Load s into a `float` variable and then the x and y numbers into two arrays. Draw a straight line between the points, i.e., plot the y coordinates versus the x coordinates.

b) Plot the velocity in x direction versus time in one plot and the velocity in y direction versus time in another plot.

[12] `http://tinyurl.com/pwyasaa/plot/pos.dat`

Hint. If $x(t)$ and $y(t)$ are the coordinates of the positions as a function of time, we have that the velocity in x direction is $v_x(t) = dx/dt$, and the velocity in y direction is $v_y = dy/dt$. Since x and y are only known for some discrete times, $t_k = ks$, $k = 0, \ldots, n-1$, we must use numerical differentiation. A simple (forward) formula is

$$v_x(t_k) \approx \frac{x(t_{k+1}) - x(t_k)}{s}, \quad v_y(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{s}, \quad k = 0, \ldots, n-2\,.$$

Compute arrays `vx` and `vy` with velocities based on the formulas above for $v_x(t_k)$ and $v_y(t_k)$, $k = 0, \ldots, n-2$.
Filename: `position2velocity.py`.

Exercise 5.22: Vectorize the Midpoint rule for integration

The Midpoint rule for approximating an integral can be expressed as

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(a - \frac{1}{2}h + ih), \tag{5.20}$$

where $h = (b-a)/n$.

a) Write a function `midpointint(f, a, b, n)` to compute Midpoint rule. Use a plain Python `for` loop to implement the sum.

b) Make a vectorized implementation of the Midpoint rule where you compute the sum by Python's built-in function `sum`.

c) Make another vectorized implementation of the Midpoint rule where you compute the sum by the `sum` function in the `numpy` package.

d) Organize the three implementations above in a module file `midpoint_vec.py`.

e) Start IPython, import the functions from `midpoint_vec.py`, define some Python implementation of a mathematical function $f(x)$ to integrate, and use the `%timeit` feature of IPython to measure the efficiency of the three alternative implementations.

Hint. The `%timeit` feature is described in Section H.5.1.
Filename: `midpoint_vec.py`.

Remarks. The lesson learned from the experiments in e) is that `numpy.sum` is much more efficient than Python's built-in function `sum`. Vectorized implementations must always make use of `numpy.sum` to compute sums.

Exercise 5.23: Implement Lagrange's interpolation formula

Imagine we have $n+1$ measurements of some quantity y that depends on x: $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. We may think of y as a function of x and ask what y is at some arbitrary point x not coinciding with any of the points $x_0, \ldots, x_n$. It is not clear how y varies between the measurement points, but we can make assumptions or models for this behavior. Such a problem is known as *interpolation*.

One way to solve the interpolation problem is to fit a continuous function that goes through all the $n+1$ points and then evaluate this function for any desired x. A candidate for such a function is the polynomial of degree n that goes through all the points. It turns out that this polynomial can be written

$$p_L(x) = \sum_{k=0}^{n} y_k L_k(x), \tag{5.21}$$

where

$$L_k(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}\,. \tag{5.22}$$

The $\prod$ notation corresponds to $\sum$, but the terms are multiplied. For example,

$$\prod_{i=0, i\neq k}^{n} x_i = x_0 x_1 \cdots x_{k-1} x_{k+1} \cdots x_n\,.$$

The polynomial $p_L(x)$ is known as Lagrange's interpolation formula, and the points $(x_0, y_0), \ldots, (x_n, y_n)$ are called interpolation points.

a) Make functions `p_L(x, xp, yp)` and `L_k(x, k, xp, yp)` that evaluate $p_L(x)$ and $L_k(x)$ by (5.21) and (5.22), respectively, at the point `x`. The arrays `xp` and `yp` contain the x and y coordinates of the $n+1$ interpolation points, respectively. That is, `xp` holds $x_0, \ldots, x_n$, and `yp` holds $y_0, \ldots, y_n$.

b) To verify the program, we observe that $L_k(x_k) = 1$ and that $L_k(x_i) = 0$ for $i \neq k$, implying that $p_L(x_k) = y_k$. That is, the polynomial p_L goes through all the points $(x_0, y_0), \ldots, (x_n, y_n)$. Write a function `test_p_L(xp, yp)` that computes $|p_L(x_k) - y_k|$ at all the interpolation points (x_k, y_k) and checks that the value is approximately zero. Call `test_p_L` with `xp` and `yp` corresponding to 5 equally spaced points along the curve $y = \sin(x)$ for $x \in [0, \pi]$. Thereafter, evaluate $p_L(x)$ for an x in the middle of two interpolation points and compare the value of $p_L(x)$ with the exact one.
Filename: `Lagrange_poly1.py`.

Exercise 5.24: Plot Lagrange's interpolating polynomial

a) Write a function `graph(f, n, xmin, xmax, resolution=1001)` for plotting $p_L(x)$ in Exercise 5.23, based on interpolation points taken from some mathematical function $f(x)$ represented by the argument `f`. The argument `n` denotes the number of interpolation points sampled from the $f(x)$ function, and `resolution` is the number of points between `xmin` and `xmax` used to plot $p_L(x)$. The x coordinates of the `n` interpolation points can be uniformly distributed between `xmin` and `xmax`. In the graph, the interpolation points $(x_0, y_0), \ldots, (x_n, y_n)$ should be marked by small circles. Test the `graph` function by choosing 5 points in $[0, \pi]$ and `f` as $\sin x$.

b) Make a module `Lagrange_poly2` containing the `p_L`, `L_k`, `test_p_L`, and `graph` functions. The call to `test_p_L` described in Exercise 5.23 and the call to `graph` described above should appear in the module's test block.

Hint. Section 4.9 describes how to make a module. In particular, a test block is explained in Section 4.9.3, test functions like `test_p_L` are demonstrated in Section 4.9.4 and also in Section 3.4.2, and how to combine `test_p_L` and `graph` calls in the test block is exemplified in Section 4.9.5.
Filename: `Lagrange_poly2.py`.

Exercise 5.25: Investigate the behavior of Lagrange's interpolating polynomials

Unfortunately, the polynomial $p_L(x)$ defined and implemented in Exercise 5.23 can exhibit some undesired oscillatory behavior that we shall explore graphically in this exercise. Call the `graph` function from Exercise 5.24 with $f(x) = |x|$, $x \in [-2, 2]$, for $n = 2, 4, 6, 10$. All the graphs of $p_L(x)$ should appear in the same plot for comparison. In addition, make a new figure with calls to `graph` for $n = 13$ and $n = 20$. All the code necessary for solving this exercise should appear in some separate program file, which imports the `Lagrange_poly2` module made in Exercise 5.24.
Filename: `Lagrange_poly2b.py`.

Remarks. The purpose of the $p_L(x)$ function is to compute (x, y) between some given (often measured) data points $(x_0, y_0), \ldots, (x_n, y_n)$. We see from the graphs that for a small number of interpolation points, $p_L(x)$ is quite close to the curve $y = |x|$ we used to generate the data points, but as n increases, $p_L(x)$ starts to oscillate, especially toward the end points (x_0, y_0) and (x_n, y_n). Much research has historically been focused on methods that do not result in such strange oscillations when fitting a polynomial to a set of points.

Exercise 5.26: Plot a wave packet

The function

$$f(x,t) = e^{-(x-3t)^2} \sin\left(3\pi(x-t)\right) \tag{5.23}$$

describes for a fixed value of t a wave localized in space. Make a program that visualizes this function as a function of x on the interval $[-4, 4]$ when $t = 0$. Filename: `plot_wavepacket.py`.

Exercise 5.27: Judge a plot

Assume you have the following program for plotting a parabola:

```
import numpy as np
x = np.linspace(0, 2, 20)
y = x*(2 - x)
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

Then you switch to the function $\cos(18\pi x)$ by altering the computation of `y` to `y = cos(18*pi*x)`. Judge the resulting plot. Is it correct? Display the $\cos(18\pi x)$ function with 1000 points in the same plot. Filename: `judge_plot.py`.

Exercise 5.28: Plot the viscosity of water

The viscosity of water, μ, varies with the temperature T (in Kelvin) according to

$$\mu(T) = A \cdot 10^{B/(T-C)}, \tag{5.24}$$

where $A = 2.414 \cdot 10^{-5}$ Pa s, $B = 247.8$ K, and $C = 140$ K. Plot $\mu(T)$ for T between 0 and 100 degrees Celsius. Label the x axis with 'temperature (C)' and the y axis with 'viscosity (Pa s)'. Note that T in the formula for μ must be in Kelvin. Filename: `water_viscosity.py`.

Exercise 5.29: Explore a complicated function graphically

The wave speed c of water surface waves depends on the length λ of the waves. The following formula relates c to λ:

$$c(\lambda) = \sqrt{\frac{g\lambda}{2\pi}\left(1 + s\frac{4\pi^2}{\rho g \lambda^2}\right) \tanh\left(\frac{2\pi h}{\lambda}\right)}\,. \tag{5.25}$$

Here, g is the acceleration of gravity (9.81 m/s^2), s is the air-water surface tension ($7.9 \cdot 10^{-2}$ N/m) , ρ is the density of water (can be taken as

1000 kg/m^3), and h is the water depth. Let us fix h at 50 m. First make a plot of $c(\lambda)$ (in m/s) for small λ (0.001 m to 0.1 m). Then make a plot $c(\lambda)$ for larger λ (1 m to 2 km. Filename: `water_wave_velocity.py`.

Exercise 5.30: Plot Taylor polynomial approximations to $\sin x$

The sine function can be approximated by a polynomial according to the following formula:

$$\sin x \approx S(x;n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}\,. \tag{5.26}$$

The expression $(2j+1)!$ is the factorial (`math.factorial` can compute this quantity). The error in the approximation $S(x;n)$ decreases as n increases and in the limit we have that $\lim_{n\rightarrow\infty} S(x;n) = \sin x$. The purpose of this exercise is to visualize the quality of various approximations $S(x;n)$ as n increases.

a) Write a Python function `S(x, n)` that computes $S(x;n)$. Use a straightforward approach where you compute each term as it stands in the formula, i.e., $(-1)^j x^{2j+1}$ divided by the factorial $(2j+1)!$. (We remark that Exercise A.14 outlines a much more efficient computation of the terms in the series.)

b) Plot $\sin x$ on $[0, 4\pi]$ together with the approximations $S(x;1)$, $S(x;2)$, $S(x;3)$, $S(x;6)$, and $S(x;12)$.
Filename: `plot_Taylor_sin.py`.

Exercise 5.31: Animate a wave packet

Display an animation of the function $f(x,t)$ in Exercise 5.26 by plotting f as a function of x on $[-6,6]$ for a set of t values in $[-1,1]$. Also make an animated GIF file.

Hint. A suitable resolution can be 1000 intervals (1001 points) along the x axis, 60 intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use the recipe in Section 5.3.4 and remember to remove the family of old plot files in the beginning of the program.
Filename: `plot_wavepacket_movie.py`.

Exercise 5.32: Animate a smoothed Heaviside function

Visualize the smoothed Heaviside function $H_\epsilon(x)$, defined in (3.25), as an animation where ϵ starts at 2 and then goes to zero. Filename: `smoothed_Heaviside_movie.py`.

Exercise 5.33: Animate two-scale temperature variations

We consider temperature oscillations in the ground as addressed in Section 5.8.2. Now we want to visualize daily and annual variations. Let A_1 be the amplitude of annual variations and A_2 the amplitude of the day/night variations. Let also $P_1 = 365$ days and $P_2 = 24$ h be the periods of the annual and the daily oscillations. The temperature at time t and depth z is then given by

$$T(z,t) = T_0 + A_1 e^{-a_1 z}\sin(\omega_1 t - a_1 z) + A_2 e^{-a_2 z}\sin(\omega_2 t - a_2 z), \quad (5.27)$$

where

$$\begin{aligned}\omega_1 &= 2\pi P_1,\\ \omega_2 &= 2\pi P_2,\\ a_1 &= \sqrt{\frac{\omega_1}{2k}},\\ a_2 &= \sqrt{\frac{\omega_2}{2k}}.\end{aligned}$$

Choose $k = 10^{-6}$ m^2/s, $A_1 = 15$ C, $A_2 = 7$ C, and the resolution Δt as $P_2/10$. Modify the `heatwave.py` program in order to animate this new temperature function. Filename: `heatwave2.py`.

Remarks. We assume in this problem that the temperature T equals the reference temperature T_0 at $t = 0$, resulting in a sine variation rather than the cosine variation in (5.13).

Exercise 5.34: Use non-uniformly distributed coordinates for visualization

Watching the animation in Exercise 5.33 reveals that there are rapid oscillations in a small layer close to $z = 0$. The variations away from $z = 0$ are much smaller in time and space. It would therefore be wise to use more z coordinates close to $z = 0$ than for larger z values. Given a set $x_0 < x_1 < \cdots < x_n$ of uniformly spaced coordinates in $[a, b]$, we can compute new coordinates $\bar{x}_i$, stretched toward $x = a$, by the formula

$$\bar{x}_i = a + (b-a)\left(\frac{x_i - a}{b - a}\right)^s,$$

for some $s > 1$. In the present example, we can use this formula to stretch the z coordinates to the left.

a) Experiment with $s \in [1.2, 3]$ and few points (say 15) and visualize the curve as a line with circles at the points so that you can easily see the distribution of points toward the left end. Identify a suitable value of s.

b) Run the animation with no circles and (say) 501 points with the found s value.
Filename: `heatwave2a.py`.

Exercise 5.35: Animate a sequence of approximations to π

Exercise 3.13 outlines an idea for approximating π as the length of a polygon inside the circle. Wrap the code from Exercise 3.13 in a function `pi_approx(N)`, which returns the approximation to π using a polygon with $N + 1$ equally distributed points. The task of the present exercise is to visually display the polygons as a movie, where each frame shows the polygon with $N + 1$ points together with the circle and a title reflecting the corresponding error in the approximate value of π. The whole movie arises from letting N run through $4, 5, 6, \ldots, K$, where K is some (large) prescribed value. Let there be a pause of 0.3 s between each frame in the movie. By playing the movie you will see how the polygons move closer and closer to the circle and how the approximation to π improves.
Filename: `pi_polygon_movie.py`.

Exercise 5.36: Animate a planet's orbit

A planet's orbit around a star has the shape of an ellipse. The purpose of this exercise is to make an animation of the movement along the orbit. One should see a small disk, representing the planet, moving along an elliptic curve. An evolving solid line shows the development of the planet's orbit as the planet moves and the title displays the planet's instantaneous velocity magnitude. As a test, run the special case of a circle and verify that the magnitude of the velocity remains constant as the planet moves.

Hint 1. The points (x, y) along the ellipse are given by the expressions

$$x = a\cos(\omega t), \quad y = b\sin(\omega t),$$

where a is the semi-major axis of the ellipse, b is the semi-minor axis, ω is an angular velocity of the planet around the star, and t denotes time. One complete orbit corresponds to $t \in [0, 2\pi/\omega]$. Let us discretize time into time points $t_k = k\Delta t$, where $\Delta t = 2\pi/(\omega n)$. Each frame in the movie corresponds to (x, y) points along the curve with t values $t_0, t_1, \ldots, t_i$, i representing the frame number ($i = 1, \ldots, n$).

Hint 2. The velocity vector is

$$(\frac{dx}{dt}, \frac{dy}{dt}) = (-\omega a \sin(\omega t), \omega b \cos(\omega t)),$$

and the magnitude of this vector becomes $\omega\sqrt{a^2 \sin^2(\omega t) + b^2 \cos^2(\omega t)}$.
Filename: `planet_orbit.py`.

Exercise 5.37: Animate the evolution of Taylor polynomials

A general series approximation (to a function) can be written as

$$S(x; M, N) = \sum_{k=M}^{N} f_k(x)\,.$$

For example, the Taylor polynomial of degree N for e^x equals $S(x; 0, N)$ with $f_k(x) = x^k/k!$. The purpose of the exercise is to make a movie of how $S(x; M, N)$ develops and improves as an approximation as we add terms in the sum. That is, the frames in the movie correspond to plots of $S(x; M, M)$, $S(x; M, M + 1)$, $S(x; M, M + 2)$, ..., $S(x; M, N)$.

a) Make a function

```
animate_series(fk, M, N, xmin, xmax, ymin, ymax, n, exact)
```

for creating such animations. The argument `fk` holds a Python function implementing the term $f_k(x)$ in the sum, `M` and `N` are the summation limits, the next arguments are the minimum and maximum x and y values in the plot, `n` is the number of x points in the curves to be plotted, and `exact` holds the function that $S(x)$ aims at approximating.

Hint. Here is some more information on how to write the `animate_series` function. The function must accumulate the $f_k(x)$ terms in a variable s, and for each k value, s is plotted against x together with a curve reflecting the exact function. Each plot must be saved in a file, say with names `tmp_0000.png`, `tmp_0001.png`, and so on (these filenames can be generated by `tmp_%04d.png`, using an appropriate counter). Use the `movie` function to combine all the plot files into a movie in a desired movie format.

In the beginning of the `animate_series` function, it is necessary to remove all old plot files of the form `tmp_*.png`. This can be done by the `glob` module and the `os.remove` function as exemplified in Section 5.3.4.

b) Call the `animate_series` function for the Taylor series for $\sin x$, where $f_k(x) = (-1)^k x^{2k+1}/(2k + 1)!$, and $x \in [0, 13\pi]$, $M = 0$, $N = 40$, $y \in [-2, 2]$.

c) Call the `animate_series` function for the Taylor series for e^{-x}, where $f_k(x) = (-x)^k/k!$, and $x \in [0, 15]$, $M = 0$, $N = 30$, $y \in [-0.5, 1.4]$.
Filename: `animate_Taylor_series.py`.

Exercise 5.38: Plot the velocity profile for pipeflow

A fluid that flows through a (very long) pipe has zero velocity on the pipe wall and a maximum velocity along the centerline of the pipe. The velocity v varies through the pipe cross section according to the following formula:

$$v(r) = \left(\frac{\beta}{2\mu_0}\right)^{1/n} \frac{n}{n+1}\left(R^{1+1/n} - r^{1+1/n}\right), \tag{5.28}$$

where R is the radius of the pipe, β is the pressure gradient (the force that drives the flow through the pipe), μ_0 is a viscosity coefficient (small for air, larger for water and even larger for toothpaste), n is a real number reflecting the viscous properties of the fluid ($n = 1$ for water and air, $n < 1$ for many modern plastic materials), and r is a radial coordinate that measures the distance from the centerline ($r = 0$ is the centerline, $r = R$ is the pipe wall).

a) Make a Python function that evaluates $v(r)$.

b) Plot $v(r)$ as a function of $r \in [0, R]$, with $R = 1$, $\beta = 0.02$, $\mu_0 = 0.02$, and $n = 0.1$.

c) Make an animation of how the $v(r)$ curves varies as n goes from 1 and down to 0.01. Because the maximum value of $v(r)$ decreases rapidly as n decreases, each curve can be normalized by its $v(0)$ value such that the maximum value is always unity.
Filename: `plot_velocity_pipeflow.py`.

Exercise 5.39: Plot sum-of-sines approximations to a function

Exercise 3.15 defines the approximation $S(t; n)$ to a function $f(t)$. Plot $S(t; 1)$, $S(t; 3)$, $S(t; 20)$, $S(t; 200)$, and the exact $f(t)$ function in the same plot. Use $T = 2\pi$. Filename: `sinesum1_plot.py`.

Exercise 5.40: Animate the evolution of a sum-of-sine approximation to a function

First perform Exercise 5.39. A natural next step is to animate the evolution of $S(t; n)$ as n increases. Create such an animation and observe how the discontinuity in $f(t)$ is poorly approximated by $S(t; n)$, even when n grows large (plot $f(t)$ in each frame). This is a well-known deficiency, called Gibb's phenomenon, when approximating discontinuous functions by sine or cosine (Fourier) series. Filename: `sinesum1_movie.py`.

Exercise 5.41: Plot functions from the command line

For quickly getting a plot a function $f(x)$ for $x \in [x_{\text{min}}, x_{\text{max}}]$ it could be nice to a have a program that takes the minimum amount of information from the command line and produces a plot on the screen and saves the plot to a file `tmp.png`. The usage of the program goes as follows:

Terminal

```
plotf.py "f(x)" xmin xmax
```

A specific example is

Terminal

```
plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

Write the `plotf.py` program with as short code as possible (we leave it to Exercise 5.42 to test for valid input).

Hint. Make x coordinates from the second and third command-line arguments and then use `eval` (or `StringFunction` from `scitools.std`, see Sections 4.3.3 and 5.5.1) on the first argument.
Filename: `plotf.py`.

Exercise 5.42: Improve command-line input

Equip the program from Exercise 5.41 with tests on valid input on the command line. Also allow an optional fourth command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Filename: `plotf2.py`.

Exercise 5.43: Demonstrate energy concepts from physics

The vertical position $y(t)$ of a ball thrown upward is given by $y(t) = v_0 t - \frac{1}{2}gt^2$, where g is the acceleration of gravity and v_0 is the velocity at $t = 0$. Two important physical quantities in this context are the potential energy, obtained by doing work against gravity, and the kinetic energy, arising from motion. The potential energy is defined as $P = mgy$, where m is the mass of the ball. The kinetic energy is defined as $K = \frac{1}{2}mv^2$, where v is the velocity of the ball, related to y by $v(t) = y'(t)$.

Make a program that can plot $P(t)$ and $K(t)$ in the same plot, along with their sum $P + K$. Let $t \in [0, 2v_0/g]$. Read m and v_0 from the command line. Run the program with various choices of m and v_0 and observe that $P + K$ is always constant in this motion. (In fact, it turns out that $P + K$ is constant for a large class of motions, and this is a very important result in physics.) Filename: `energy_physics.py`.

Exercise 5.44: Plot a w-like function

Define mathematically a function that looks like the "w" character. Plot this function. Filename: `plot_w.py`.

Exercise 5.45: Plot a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.26. Make a Python function `plot_piecewise(data, xmax)` that draws a graph of the function, where `data` is the nested list explained in Exercise 3.26 and `xmax` is the maximum x coordinate. Use ideas from Section 5.4.1. Filename: `plot_piecewise_constant.py`.

Exercise 5.46: Vectorize a piecewise constant function

Consider the piecewise constant function defined in Exercise 3.26. Make a vectorized implementation `piecewise_constant_vec(x, data, xmax)` of such a function, where `x` is an array.

Hint. You can use ideas from the `Nv1` function in Section 5.5.3. However, since the number of intervals is not known, it is necessary to store the various intervals and conditions in lists.
Filename: `piecewise_constant_vec.py`.

Remarks. Plotting the array returned from `piecewise_constant_vec` faces the same problems as encountered in Section 5.4.1. It is better to make a custom plotting function that simply draws straight horizontal lines in each interval (Exercise 5.45).

Exercise 5.47: Visualize approximations in the Midpoint integration rule

Consider the midpoint rule for integration from Exercise 3.7. Use Matplotlib to make an illustration of the midpoint rule as shown to the left in Figure 5.12.

The $f(x)$ function used in Figure 5.12 is

$$f(x) = x(12 - x) + \sin(\pi x), \quad x \in [0, 10]\,.$$

Hint. Look up the documentation of the Matplotlib function `fill_between` and use this function to create the filled areas between $f(x)$ and the approximating rectangles.

Note that the `fill_between` requires the two curves to have the same number of points. For accurate visualization of $f(x)$ you need quite many x coordinates, and the rectangular approximation to $f(x)$ must be drawn using the same set of x coordinates.
Filename: `viz_midpoint.py`.

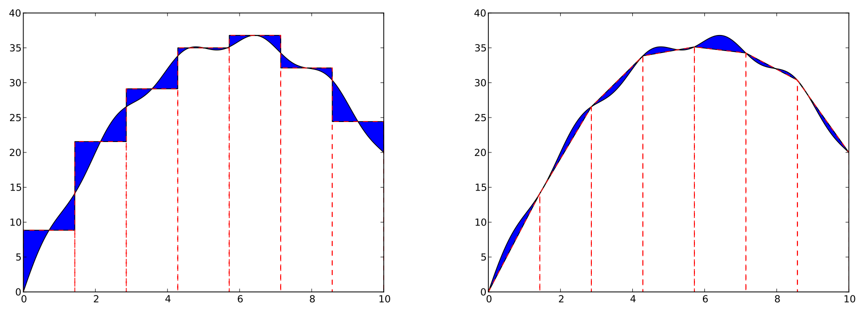


Fig. 5.12 Visualization of numerical integration rules, with the Midpoint rule to the left and the Trapezoidal rule to the right. The filled areas illustrate the deviations in the approximation of the area under the curve.

Exercise 5.48: Visualize approximations in the Trapezoidal integration rule

Redo Exercise 5.47 for the Trapezoidal rule from Exercise 3.6 to produce the graph shown to the right in Figure 5.12. Filename: `viz_trapezoidal.py`.

Exercise 5.49: Experience overflow in a function

We are give the mathematical function

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}},$$

where μ is a parameter.

a) Make a Python function `v(x, mu=1E-6, exp=math.exp)` for calculating the formula for $v(x)$ using `exp` as a possibly user-given exponential function. Let the `v` function return the nominator and denominator in the formula as well as the fraction.

b) Call the `v` function for various `x` values between 0 and 1 in a `for` loop, let `mu` be `1E-3`, and have an inner `for` loop over two different `exp` functions: `math.exp` and `numpy.exp`. The output will demonstrate how the denominator is subject to overflow and how difficult it is to calculate this function on a computer.

c) Plot $v(x)$ for $\mu = 1, 0.01, 0.001$ on $[0, 1]$ using 10,000 points to see what the function looks like.

d) Convert `x` and `eps` to a higher precision representation of real numbers, with the aid of the NumPy type `float96`, before calling `v`:

```
import numpy
x = numpy.float96(x); mu = numpy.float96(e)
```

Repeat point b) with these type of variables and observe how much better results we get with `float96` compared with the standard `float` value, which is `float64` (the number reflects the number of bits in the machine's representation of a real number).

e) Call the `v` function with `x` and `mu` as `float32` variables and report how the function now behaves.
Filename: `boundary_layer_func1.py`.

Remarks. When an object (ball, car, airplane) moves through the air, there is a very, very thin layer of air close to the object's surface where the air velocity varies dramatically, from the same value as the velocity of the object at the object's surface to zero a few centimeters away. This layer is called a *boundary layer*. The physics in the boundary layer is important for air resistance and cooling/heating of objects. The change in velocity in the boundary layer is quite abrupt and can be modeled by the functiion $v(x)$, where $x = 1$ is the object's surface, and $x = 0$ is some distance away where one cannot notice any wind velocity v because of the passing object ($v = 0$). The wind velocity coincides with the velocity of the object at $x = 1$, here set to $v = 1$. The parameter μ is very small and related to the viscosity of air. With a small value of μ, it becomes difficult to calculate $v(x)$ on a computer. The exercise demonstrates the difficulties and provides a remedy.

Exercise 5.50: Apply a function to a rank 2 array

Let A be the two-dimensional array

$$\begin{bmatrix} 0 & 2 & -1 \\ -1 & -1 & 0 \\ 0 & 5 & 0 \end{bmatrix}$$

Apply the function f from Exercise 5.5 to each element in A. Then calculate the result of the array expression `A**3 + A*exp(A) + 1`, and demonstrate that the end result of the two methods are the same.

Exercise 5.51: Explain why array computations fail

The following loop computes the array `y` from `x`:

```
>>> import numpy as np
>>> x = np.linspace(0, 1, 3)
>>> y = np.zeros(len(x))
>>> for i in range(len(x)):
...     y[i] = x[i] + 4
```

However, the alternative loop

```
>>> for xi, yi in zip(x, y):
...     yi = xi + 5
```

leaves `y` unchanged. Why? Explain in detail what happens in each pass of this loop and write down the contents of `xi`, `yi`, `x`, and `y` as the loop progresses.

```
        return 1
    else:
        return 0
```

We can now pass on `sort_names` to the `sorted` function to get a sequence that is sorted with respect to the last word in the students' names:

```
for name in sorted(data, sort_names):
    print '%s: %s' % (name, average_grade(data, name))
```

6.7 Exercises

Exercise 6.1: Make a dictionary from a table

The file `src/files/constants.txt`[5] contains a table of the values and the dimensions of some fundamental constants from physics. We want to load this table into a dictionary `constants`, where the keys are the names of the constants. For example, `constants['gravitational constant']` holds the value of the gravitational constant ($6.67259 \cdot 10^{-11}$) in Newton's law of gravitation. Make a function that reads and interprets the text in the file, and finally returns the dictionary. Filename: `fundamental_constants.py`.

Exercise 6.2: Explore syntax differences: lists vs. dicts

Consider this code:

```
t1 = {}
t1[0] = -5
t1[1] = 10.5
```

Explain why the lines above work fine while the ones below do not:

```
t2 = []
t2[0] = -5
t2[1] = 10.5
```

What must be done in the last code snippet to make it work properly? Filename: `list_vs_dict.py`.

Exercise 6.3: Use string operations to improve a program

Consider the program `density.py` from Section 6.1.5. One problem we face when implementing this program is that the name of the substance can contain one or two words, and maybe more words in a more comprehensive table. The purpose of this exercise is to use string operations to shorten the code and make it more general.

[5] `http://tinyurl.com/pwyasaa/files/constants.txt`

a) Make a Python function that lets `substance` consist of all the words but the last, using the `join` method in string objects to combine the words.

b) Observe that all the densities start in the same column. Write an alternative function that makes use of substring indexing to divide `line` into two parts.

Hint. Remember to strip the first part such that, e.g., the density of ice is obtained as `densities['ice']` and not `densities['ice ']`.

c) Make a test function that calls the two other functions and tests that they produce the same result.
Filename: `density_improved.py`.

Exercise 6.4: Interpret output from a program

The program `src/funcif/lnsum.py` produces, among other things, this output:

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

Redirect the output to a file (by `python lnsum.py > file`). Write a Python program that reads the file and extracts the numbers corresponding to `epsilon`, `exact error`, and `n`. Store the numbers in three arrays and plot `epsilon` and the `exact error` versus `n`. Use a logarithmic scale on the y axis.

Hint. The function `semilogy` is an alternative to `plot` and gives logarithmic scale on y axis.
Filename: `read_error.py`.

Exercise 6.5: Make a dictionary

Based on the stars data in Exercise 3.33, make a dictionary where the keys contain the names of the stars and the values correspond to the luminosity. Filename: `stars_data_dict1.py`.

Exercise 6.6: Make a nested dictionary

Store the data about stars from Exercise 3.33 in a nested dictionary such that we can look up the distance, the apparent brightness, and the luminosity of a star with name `N` by

```
stars[N]['distance']
stars[N]['apparent brightness']
stars[N]['luminosity']
```

Filename: `stars_data_dict2.py`.

Exercise 6.7: Make a nested dictionary from a file

The file `src/files/human_evolution.txt`[6] holds information about various human species and their height, weight, and brain volume. Make a program that reads this file and stores the tabular data in a nested dictionary `humans`. The keys in `humans` correspond to the specie name (e.g., `homo erectus`), and the values are dictionaries with keys for `height`, `weight`, `brain volume`, and `when` (the latter for when the specie lived). For example, `humans['homo neanderthalensis']['mass']` should equal `'55-70'`. Let the program write out the `humans` dictionary in a nice tabular form similar to that in the file. Filename: `humans.py`.

Exercise 6.8: Make a nested dictionary from a file

The viscosity μ of gases depends on the temperature. For some gases the following formula is relevant:

$$\mu(T) = \mu_0 \frac{T_0 - C}{T + C} \left(\frac{T}{T_0}\right)^{1.5},$$

where the values of the constants C, T_0, and μ_0 are found in the file `src/files/viscosity_of_gases.dat`[7]. The temperature is measured in Kelvin.

a) Load the file into a nested dictionary `mu_data` such that we can look up C, T_0, and μ_0 for a gas with name `name` by `mu_data[name][X]`, where `X` is `'C'` for C, `'T_0'` for T_0, and `'mu_0'` for μ_0.

b) Make a function `mu(T, gas, mu_data)` for computing $\mu(T)$ for a gas with name `gas` (according to the file) and information about constants C, T_0, and μ_0 in `mu_data`.

c) Plot $\mu(T)$ for air, carbon dioxide, and hydrogen with $T \in [223, 373]$. Filename: `viscosity_of_gases.py`.

Exercise 6.9: Compute the area of a triangle

The purpose of this exercise is to write an `area` function as in Exercise 3.11, but now we assume that the vertices of the triangle is stored

[6] `http://tinyurl.com/pwyasaa/files/human_evolution.txt`
[7] `http://tinyurl.com/pwyasaa/files/viscosity_of_gases.txt`

in a dictionary and not a list. The keys in the dictionary correspond to the vertex number (1, 2, or 3) while the values are 2-tuples with the x and y coordinates of the vertex. For example, in a triangle with vertices $(0,0)$, $(1,0)$, and $(0,2)$ the `vertices` argument becomes

```
{1: (0,0), 2: (1,0), 3: (0,2)}
```

Filename: `area_triangle_dict.py`.

Exercise 6.10: Compare data structures for polynomials

Write a code snippet that uses both a list and a dictionary to represent the polynomial $-\frac{1}{2} + 2x^{100}$. Print the list and the dictionary, and use them to evaluate the polynomial for $x = 1.05$.

Hint. You can apply the `poly1` and `poly2` functions from Section 6.1.3). Filename: `poly_repr.py`.

Exercise 6.11: Compute the derivative of a polynomial

A polynomial can be represented by a dictionary as explained in Section 6.1.3. Write a function `diff` for differentiating such a polynomial. The `diff` function takes the polynomial as a dictionary argument and returns the dictionary representation of the derivative. Here is an example of the use of the function `diff`:

```
>>> p = {0: -3, 3: 2, 5: -1}    # -3 + 2*x**3 - x**5
>>> diff(p)                     # should be 6*x**2 - 5*x**4
{2: 6, 4: -5}
```

Hint. Recall the formula for differentiation of polynomials:

$$\frac{d}{dx}\sum_{j=0}^{n} c_j x^j = \sum_{j=1}^{n} j c_j x^{j-1}\,. \tag{6.1}$$

This means that the coefficient of the x^{j-1} term in the derivative equals j times the coefficient of x^j term of the original polynomial. With `p` as the polynomial dictionary and `dp` as the dictionary representing the derivative, we then have `dp[j-1] = j*p[j]` for `j` running over all keys in `p`, except when `j` equals 0.
Filename: `poly_diff.py`.

Exercise 6.12: Specify functions on the command line

Explain what the following two code snippets do and give an example of how they can be used.

Hint. Read about the `StringFunction` tool in Section 4.3.3 and about a variable number of keyword arguments in Section H.4.

```
import sys
from scitools.StringFunction import StringFunction
parameters = {}
for prm in sys.argv[4:]:
    key, value = prm.split('=')
    parameters[key] = eval(value)
f = StringFunction(sys.argv[1], independent_variables=sys.argv[2],
                   **parameters)
var = float(sys.argv[3])
print f(var)
```

a)

```
import sys
from scitools.StringFunction import StringFunction
f = eval('StringFunction(sys.argv[1], ' + \
         'independent_variables=sys.argv[2], %s)' % \
         (', '.join(sys.argv[4:])))
var = float(sys.argv[3])
print f(var)
```

b)
Filename: `cml_functions.py`.

Exercise 6.13: Interpret function specifications

To specify arbitrary functions $f(x_1, x_2, \ldots; p_1, p_2, \ldots)$ with independent variables $x_1, x_2, \ldots$ and a set of parameters $p_1, p_2, \ldots$, we allow the following syntax on the command line or in a file:

```
<expression> is function of <list1> with parameter <list2>
```

where `<expression>` denotes the function formula, `<list1>` is a comma-separated list of the independent variables, and `<list2>` is a comma-separated list of name=value parameters. The part `with parameters <list2>` is omitted if there are no parameters. The names of the independent variables and the parameters can be chosen freely as long as the names can be used as Python variables. Here are four different examples of what we can specify on the command line using this syntax:

```
sin(x) is a function of x
sin(a*y) is a function of y with parameter a=2
sin(a*x-phi) is a function of x with parameter a=3, phi=-pi
exp(-a*x)*cos(w*t) is a function of t with parameter a=1,w=pi,x=2
```

Create a Python function that takes such function specifications as input and returns an appropriate `StringFunction` object. This object must be created from the function expression and the list of independent variables and parameters. For example, the last function specification above leads to the following `StringFunction` creation:

```
f = StringFunction('exp(-a*x)*cos(w*t)',
                   independent_variables=['t'],
                   a=1, w=pi, x=2)
```

Hint. Use string operations to extract the various parts of the string. For example, the expression can be split out by calling `split('is a function of')`. Typically, you need to extract `<expression>`, `<list1>`, and `<list2>`, and create a string like

```
StringFunction(<expression>, independent_variables=[<list1>],
               <list2>)
```

and sending it to `eval` to create the object.
Filename: `text2func.py`.

Exercise 6.14: Compare average temperatures in cities

The tarfile `src/misc/city_temp.tar.gz`[8] contains a set of files with temperature data for a large number of cities around the world. The files are in text format with four columns, containing the month number, the date, the year, and the temperature, respectively. Missing temperature observations are represented by the value -99. The mapping between the names of the text files and the names of the cities are defined in an HTML file `citylistWorld.htm`.

a) Write a function that can read the `citylistWorld.htm` file and create a dictionary with mapping between city and filenames.

b) Write a function that takes this dictionary and a city name as input, opens the corresponding text file, and loads the data into an appropriate data structure (dictionary of arrays and city name is a suggestion).

c) Write a function that can take a number of data structures and the corresponding city names to create a plot of the temperatures over a certain time period.
Filename: `temperature_data.py`.

Exercise 6.15: Generate an HTML report with figures

The goal of this exercise is to let a program write a report in HTML format containing the solution to Exercise 5.31. First, include the program from that exercise, with additional explaining text if necessary. Program code can be placed inside `<pre>` and `</pre>` tags. Second, insert three plots of the $f(x,t)$ function for three different t values (find suitable t values that illustrate the displacement of the wave packet). Third, add an animated GIF file with the movie of $f(x,t)$. Insert headlines (`<h1>` tags) wherever appropriate. Filename: `wavepacket_report.py`.

[8] `http://tinyurl.com/pwyasaa/misc/city_temp.tar.gz`

Exercise 6.16: Allow different types for a function argument

Consider the family of `find_consensus_v*` functions from Section 6.5.2. The different versions work on different representations of the frequency matrix. Make a unified `find_consensus` function that accepts different data structures for the `frequency_matrix`. Test on the type of data structure and perform the necessary actions. Filename: `find_consensus.py`.

Exercise 6.17: Make a function more robust

Consider the function `get_base_counts(dna)` from Section 6.5.3, which counts how many times `A`, `C`, `G`, and `T` appears in the string `dna`:

```
def get_base_counts(dna):
    counts = {'A': 0, 'T': 0, 'G': 0, 'C': 0}
    for base in dna:
        counts[base] += 1
    return counts
```

Unfortunately, this function crashes if other letters appear in `dna`. Write an enhanced function `get_base_counts2` which solves this problem. Test it on a string like `'ADLSTTLLD'`. Filename: `get_base_counts2.py`.

Exercise 6.18: Find proportion of bases inside/outside exons

Consider the lactase gene as described in Sections 6.5.4 and 6.5.5. What is the proportion of base A inside and outside exons of the lactase gene?

Hint. Write a function `get_exons`, which returns all the substrings of the exon regions concatenated. Also write a function `get_introns`, which returns all the substrings between the exon regions concatenated. The function `get_base_frequencies` from Section 6.5.3 can then be used to analyze the frequencies of bases A, C, G, and T in the two strings. Filename: `prop_A_exons.py`.

Here, a 20% uncertainty in R gives almost 60% uncertainty in V, and the mean of the V interval is significantly different from computing the volume with the mean of R.

The complete code of class `IntervalMath` is found in `IntervalMath.py`. Compared to the implementations shown above, the real implementation in the file employs some ingenious constructions and help methods to save typing and repeating code in the special methods for arithmetic operations.

7.8 Exercises

Exercise 7.1: Make a function class

Make a class `F` that implements the function

$$f(x; a, w) = e^{-ax} \sin(wx)\,.$$

A `value(x)` method computes values of f, while a and w are class attributes. Test the class in an interactive session:

```
>>> from F import F
>>> f = F(a=1.0, w=0.1)
>>> from math import pi
>>> print f.value(x=pi)
0.013353835137
>>> f.a = 2
>>> print f.value(pi)
0.00057707154012
```

Filename: `F.py`.

Exercise 7.2: Add an attribute to a class

Add an attribute `transactions` to the `Account` class from Section 7.2.1. The new attribute counts the number of transactions done in the `deposit` and `withdraw` methods. Print the total number of transactions in the `dump` method. Write a test function `test_Account()` for testing that the implementation of the extended class `Account` is correct. Filename: `Account2.py`.

Exercise 7.3: Add functionality to a class

In class `Account` from Section 7.2.1, introduce a list `transactions`, where each element holds a dictionary with the amount of a transaction and the point of time the transaction took place. Remove the `balance` attribute and use instead the `transactions` list to compute the balance in `get_balance`. Print out a nicely formatted table of all transactions, their amounts, and their time in a method `print_transactions`.

Hint. Use the `time` module to get the date and local time.
Filename: `Account3.py`.

Exercise 7.4: Make classes for a rectangle and a triangle

The purpose of this exercise is to create classes like class `Circle` from Section 7.2.3 for representing other geometric figures: a rectangle with width W, height H, and lower left corner (x_0, y_0); and a general triangle specified by its three vertices (x_0, y_0), (x_1, y_1), and (x_2, y_2) as explained in Exercise 3.11. Provide three methods: `__init__` (to initialize the geometric data), `area`, and `perimeter`. Write test functions `test_Rectangle()` and `test_Triangle()` for checking that the results produced by `area` and `perimeter` coincide with exact values within a small tolerance. Filename: `geometric_shapes.py`.

Exercise 7.5: Make a class for quadratic functions

Consider a quadratic function $f(x; a, b, c) = ax^2 + bx + c$. Make a class `Quadratic` for representing f, where a, b, and c are attributes, and the methods are

- `value` for computing a value of f at a point x,
- `table` for writing out a table of x and f values for n x values in the interval $[L, R]$,
- `roots` for computing the two roots.

The file with class `Quadratic` and corresponding demonstrations and/or tests should be organized as a module such that other programs can do a `from Quadratic import Quadratic` to use the class. Filename: `Quadratic.py`.

Exercise 7.6: Make a class for straight lines

Make a class `Line` whose constructor takes two points `p1` and `p2` (2-tuples or 2-lists) as input. The line goes through these two points (see function `line` in Section 3.1.11 for the relevant formula of the line). A `value(x)` method computes a value on the line at the point `x`. Also make a function `test_Line()` for verifying the implementation. Here is a demo in an interactive session:

```
>>> from Line import Line, test_Line
>>> line = Line((0,-1), (2,4))
>>> print line.value(0.5), line.value(0), line.value(1)
0.25 -1.0 1.5
>>> test_Line()
```

Filename: `Line.py`.

Exercise 7.7: Flexible handling of function arguments

The constructor in class `Line` in Exercise 7.6 takes two points as arguments. Now we want to have more flexibility in the way we specify a straight line: we can give two points, a point and a slope, or a slope and the line's interception with the y axis. Write this extended class and a test function for checking that the increased flexibility does work.

Hint. Let the constructor take two arguments `p1` and `p2` as before, and test with `isinstance` whether the arguments are `float` versus `tuple` or `list` to determine what kind of data the user supplies:

```
if isinstance(p1, (tuple,list)) and isinstance(p2, (float,int)):
    # p1 is a point and p2 is slope
    self.a = p2
    self.b = p1[1] - p2*p1[0]
elif ...
```

Filename: `Line2.py`.

Exercise 7.8: Wrap functions in a class

The purpose of this exercise is to make a class interface to an already existing set of functions implementing Lagrange's interpolation method from Exercise 5.23. We want to construct a class `LagrangeInterpolation` with a typical usage like:

```
import numpy as np
# Compute some interpolation points along y=sin(x)
xp = np.linspace(0, np.pi, 5)
yp = np.sin(xp)

# Lagrange's interpolation polynomial
p_L = LagrangeInterpolation(xp, yp)
x = 1.2
print 'p_L(%g)=%g' % (x, p_L(x)),
print 'sin(%g)=%g' % (x, np.sin(x))
p_L.plot()   # show graph of p_L
```

The `plot` method visualizes $p_L(x)$ for x between the first and last interpolation point (`xp[0]` and `xp[-1]`). In addition to writing the class itself, you should write code to verify the implementation.

Hint. The class does not need much code as it can call the functions `p_L` from Exercise 5.23 and `graph` from Exercise 5.24, available in the `Lagrange_poly2` module made in the latter exercise.
Filename: `Lagrange_poly3.py`.

Exercise 7.9: Flexible handling of function arguments

Instead of manually computing the interpolation points, as demonstrated in Exercise 7.8, we now want the constructor in class

LagrangeInterpolation to also accept some Python function f(x) for computing the interpolation points. Typically, we would like to write this code:

```
from numpy import exp, sin, pi

def myfunction(x):
    return exp(-x/2.0)*sin(x)

p_L = LagrangeInterpolation(myfunction, x=[0, pi], n=11)
```

With such a code, $n = 11$ uniformly distributed x points between 0 and π are computed, and the corresponding y values are obtained by calling myfunction. The Lagrange interpolation polynomial is then constructed from these points. Note that the previous types of calls, LangrangeInterpolation(xp, yp), must still be valid.

Hint. The constructor in class LagrangeInterpolation must now accept two different sets of arguments: xp, yp vs. f, x, n. You can use the isinstance(a, t) function to test if object a is of type t. Declare the constructor with three arguments arg1, arg2, and arg3=None. Test if arg1 and arg2 are arrays (isinstance(arg1, numpy.ndarray)), and in that case, set xp=arg1 and yp=arg2. On the other hand, if arg1 is a function (callable(arg1) is True), arg2 is a list or tuple (isinstance(arg2, (list,tuple))), and arg3 is an integer, set f=arg1, x=arg2, and n=arg3.
Filename: Lagrange_poly4.py.

Exercise 7.10: Deduce a class implementation

Write a class Hello that behaves as illustrated in the following session:

```
>>> a = Hello()
>>> print a('students')
Hello, students!
>>> print a
Hello, World!
```

Filename: Hello.py.

Exercise 7.11: Implement special methods in a class

Modify the class from Exercise 7.1 such that the following interactive session can be run:

```
>>> from F import F
>>> f = F(a=1.0, w=0.1)
>>> from math import pi
>>> print f(x=pi)
0.013353835137
```

```
>>> f.a = 2
>>> print f(pi)
0.00057707154012
>>> print f
exp(-a*x)*sin(w*x)
```

Filename: `F2.py`.

Exercise 7.12: Make a class for summation of series

The task in this exercise is to calculate a sum $S(x) = \sum_{k=M}^{N} f_k(x)$, where $f_k(x)$ is some user-given formula for the terms in the sum. The following snippet demonstrates the typical use and functionality of a class `Sum` for computing $S(x) = \sum_{k=0}^{N}(-x)^k$:

```
def term(k, x):
    return (-x)**k

S = Sum(term, M=0, N=3)
x = 0.5
print S(x)
print S.term(k=4)
```

The latter statement prints the term $(-x)^4$.

a) Implement class `Sum` such that the code snippet above works.

b) Implement a test function `test_Sum()` for verifying the results of the various methods in class `Sum` for a specific choice of $f_k(x)$.

c) Apply class `Sum` to compute the Taylor polynomial approximation to $\sin x$ for $x = \pi$ and some chosen x and N.
Filename: `Sum.py`.

Exercise 7.13: Apply a numerical differentiation class

Isolate class `Derivative` from Section 7.3.2 in a module file. Also isolate class `Y` from Section 7.1.2 in a module file. Make a program that imports class `Derivative` and class `Y` and applies the former to differentiate the function $y(t) = v_0 t - \frac{1}{2}gt^2$ represented by class `Y`. Compare the computed derivative with the exact value for $t = 0, \frac{1}{2}v_0/g, v_0/g$. Filenames: `dYdt.py`, `Derivative.py`, `Y.py`.

Exercise 7.14: Apply symbolic differentiation

Class `Derivative` from Section 7.3.2 applies numerical differentiation. With the aid of `sympy` we can quite easily offer exact differentiation. Extend class `Derivative` such that the differentiation is exact if the user supplies a `sympy` expression as argument to the constructor:

```
>>> def f(x):
        return x**3
...
>>> import sympy
>>> x = sympy.symbols('x')
>>> symbolic_formula = f(x)
>>> dfdx = Derivative(symbolic_formula)
>>> x = 2
>>> dfdx(x)
12
```

Hint. Introduce a test on the type of argument in the constructor in class `Derivative`: if `str(type(f))` starts with `sympy`, `f` is a `sympy` expression that we can differentiate with `sympy.diff` and turn the result into a plain Python with `sympy.lamdify`, see Section 1.7.1.
Filename: `Derivative_sympy.py`.

Exercise 7.15: Implement in-place += and -= operators

As alternatives to the `deposit` and `withdraw` methods in class `Account` class from Section 7.2.1, we could use the operation `+=` for `deposit` and `-=` for `withdraw`. Implement the `+=` and `-=` operators, a `__str__` method, and preferably a `__repr__` method in class `Account`. Write a `test_Account()` function to verify the implementation of all functionality in class `Account`.

Hint. The special methods `__iadd__` and `__isub__` implement the `+=` and `-=` operators, respectively. For instance, `a -= p` implies a call to `a.__isub__(p)`. One important feature of `__iadd__` and `__isub__` is that they must return `self` to work properly, see the documentation of these methods in Chapter 3 of the Python Language Reference[3].
Filename: `Account4.py`.

Exercise 7.16: Implement a class for numerical differentiation

A widely used formula for numerical differentiation of a function $f(x)$ takes the form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}\,. \tag{7.8}$$

This formula usually gives more accurate derivatives than (7.1) because it applies a centered, rather than a one-sided, difference.

The goal of this exercise is to use the formula (7.8) to automatically differentiate a mathematical function $f(x)$ implemented as a Python function `f(x)`. More precisely, the following code should work:

[3] `http://docs.python.org/2/reference/`

```
def f(x):
    return 0.25*x**4

df = Central(f)  # make function-like object df
# df(x) computes the derivative of f(x) approximately
x = 2
print 'df(%g)=%g' % (x, df(x))
print 'exact:',  x**3
```

a) Implement class `Central` and test that the code above works. Include an optional argument `h` to the constructor in class `Central` so that h in the approximation (7.8) can be specified.

b) Write a test function `test_Central()` to verify the implementation. Utilize the fact that the formula (7.8) is exact for quadratic polynomials.

c) Write a function `table(f, x, h=1E-5)` that prints a table of errors in the numerical derivative (7.8) applied to a function `f` at some points `x`. The argument `f` is a `sympy` expression for a function. This `f` object can be transformed to a Python function and fed to the constructor of class `Central`, and `f` can be used to compute the exact derivative symbolically. The argument `x` is a list or array of points x, and `h` is the h in (7.8).

Hint. The following session demonstrates how `sympy` can differentiate a mathematical expression and turn the result into a Python function:

```
>>> import sympy
>>> x = sympy.Symbol('x')
>>> f_expr = 'x*sin(2*x)'
>>> df_expr = sympy.diff(f_expr)
>>> df_expr
2*x*cos(2*x) + sin(2*x)
>>> df = sympy.lambdify([x], df_expr)  # make Python function
>>> df(0)
0.0
```

d) Organize the file with the class and functions such that it can be used a module.
Filename: `Central.py`.

Exercise 7.17: Examine a program

Consider this program file for computing a backward difference approximation to the derivative of a function `f(x)`:

```
from math import *

class Backward:
    def __init__(self, f, h=e-9):
        self.f, self.h = f, h
    def __call__(self, x):
        h, f = self.h, self.f
        return (f(x) - f(x-h))/h  # finite difference
```

```
dsin = Backward(sin)
e = dsin(0) - cos(0); print 'error:', e
dexp = Backward(exp, h=e-7)
e = dexp(0) - exp(0); print 'error:', e
```

The output becomes

```
error: -1.00023355634
error: 371.570909212
```

Is the approximation that bad, or are there bugs in the program?

Exercise 7.18: Modify a class for numerical differentiation

Make the two attributes `h` and `f` of class `Derivative` from Section 7.3.2 protected as explained in Section 7.2.1. That is, prefix `h` and `f` with an underscore to tell users that these attributes should not be accessed directly. Add two methods `get_precision()` and `set_precision(h)` for reading and changing `h`. Filename: `Derivative_protected.py`.

Exercise 7.19: Make a class for the Heaviside function

a) Use a class to implement the discontinuous Heaviside function (3.24) from Exercise 3.23 and the smoothed continuous version (3.25) from Exercise 3.24 such that the following code works:

```
H = Heaviside()         # original discontinous Heaviside function
print H(0.1)
H = Heaviside(eps=0.8)  # smoothed continuous Heaviside function
print H(0.1)
```

b) Extend class `Heaviside` such that array arguments are allowed:

```
H = Heaviside()         # original discontinous Heaviside function
x = numpy.linspace(-1, 1, 11)
print H(x)
H = Heaviside(eps=0.8)  # smoothed Heaviside function
print H(x)
```

Hint. Use ideas from Section 5.5.2.

c) Extend class `Heaviside` such that it supports plotting:

```
H = Heaviside()
x, y = H.plot(xmin=-4, xmax=4)  # x in [-4, 4]
from matplotlib.pyplot import plot
plot(x, y)

H = Heaviside(eps=1)
x, y = H.plot(xmin=-4, xmax=4)
plot(x, y)
```

Hint. Techniques from Section 5.4.1 must in the first case be used to return arrays `x` and `y` such that the discontinuity is exactly reproduced. In the continuous (smoothed) case, one needs to compute a sufficiently fine resolution (`x`) based on the `eps` parameter, e.g., $201/\epsilon$ points in the interval $[-\epsilon, \epsilon]$, with a coarser set of coordinates outside this interval where the smoothed Heaviside function is almost constant, 0 or 1.

d) Write a test function `test_Heaviside()` for verifying the result of the various methods in class `Heaviside`.
Filename: `Heaviside_class.py`.

Exercise 7.20: Make a class for the indicator function

Consider the indicator function from Exercise 3.25. Make a class implementation of this function, using the definition (3.27) in terms of Heaviside functions. Allow for an ϵ parameter in the calls to the Heaviside function, as in Exercise 7.19, such that we can easily choose between a discontinuous and a smoothed, continuous version of the indicator function:

```
I = Indicator(a, b)          # indicator function on [a,b]
print I(b+0.1), I((a+b)/2.0)
I = Indicator(0, 2, eps=1)   # smoothed indicator function on [0,2]
print I(0), I(1), I(1.9)
```

Note that if you build on the version of class `Heaviside` in Exercise 7.19b, any `Indicator` instance will accept array arguments too. Filename: `Indicator.py`.

Exercise 7.21: Make a class for piecewise constant functions

The purpose of this exercise is to implement a piecewise constant function, as explained in Exercise 3.26, in a Python class.

a) Implement the minimum functionality such that the following code works:

```
f = PiecewiseConstant([(0.4, 1), (0.2, 1.5), (0.1, 3)], xmax=4)
print f(1.5), f(1.75), f(4)

x = np.linspace(0, 4, 21)
print f(x)
```

b) Add a `plot` method to class `PiecewiseConstant` such that we can easily plot the graph of the function:

```
x, y = f.plot()
from matplotlib.pyplot import plot
plot(x, y)
```

Filename: `PiecewiseConstant.py`.

Exercise 7.22: Speed up repeated integral calculations

The observant reader may have noticed that our `Integral` class from Section 7.3.3 is very inefficient if we want to tabulate or plot a function $F(x) = \int_a^x f(x)$ for several consecutive values of x, say $x_0 < x_1 < \cdots < x_n$. Requesting $F(x_k)$ will recompute the integral computed as part of $F(x_{k-1})$, and this is of course waste of computer work. Use the ideas from Section A.1.7 to modify the `__call__` method such that if `x` is an array, assumed to contain coordinates of increasing value: $x_0 < x_1 < \cdots < x_n$, the method returns an array with $F(x_0), F(x_1), \ldots, F(x_n)$ with the minimum computational work. Filename: `Integral_eff.py`.

Exercise 7.23: Apply a class for polynomials

The Taylor polynomial of degree N for the exponential function e^x is given by

$$p(x) = \sum_{k=0}^{N} \frac{x^k}{k!}\,.$$

Make a program that (i) imports class `Polynomial` from Section 7.3.7, (ii) reads x and a series of N values from the command line, (iii) creates a `Polynomial` instance representing the Taylor polynomial for each N value, and (iv) prints the values of $p(x)$ for all the given N values as well as the exact value e^x. Try the program out with $x = 0.5, 3, 10$ and $N = 2, 5, 10, 15, 25$. Filename: `Polynomial_exp.py`.

Exercise 7.24: Find a bug in a class for polynomials

Go through this alternative implementation of class `Polynomial` from Section 7.3.7 and explain each line in detail:

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        return sum([c*x**i for i, c in enumerate(self.coeff)])

    def __add__(self, other):
        maxlength = max(len(self), len(other))
        # Extend both lists with zeros to this maxlength
        self.coeff += [0]*(maxlength - len(self.coeff))
        other.coeff += [0]*(maxlength - len(other.coeff))
        result_coeff = self.coeff
        for i in range(maxlength):
            result_coeff[i] += other.coeff[i]
        return Polynomial(result_coeff)
```

The `enumerate` function, used in the `__call__` method, enables us to iterate over a list `somelist` with both list indices and list elements: `for index, element in enumerate(somelist)`. Write the code above in a file, and demonstrate that adding two polynomials does not work. Find the bug and correct it. Filename: `Polynomial_error.py`.

Exercise 7.25: Implement subtraction of polynomials

Implement the special method `__sub__` in class `Polynomial` from Section 7.3.7. Add a test for this functionality in function `test_Polynomial`. Filename: `Polynomial_sub.py`.

Exercise 7.26: Test the functionality of pretty print of polynomials

Verify the functionality of the `__str__` method in class `Polynomial` from Section 7.3.7 by writing a new test function `test_Polynomial_str()`. Filename: `Polynomial_test_str.py`.

Exercise 7.27: Vectorize a class for polynomials

Introducing an array instead of a list in class `Polynomial` does not enhance the efficiency of the implementation unless the mathematical computations are also vectorized. That is, all explicit Python loops must be substituted by vectorized expressions.

a) Go through class `Polynomial.py` and make sure the `coeff` attribute is always a `numpy` array with `float` elements.

b) Update the test function `test_Polynomial` to make use of the fact that the `coeff` attribute is always a `numpy` array with `float` elements. Run `test_Polynomial` to check that the new implementation is correct.

c) Vectorize the `__add__` method by adding the common parts of the coefficients arrays and then appending the rest of the longest array to the result.

Hint. Appending an array `a` to an array `b` can be done by `concatenate(a, b)`.

d) Vectorize the `__call__` method by observing that evaluation of a polynomial, $\sum_{i=0}^{n-1} c_i x^i$, can be computed as the inner product of two arrays: $(c_0, \ldots, c_{n-1})$ and $(x^0, x^1, \ldots, x^{n-1})$. The latter array can be computed by `x**p`, where `p` is an array with powers $0, 1, \ldots, n-1$, and `x` is a scalar.

e) The `differentiate` method can be vectorized by the statements

```
n = len(self.coeff)
self.coeff[:-1] = linspace(1, n-1, n-1)*self.coeff[1:]
self.coeff = self.coeff[:-1]
```

Show by hand calculations in a case where `n` is 3 that the vectorized statements produce the same result as the original `differentiate` method.
Filename: `Polynomial_vec.py`.

Remarks. The `__mul__` method is more challenging to vectorize so you may leave this unaltered. Check that the vectorized versions of `__add__`, `__call__`, and `differentiate` work as intended by calling the `test_Polynomial` function.

Exercise 7.28: Use a dict to hold polynomial coefficients

Use a dictionary for the `coeff` attribute in class `Polynomial` from Section 7.3.7 such that `self.coeff[k]` holds the coefficient of the x^k term. The advantage with a dictionary is that only the nonzero coefficients need to be stored.

a) Implement a constructor and the `__add__` method.

b) Implement the `__mul__` method.

c) Write a test function for verifying the implementations in a) and b) when the polynomials $x - 3x^{100}$ and $x^{20} - x + 4x^{100}$ are added and multiplied.
Filename: `Polynomial_dict.py`.

Exercise 7.29: Extend class Vec2D to work with lists/tuples

The `Vec2D` class from Section 7.4 supports addition and subtraction, but only addition and subtraction of two `Vec2D` objects. Sometimes we would like to add or subtract a point that is represented by a list or a tuple:

```
u = Vec2D(-2, 4)
v = u + (1,1.5)
w = [-3, 2] - v
```

That is, a list or a tuple must be allowed in the right or left operand. Implement such an extension of class `Vec2D`.

Hint. Ideas are found in Sections 7.5.3 and 7.5.5.
Filename: `Vec2D_lists.py`.

Exercise 7.30: Extend class Vec2D to 3D vectors

Extend the implementation of class `Vec2D` from Section 7.4 to a class `Vec3D` for vectors in three-dimensional space. Add a method `cross` for computing the cross product of two 3D vectors. Filename: `Vec3D.py`.

Exercise 7.31: Use NumPy arrays in class Vec2D

The internal code in class `Vec2D` from Section 7.4 can be valid for vectors in any space dimension if we represent the vector as a NumPy array in the class instead of separate variables `x` and `y` for the vector components. Make a new class `Vec` where you apply NumPy functionality in the methods. The constructor should be able to treat all the following ways of initializing a vector:

```
a = array([1, -1, 4], float)  # numpy array
v = Vec(a)
v = Vec([1, -1, 4])           # list
v = Vec((1, -1, 4))           # tuple
v = Vec(1, -1)                # coordinates
```

Hint. In the constructor, use variable number of arguments as described in Section H.4. All arguments are then available as a tuple, and if there is only one element in the tuple, it should be an array, list, or tuple you can send through `asarray` to get a NumPy array. If there are many arguments, these are coordinates, and the tuple of arguments can be transformed by `array` to a NumPy array. Assume in all operations that the involved vectors have equal dimension (typically that `other` has the same dimension as `self`). Recall to return `Vec` objects from all arithmetic operations, not NumPy arrays, because the next operation with the vector will then not take place in `Vec` but in NumPy. If `self.v` is the attribute holding the vector as a NumPy array, the addition operator will typically be implemented as

```
class Vec:
    ...
    def __add__(self, other):
        return Vec(selv.v + other.v)
```

Filename: `Vec.py`.

Exercise 7.32: Make classes for students and courses

Redo the summarizing problem in Section 6.6.2 by using classes. More precisely, introduce a class `Student` and a class `Course`. Find appropriate attributes and methods. The classes should have a `__str__` method for pretty-printing of the contents. Filename: `Student_Course.py`.

Exercise 7.33: Find local and global extrema of a function

Extreme points of a function $f(x)$ are normally found by solving $f'(x) = 0$. A much simpler method is to evaluate $f(x)$ for a set of discrete points in the interval $[a, b]$ and look for local minima and maxima among these points. We work with $n + 1$ equally spaced points $a = x_0 < x_1 < \cdots < x_n = b$, $x_i = a + ih$, $h = (b - a)/n$.

First we find all local extreme points in the interior of the domain. Local minima are recognized by

$$f(x_{i-1}) > f(x_i) < f(x_{i+1}), \quad i = 1, \ldots, n-1\,.$$

Similarly, at a local maximum point x_i we have

$$f(x_{i-1}) < f(x_i) > f(x_{i+1}), \quad i = 1, \ldots, n-1\,.$$

Let P_{min} be the set of x values for local minima and F_{min} the set of the corresponding $f(x)$ values at these minima. Two sets P_{max} and F_{max} are defined correspondingly for the maxima.

The boundary points $x = a$ and $x = b$ are for algorithmic simplicity also defined as local extreme points: $x = a$ is a local minimum if $f(a) < f(x_1)$, and a local maximum otherwise. Similarly, $x = b$ is a local minimum if $f(b) < f(x_{n-1})$, and a local maximum otherwise. The end points a and b and the corresponding function values must be added to the sets $P_{\text{min}}, P_{\text{max}}, F_{\text{min}}, F_{\text{max}}$.

The global maximum point is defined as the x value corresponding to the maximum value in F_{max}. The global minimum point is the x value corresponding to the minimum value in F_{min}.

a) Make a class `MinMax` with the following functionality:

- `__init__` takes $f(x)$, a, b, and n as arguments, and calls a method `_find_extrema` to compute the local and global extreme points.
- `_find_extrema` implements the algorithm above for finding local and global extreme points, and stores the sets $P_{\text{min}}, P_{\text{max}}, F_{\text{min}}, F_{\text{max}}$ as list attributes in the (`self`) instance.
- `get_global_minimum` returns the global minimum point as a pair $(x, f(x))$.
- `get_global_maximum` returns the global maximum point as a pair $(x, f(x))$.
- `get_all_minima` returns a list or array of all $(x, f(x))$ minima.
- `get_all_maxima` returns a list or array of all $(x, f(x))$ maxima.
- `__str__` returns a string where a nicely formatted table of all the min/max points are listed, plus the global extreme points.

Here is a sample code using class `MinMax`:

```
def f(x):
    return x**2*exp(-0.2*x)*sin(2*pi*x)

m = MinMax(f, 0, 4, 5001)
print m
```

The output becomes

```
All minima: 0.8056, 1.7736, 2.7632, 3.7584, 0
All maxima: 0.3616, 1.284, 2.2672, 3.2608, 4
Global minimum: 3.7584
Global maximum: 3.2608
```

Make sure that the program also works for functions without local extrema, e.g., linear functions $f(x) = ax + b$.

b) The algorithm sketched above finds local extreme points x_i, but all we know is that the true extreme point is in the interval (x_{i-1}, x_{i+1}). A more accurate algorithm may take this interval as a starting point and run a Bisection method (see Section 4.10.2) to find the extreme point $\bar{x}$ such that $f'(\bar{x}) = 0$. Add a method `_refine_extrema` in class `MinMax`, which goes through all the interior local minima and maxima and solves $f'(\bar{x}) = 0$. Compute $f'(x)$ using the `Derivative` class (Section 7.3.2 with $h \ll x_{i+1} - x_{i-1}$.
Filename: `minmaxf.py`.

Exercise 7.34: Find the optimal production for a company

The company PROD produces two different products, P_1 and P_2, based on three different raw materials, M_1, M_2 and M_3. The following table shows how much of each raw material M_i that is required to produce *a single unit* of each product P_j:

	P_1	P_2
M_1	2	1
M_2	5	3
M_3	0	4

For instance, to produce one unit of P_2 one needs 1 unit of M_1, 3 units of M_2 and 4 units of M_3. Furthermore, PROD has available 100, 80 and 150 units of material M_1, M_2 and M_3 respectively (for the time period considered). The revenue per produced unit of product P_1 is 150 NOK, and for one unit of P_2 it is 175 NOK. On the other hand the raw materials M_1, M_2 and M_3 cost 10, 17 and 25 NOK per unit, respectively. The question is: how much should PROD produce of each product? We here assume that PROD wants to maximize its net revenue (which is revenue minus costs).

a) Let x and y be the number of units produced of product P_1 and P_2, respectively. Explain why the total revenue $f(x, y)$ is given by

$$f(x,y) = 150x - (10 \cdot 2 + 17 \cdot 5)x + 175y - (10 \cdot 1 + 17 \cdot 3 + 25 \cdot 4)y$$

and simplify this expression. The function $f(x, y)$ is *linear* in x and y (make sure you know what linearity means).

b) Explain why PROD's problem may be stated mathematically as follows:

$$\begin{aligned} &\text{maximize} \quad f(x,y) \\ &\text{subject to} \\ &\qquad 2x + y \le 100 \\ &\qquad 5x + 3y \le 80 \\ &\qquad 4y \le 150 \\ &\qquad x \ge 0, y \ge 0. \end{aligned} \tag{7.9}$$

This is an example of a *linear optimization problem.*

c) The production (x, y) may be considered as a point in the plane. Illustrate geometrically the set T of all such points that satisfy the constraints in model (7.9). Every point in this set is called a *feasible point.*

Hint. For every inequality determine first the straight line obtained by replacing the inequality by equality. Then, find the points satisfying the inequality (a half-plane), and finally, intersect these half-planes.

d) Make a program for drawing the straight lines defined by the inequalities. Each line can be written as $ax + by = c$. Let the program read each line from the command line as a list of the a, b, and c values. In the present case the command-line arguments will be

```
'[2,1,100]' '[5,3,80]' '[0,4,150]' '[1,0,0]' '[0,1,0]'
```

Hint. Perform an `eval` on the elements of `sys.argv[1:]` to get a, b, and c for each line as a list in the program.

e) Let α be a positive number and consider the *level set* of the function f, defined as the set

$$L_\alpha = \{(x,y) \in T : f(x,y) = \alpha\}.$$

This set consists of all feasible points having the same net revenue α. Extend the program with two new command-line arguments holding p and q for a function $f(x, y) = px + qy$. Use this information to compute the level set lines $y = \alpha/q - px/q$, and plot the level set lines for some different values of α (use the α value in the legend for each line).

f) Use what you saw in e) to solve the problem (7.9) geometrically. This solution is called an *optimal solution*.

Hint. How large can you choose α such that L_α is nonempty?

g) Assume that we have other values on the revenues and costs than the actual numbers in a). Explain why (7.9), with these new parameter values, still has an optimal solution lying in a corner point of T. Extend the program to calculate all the corner points of a region T in the plane determined by the linear inequalities like those listed above. Moreover, the program shall compute the maximum of a given linear function $f(x, y) = ax + by$ over T by calculating the function values in the corner points and finding the smallest function value.
Filename: `optimization.py`.

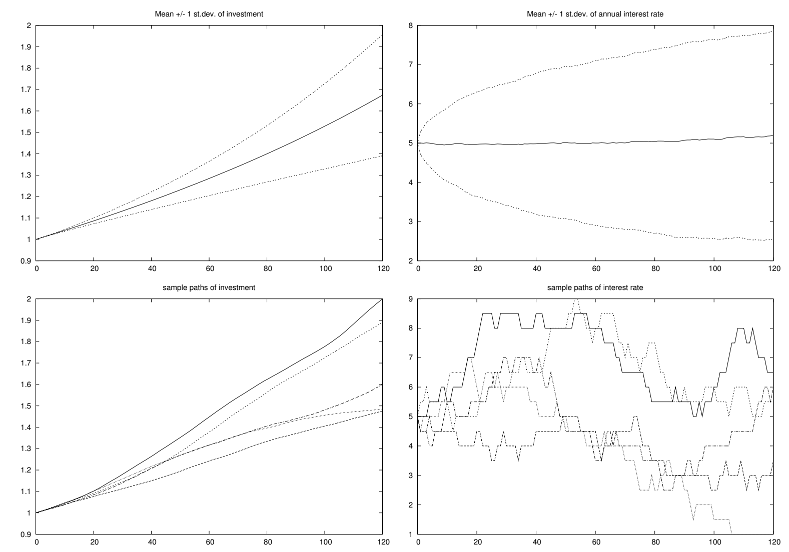


Fig. 8.8 Development of an investment with random jumps of the interest rate at random points of time. Top left: mean value of investment ± one standard deviation. Top right: mean value of the interest rate ± one standard deviation. Bottom left: five paths of the investment development. Bottom right: five paths of the interest rate development.

8.9 Exercises

Exercise 8.1: Flip a coin times

Make a program that simulates flipping a coin N times. Print out `tail` or `head` for each flip and let the program count and print the number of heads.

Hint. Use `r = random.random()` and define head as `r <= 0.5`, or draw an integer among $\{0, 1\}$ with `r = random.randint(0,1)` and define head when `r` is 0.
Filename: `flip_coin.py`.

Exercise 8.2: Compute a probability

What is the probability of getting a number between 0.5 and 0.6 when drawing uniformly distributed random numbers from the interval $[0, 1)$? To answer this question empirically, let a program draw N such random numbers using Python's standard `random` module, count how many of them, M, that fall in the interval $[0.5, 0.6]$, and compute the probability as M/N. Run the program with the four values $N = 10^i$ for $i = 1, 2, 3, 6$.
Filename: `compute_prob.py`.

Exercise 8.3: Choose random colors

Suppose we have eight different colors. Make a program that chooses one of these colors at random and writes out the color.

Hint. Use a list of color names and use the `choice` function in the `random` module to pick a list element.
Filename: `choose_color.py`.

Exercise 8.4: Draw balls from a hat

Suppose there are 40 balls in a hat, of which 10 are red, 10 are blue, 10 are yellow, and 10 are purple. What is the probability of getting two blue and two purple balls when drawing 10 balls at random from the hat? Filename: `4balls_from10.py`.

Exercise 8.5: Computing probabilities of rolling dice

- You throw a die. What is the probability of getting a 6?
  - You throw a die four times in a row. What is the probability of getting 6 all the times?
  - Suppose you have thrown the die three times with 6 coming up all times. What is the probability of getting a 6 in the fourth throw?
  - Suppose you have thrown the die 100 times and experienced a 6 in every throw. What do you think about the probability of getting a 6 in the next throw?

First try to solve the questions from a theoretical or common sense point of view. Thereafter, make functions for simulating cases 1, 2, and 3. Filename: `rolling_dice.py`.

Exercise 8.6: Estimate the probability in a dice game

Make a program for estimating the probability of getting at least one die with six eyes when throwing n dice. Read n and the number of experiments from the command line.

As a partial verification, compare the Monte Carlo simulation results to the exact answer 11/36 for $n = 2$ and observe that the approximate probabilities approach the exact probability as the number of simulations grow. Filename: `one6_ndice.py`.

Exercise 8.7: Compute the probability of hands of cards

Use the `Deck.py` module (see Section 8.2.5) and the `same_rank` and `same_suit` functions from the `cards` module (see Section 8.2.4) to compute the following probabilities by Monte Carlo simulation:

- exactly two pairs among five cards,
- four or five cards of the same suit among five cards,
- four-of-a-kind among five cards.

Filename: `card_hands.py`.

Exercise 8.8: Decide if a dice game is fair

Somebody suggests the following game. You pay 1 euro and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you get paid r euros back, otherwise you lose the 1 euro investment. Assume $r = 10$. Will you then in the long run win or lose money by playing this game? Answer the question by making a program that simulates the game. Read r and the number of experiments N from the command line. Filename: `sum_4dice.py`.

Exercise 8.9: Adjust a game to make it fair

It turns out that the game in Exercise 8.8 is not fair, since you lose money in the long run. The purpose of this exercise is to adjust the winning award so that the game becomes fair, i.e., that you neither lose nor win money in the long run.

Make a program that computes the probability p of getting a sum less than s when rolling n dice. Use the reasoning in Section 8.3.2 to find the award per game, r, that makes the game fair. Run the program from Exercise 8.8 with this r on the command line and verify that the game is now fair. Filename: `sum_s_ndice_fair.py`.

Exercise 8.10: Generalize a game

Consider the game in Section 8.3.2. A generalization is to think as follows: you throw one die until the number of eyes is less than or equal to the previous throw. Let m be the number of throws in a game.

a) Use Monte Carlo simulation to compute the probability of getting $m = 2, 3, 4, \ldots$.

Hint. For $m \geq 6$ the throws must be exactly $1, 2, 3, 4, 5, 6, 6, 6, \ldots$, and the probability of each is $1/6$, giving the total probability 6^{-m}. Use $N = 10^6$ experiments as this should suffice to estimate the probabilities for $m \leq 5$, and beyond that we have the analytical expression.

b) If you pay 1 euro to play this game, what is the fair amount to get paid when win? Answer this question for each of the cases $m = 2, 3, 4, 5$. Filename: `incr_eyes.py`.

Exercise 8.11: Compare two playing strategies

Suggest a player strategy for the game in Section 8.4.2. Remove the question in the `player_guess` function in the file `ndice2.py`, and implement the chosen strategy instead. Let the program play a large number of games, and record the number of times the computer wins. Which strategy is best in the long run: the computer's or yours? Filename: `simulate_strategies1.py`.

Exercise 8.12: Investigate strategies in a game

Extend the program from Exercise 8.11 such that the computer and the player can use a different number of dice. Let the computer choose a random number of dice between 2 and 20. Experiment to find out if there is a favorable number of dice for the player. Filename: `simulate_strategies2.py`.

Exercise 8.13: Investigate the winning chances of some games

An amusement park offers the following game. A hat contains 20 balls: 5 red, 5 yellow, 3 green, and 7 brown. At a cost of $2n$ euros you can draw $4 \leq n \leq 10$ balls at random from the hat (without putting them back). Before you are allowed to look at the drawn balls, you must choose one of the following options:

- win 60 euros if you have drawn exactly three red balls
- win $7 + 5\sqrt{n}$ euros if you have drawn at least three brown balls
- win $n^3 - 26$ euros if you have drawn exactly one yellow ball and one brown ball
- win 23 euros if you have drawn at least one ball of each color

For each of the four different types of games you can play, compute the net income (per play) and the probability of winning for $n = 1, 5, 10, 20$. Is there any of the games (i.e., any combinations of n and the four options above) where you will win money in the long run? Filename: `draw_balls.py`.

Exercise 8.14: Compute probabilities of throwing two dice

Throw two dice a large number of times in a program. Record the sum of the eyes each time and count how many times each of the possibilities for the sum (2, 3, ..., 12) appear. Compute the corresponding probabilities and compare them with the exact values. (To find the exact probabilities, set up all the 6×6 possible outcomes of throwing two dice, and then count how many of them that has a sum s for $s = 2, 3, \ldots, 12$.) Filename: `freq_2dice.py`.

Exercise 8.15: Vectorize flipping a coin

Simulate flipping a coin N times and write out the number of tails. The code should be vectorized, i.e., there must be no loops in Python.

Hint. Constructions like `r[r<=0.5]` or `numpy.where(r<=0.5, 0, 1)`, combined with `numpy.sum`, are useful, where `r` is a vector produced by `numpy.random.random(N)`.
Filename: `flip_coin_vec.py`.

Exercise 8.16: Vectorize a probablility computation

The purpose of this exercise is to speed up the code in Exercise 8.2 by vectorization.

Hint. For an array `r` of uniformly distributed random numbers on $[0, 1)$, make use of `r1 = r[r>0.5]` and `r1[r1<0.6]`. An alternative is `numpy.where` combine with a compound boolean expression with `numpy.logical_and(0.5>=r, r<=0.6)`. See the discussion of this topic in Section 5.5.3.
Filename: `compute_prob_vec.py`.

Exercise 8.17: Throw dice and compute a small probability

Use Monte Carlo simulation to compute the probability of getting 6 eyes on all dice when rolling 7 dice.

Hint. You need a large number of experiments in this case because of the small probability (see the first paragraph of Section 8.3), so a vectorized implementation may be important.
Filename: `roll_7dice.py`.

Exercise 8.18: Difference equation for random numbers

Simple random number generators are based on simulating difference equations. Here is a typical set of two equations:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{8.16}$$

$$y_n = x_n/m, \tag{8.17}$$

for $n = 1, 2, \ldots$. A seed x_0 must be given to start the sequence. The numbers $y_1, y_2, \ldots$ represent the random numbers and $x_0, x_1, \ldots$ are "help" numbers. Although y_n is completely deterministic from (8.16)-(8.17), the sequence y_n *appears* random. The mathematical expression $p \bmod q$ is coded as `p % q` in Python.

Use $a = 8121$, $c = 28411$, and $m = 134456$. Solve the system (8.16)-(8.17) in a function to generate N random numbers. Make a histogram to examine the distribution of the numbers (the y_n numbers are uniformly distributed if the histogram is approximately flat). Filename: `diffeq_random.py`.

Exercise 8.19: Make a class for drawing balls from a hat

Consider the example about drawing colored balls from a hat in Section 8.3.3. It could be handy to have an object that acts as a hat:

```
hat = Hat(red=3, blue=4, green=6)
balls = hat.draw(3)
if balls.count('red') == 1 and balls.count('green') == 2:
    ...
```

a) Write such a class `Hat` with the shown functionality.

Hint 1. The flexible syntax in the constructor, where the colors of the balls and the number of balls of each color are freely specified, requires use of a dictionary (`**kwargs`) for handling a variable number of keyword arguments, see Section H.4.2.

Hint 2. You can borrow useful code from the `balls_in_hat.py` program and ideas from Section 8.2.5.

b) Apply class `Hat` to compute the probability of getting 2 brown and 2 blue galls when drawing 6 balls from a hat with 6 blue, 8 brown, and 3 green balls.
Filename: `Hat.py`.

Exercise 8.20: Independent versus dependent random numbers

a) Generate a sequence of N independent random variables with values 0 or 1 and print out this sequence without space between the numbers (i.e., as `001011010110111010`).

b) The purpose now is to generate random zeros and ones that are dependent. If the last generated number was 0, the probability of generating a new 0 is p and a new 1 is $1 - p$. Conversely, if the last generated was 1, the probability of generating a new 1 is p and a new 0 is $1 - p$. Since the new value depends on the last one, we say the variables are dependent. Implement this algorithm in a function returning an array of N zeros and ones. Print out this array in the condense format as described above.

c) Choose $N = 80$ and try the probabilities $p = 0.5$, $p = 0.8$ and $p = 0.9$. Can you by visual inspection of the output characterize the differences between sequences of independent and dependent random variables?
Filename: `dependent_random_numbers.py`.

Exercise 8.21: Compute the probability of flipping a coin

a) Simulate flipping a coin N times.

Hint. Draw N random integers 0 and 1 using `numpy.random.randint`.

b) Look at a subset $N_1 \leq N$ of the experiments in a) and compute the probability of getting a head (M_1/N_1, where M_1 is the number of heads in N_1 experiments). Choose $N = 1000$ and print out the probability for $N_1 = 10, 100, 500, 1000$. Generate just N numbers once in the program. How do you think the accuracy of the computed probability vary with N_1? Is the output compatible with this expectation?

c) Now we want to study the probability of getting a head, p, as a function of N_1, i.e., for $N_1 = 1, \ldots, N$. A first try to compute the probability array for p is

```
import numpy as np
h = np.where(r <= 0.5, 1, 0)
p = np.zeros(N)
for i in range(N):
    p[i] = np.sum(h[:i+1])/float(i+1)
```

Implement these computations in a function.

d) An array `q[i] = np.sum(h([:i]))` reflects a *cumulative sum* and can be efficiently generated by `np.cumsum`: `q = np.cumsum(h)`. Thereafter we can compute `p` by `q/I`, where `I[i]=i+1` and `I` can be computed by `np.arange(1,N+1)` or `r_[1:N+1]` (integers 1, 2, ..., up to but not including `N+1`). Use `cumsum` to make an alternative vectorized version of the function in c).

e) Write a test function that verifies that the implementations in c) and d) give the same results.

Hint. Use `numpy.allclose` to compare two arrays.

f) Make a function that applies the `time` module to measure the relative efficiency of the implementations in c) and d).

g) Plot `p` against `I` for the case where $N = 10000$. Annotate the axis and the plot with relevant text.
Filename: `flip_coin_prob.py`.

Exercise 8.22: Simulate binomial experiments

Exercise 4.23 describes some problems that can be solved exactly using the formula (4.8), but we can also simulate these problems and find approximate numbers for the probabilities. That is the task of this exercise.

Make a general function `simulate_binomial(p, n, x)` for running n experiments, where each experiment have two outcomes, with probabilities p and $1 - p$. The n experiments constitute a *success* if the outcome with probability p occurs exactly x times. The `simulate_binomial` function must repeat the n experiments N times. If M is the number of successes in the N experiments, the probability estimate is M/N. Let the function return this probability estimate together with the error (the exact result is (4.8)). Simulate the three cases in Exercise 4.23 using this function. Filename: `simulate_binomial.py`.

Exercise 8.23: Simulate a poker game

Make a program for simulating the development of a poker (or simplified poker) game among n players. Use ideas from Section 8.2.4. Filename: `poker.py`.

Exercise 8.24: Estimate growth in a simulation model

The simulation model in Section 8.3.5 predicts the number of individuals from generation to generation. Make a simulation of the one son policy with 10 generations, a male portion of 0.51 among newborn babies, set the fertility to 0.92, and assume that a fraction 0.06 of the population will break the law and want 6 children in a family. These parameters implies a significant growth of the population. See if you can find a factor r such that the number of individuals in generation n fulfills the difference equation

$$x_n = (1 + r)x_{n-1}\,.$$

Hint. Compute r for two consecutive generations x_{n-1} and x_n ($r = x_n/x_{n-1} - 1$) and see if r is approximately constant as n increases.
Filename: `estimate_growth.py`.

Exercise 8.25: Investigate guessing strategies

In the game from Section 8.4.1 it is smart to use the feedback from the program to track an interval $[p, q]$ that must contain the secret number. Start with $p = 1$ and $q = 100$. If the user guesses at some number n, update p to $n + 1$ if n is less than the secret number (no need to care about numbers smaller than $n + 1$), or update q to $n - 1$ if n is larger than the secret number (no need to care about numbers larger than $n - 1$).

Are there any smart strategies to pick a new guess $s \in [p, q]$? To answer this question, investigate two possible strategies: s as the midpoint in the interval $[p, q]$, or s as a uniformly distributed random integer in $[p, q]$. Make a program that implements both strategies, i.e., the player is not prompted for a guess but the computer computes the guess based on the chosen strategy. Let the program run a large number of games and see if one of the strategies can be considered as superior in the long run. Filename: `strategies4guess.py`.

Exercise 8.26: Vectorize a dice game

Vectorize the simulation program from Exercise 8.8 with the aid of the module `numpy.random` and the `numpy.sum` function. Filename: `sum9_4dice_vec.py`.

Exercise 8.27: Compute π by a Monte Carlo method

Use the method in Section 8.5.3 to compute π by computing the area of a circle. Choose G as the circle with its center at the origin and with unit radius, and choose B as the rectangle $[-1, 1] \times [-1, 1]$. A point (x, y) lies within G if $x^2 + y^2 < 1$. Compare the approximate π with `math.pi`. Filename: `MC_pi.py`.

Exercise 8.28: Compute π by a Monte Carlo method

This exercise has the same purpose of computing π as in Exercise 8.27, but this time you should choose G as a circle with center at $(2, 1)$ and radius 4. Select an appropriate rectangle B. A point (x, y) lies within a circle with center at (x_c, y_c) and with radius R if $(x-x_c)^2+(y-y_c)^2 < R^2$. Filename: `MC_pi2.py`.

Exercise 8.29: Compute π by a random sum

a) Let $x_0, \ldots, x_N$ be $N+1$ uniformly distributed random numbers between 0 and 1. Explain why the random sum $S_N = (N+1)^{-1} \sum_{i=0}^{N} 2(1-x_i^2)^{-1/2}$ is an approximation to π.

Hint. Interpret the sum as Monte Carlo integration and compute the corresponding integral by hand or `sympy`.

b) Compute $S_0, S_1, \ldots, S_N$ (using just one set of $N+1$ random numbers). Plot this sequence versus N. Also plot the horizontal line corresponding to the value of π. Choose N large, e.g., $N = 10^6$.
Filename: `MC_pi_plot.py`.

Exercise 8.30: 1D random walk with drift

Modify the `walk1D.py` program such that the probability of going to the right is r and the probability of going to the left is $1-r$ (draw numbers in $[0, 1)$ rather than integers in $\{1, 2\}$). Compute the average position of n_p particles after 100 steps, where n_p is read from the command line. Mathematically one can show that the average position approaches $rn_s - (1-r)n_s$ as $n_p \to \infty$ (n_s is the number of walks). Write out this exact result together with the computed mean position with a finite number of particles. Filename: `walk1D_drift.py`.

Exercise 8.31: 1D random walk until a point is hit

Set `np=1` in the `walk1Dv.py` program and modify the program to measure how many steps it takes for one particle to reach a given point $x = x_p$. Give x_p on the command line. Report results for $x_p = 5, 50, 5000, 50000$.
Filename: `walk1Dv_hit_point.py`.

Exercise 8.32: Simulate making a fortune from gaming

A man plays a game where the probability of winning is p and that of losing is consequently $1-p$. When winning he earns 1 euro and when losing he loses 1 euro. Let x_i be the man's fortune from playing this game i number of times. The starting fortune is x_0. We assume that the man gets a necessary loan if $x_i < 0$ such that the gaming can continue. The target is a fortune F, meaning that the playing stops when $x = F$ is reached.

a) Explain why x_i is a 1D random walk.

b) Modify one of the 1D random walk programs to simulate the average number of games it takes to reach the target fortune $x = F$. This average must be computed by running a large number of random walks that start at x_0 and reach F. Use $x_0 = 10$, $F = 100$, and $p = 0.49$ as example.

c) Suppose the average number of games to reach $x = F$ is proportional to $(F - x_0)^r$, where r is some exponent. Try to find r by experimenting with the program. The r value indicates how difficult it is to make a substantial fortune by playing this game. Note that the *expected* earning is negative when $p < 0.5$, but there is still a small probability for hitting $x = F$.
Filename: `game_as_walk1D.py`.

Exercise 8.33: Make classes for 2D random walk

The purpose of this exercise is to reimplement the `walk2D.py` program from Section 8.7.1 with the aid of classes.

a) Make a class `Particle` with the coordinates (x, y) and the time step number of a particle as attributes. A method `move` moves the particle in one of the four directions and updates the (x, y) coordinates. Another class, `Particles`, holds a list of `Particle` objects and a `plotstep` parameter (as in `walk2D.py`). A method `move` moves all the particles one step, a method `plot` can make a plot of all particles, while a method `moves` performs a loop over time steps and calls `move` and `plot` in each step.

b) Equip the `Particle` and `Particles` classes with print functionality such that one can print out all particles in a nice way by saying `print p` (for a `Particles` instance `p`) or `print self` (inside a method).

Hint. In `__str__`, apply the `pformat` function from the `pprint` module to the list of particles, and make sure that `__repr__` just reuse `__str__` in both classes so the output looks nice.

c) Make a test function that compares the first three positions of four particles with the corresponding results computed by the `walk2D.py` program. The seed of the random number generator must of course be fixed identically in the two programs.

d) Organize the complete code as a module such that the classes `Particle` and `Particles` can be reused in other programs. The test block should read the number of particles from the command line and perform a simulation.

e) Compare the efficiency of the class version against the vectorized version in `walk2Dv.py`, using the techniques in Section H.5.1.

f) The program developed above cannot be vectorized as long as we base the implementation on class `Particle`. However, if we remove that class and focus on class `Particles`, the latter can employ arrays for holding the positions of all particles and vectorized updates of these positions in the `moves` method. Use ideas from the `walk2Dv.py` program to make a new class `Particles_vec` which vectorizes `Particles`.

g) Verify the code against the `walk2Dv.py` program as explained in c). Automate the verification in a test function.

h) Write a Python function that measures the computational efficiency the vectorized class `Particles_vec` and the scalar class `Particles`.
Filename: `walk2D_class.py`.

Exercise 8.34: 2D random walk with walls; scalar version

Modify the `walk2D.py` or `walk2Dc.py` programs from Exercise 8.33 so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$. Do not move the particle if its new position is outside A. Filename: `walk2D_barrier.py`.

Exercise 8.35: 2D random walk with walls; vectorized version

Modify the `walk2Dv.py` program so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$.

Hint. First perform the moves of one direction. Then test if new positions are outside A. Such a test returns a boolean array that can be used as index in the position arrays to pick out the indices of the particles that have moved outside A and move them back to the relevant boundary of A.
Filename: `walk2Dv_barrier.py`.

Exercise 8.36: Simulate mixing of gas molecules

Suppose we have a box with a wall dividing the box into two equally sized parts. In one part we have a gas where the molecules are uniformly distributed in a random fashion. At $t = 0$ we remove the wall. The gas molecules will now move around and eventually fill the whole box.

This physical process can be simulated by a 2D random walk inside a fixed area A as introduced in Exercises 8.34 and 8.35 (in reality the motion is three-dimensional, but we only simulate the two-dimensional part of it since we already have programs for doing this). Use the program from either Exercises 8.34 or 8.35 to simulate the process for $A = [0, 1] \times [0, 1]$. Initially, place 10000 particles at uniformly distributed random positions in $[0, 1/2] \times [0, 1]$. Then start the random walk and visualize what happens. Simulate for a long time and make a hardcopy of the animation (an animated GIF file, for instance). Is the end result what you would expect? Filename: `disorder1.py`.

Remarks. Molecules tend to move randomly because of collisions and forces between molecules. We do not model collisions between particles in the random walk, but the nature of this walk, with random movements, simulates the effect of collisions. Therefore, the random walk can be used to model molecular motion in many simple cases. In particular, the random walk can be used to investigate how a quite ordered system, where one gas fills one half of a box, evolves through time to a more disordered system.

Exercise 8.37: Simulate slow mixing of gas molecules

Solve Exercise 8.36 when the wall dividing the box is not completely removed, but instead has a small hole. Filename: `disorder2.py`.

Exercise 8.38: Guess beer brands

You are presented n glasses of beer, each containing a different brand. You are informed that there are $m \geq n$ possible brands in total, and the names of all brands are given. For each glass, you can pay p euros to taste the beer, and if you guess the right brand, you get $q \geq p$ euros back. Suppose you have done this before and experienced that you typically manage to guess the right brand T times out of 100, so that your probability of guessing the right brand is $b = T/100$.

Make a function `simulate(m, n, p, q, b)` for simulating the beer tasting process. Let the function return the amount of money earned and how many correct guesses ($\leq n$) you made. Call `simulate` a large number of times and compute the average earnings and the probability of getting full score in the case $m = n = 4$, $p = 3$, $q = 6$, and $b = 1/m$ (i.e., four glasses with four brands, completely random guessing, and a payback of twice as much as the cost). How much more can you earn from this game if your ability to guess the right brand is better, say $b = 1/2$? Filename: `simulate_beer_tasting.py`.

Exercise 8.39: Simulate stock prices

A common mathematical model for the evolution of stock prices can be formulated as a difference equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1}\sqrt{\Delta t} r_{n-1}, \tag{8.18}$$

where x_n is the stock price at time t_n, Δt is the time interval between two time levels ($\Delta t = t_n - t_{n-1}$), μ is the growth rate of the stock price, σ is the volatility of the stock price, and $r_0, \ldots, r_{n-1}$ are normally distributed random numbers with mean zero and unit standard deviation. An initial stock price x_0 must be prescribed together with the input data μ, σ, and Δt.

We can make a remark that (8.18) is a Forward Euler discretization of a stochastic differential equation for a continuous price function $x(t)$:

$$\frac{dx}{dt} = \mu x + \sigma N(t),$$

where $N(t)$ is a so-called white noise random time series signal. Such equations play a central role in modeling of stock prices.

Make R realizations of (8.18) for $n = 0, \ldots, N$ for $N = 5000$ steps over a time period of $T = 180$ days with a step size $\Delta t = T/N$. Filename: `stock_prices.py`.

Exercise 8.40: Compute with option prices in finance

In this exercise we are going to consider the pricing of so-called Asian options. An Asian option is a financial contract where the owner earns money when certain market conditions are satisfied.

The contract is specified by a *strike price* K and a maturity time T. It is written on the average price of the underlying stock, and if this average is bigger than the strike K, the owner of the option will earn the difference. If, on the other hand, the average becomes less, the owner receives nothing, and the option matures in the value zero. The average is calculated from the last trading price of the stock for each day.

From the theory of options in finance, the price of the Asian option will be the expected present value of the payoff. We assume the stock price dynamics given as,

$$S(t+1) = (1+r)S(t) + \sigma S(t)\epsilon(t), \tag{8.19}$$

where r is the interest-rate, and σ is the volatility of the stock price. The time t is supposed to be measured in days, $t = 0, 1, 2, \ldots$, while $\epsilon(t)$ are independent identically distributed normal random variables with mean zero and unit standard deviation. To find the option price, we must calculate the expectation

$$p = (1+r)^{-T}\mathrm{E}\left[\max\left(\frac{1}{T}\sum_{t=1}^{T} S(t) - K, 0\right)\right]. \tag{8.20}$$

The price is thus given as the expected discounted payoff. We will use Monte Carlo simulations to estimate the expectation. Typically, r and σ can be set to $r = 0.0002$ and $\sigma = 0.015$. Assume further $S(0) = 100$.

a) Make a function that simulates a path of $S(t)$, that is, the function computes $S(t)$ for $t = 1, \ldots, T$ for a given T based on the recursive definition in (8.19). The function should return the path as an array.

b) Create a function that finds the average of $S(t)$ from $t = 1$ to $t = T$. Make another function that calculates the price of the Asian option based on N simulated averages. You may choose $T = 100$ days and $K = 102$.

c) Plot the price p as a function of N. You may start with $N = 1000$.

d) Plot the error in the price estimation as a function N (assume that the p value corresponding to the largest N value is the "right" price). Try to fit a curve of the form $c/\sqrt{N}$ for some c to this error plot. The purpose is to show that the error is reduced as $1/\sqrt{N}$.
Filename: `option_price.py`.

Remarks. If you wonder where the values for r and σ come from, you will find the explanation in the following. A reasonable level for the yearly interest-rate is around 5 percent, which corresponds to a daily rate $0.05/250 = 0.0002$. The number 250 is chosen because a stock exchange is on average open this amount of days for trading. The value for σ is calculated as the volatility of the stock price, corresponding to the standard deviation of the daily returns of the stock defined as $(S(t + 1) - S(t))/S(t)$. "Normally", the volatility is around 1.5 percent a day. Finally, there are theoretical reasons why we assume that the stock price dynamics is driven by r, meaning that we consider the *risk-neutral* dynamics of the stock price when pricing options. There is an exciting theory explaining the appearance of r in the dynamics of the stock price. If we want to simulate a stock price dynamics mimicing what we see in the market, r in (8.19) must be substituted with μ, the expected return of the stock. Usually, μ is higher than r.

Exercise 8.41: Differentiate noise measurements

In a laboratory experiment waves are generated through the impact of a model slide into a wave tank. (The intention of the experiment is to model a future tsunami event in a fjord, generated by loose rocks that fall into the fjord.) At a certain location, the elevation of the surface, denoted by η, is measured at discrete points in time using an ultra-sound wave gauge. The result is a time series of vertical positions of the water surface elevations in meter: $\eta(t_0), \eta(t_1), \eta(t_2), \ldots, \eta(t_n)$. There are 300 observations per second, meaning that the time difference between two neighboring measurement values $\eta(t_i)$ and $\eta(t_{i+1})$ is $h = 1/300$ second.

a) Read the η values in the file `gauge.dat`[3] into an array `eta`. Read h from the command line.

b) Plot `eta` versus the time values.

c) Compute the velocity v of the surface by the formula

$$v_i \approx (\eta_{i+1} - \eta_{i-1})/(2h), \quad i = 1, \ldots, n-1 \,.$$

Plot v versus time values in a separate plot.

d) Compute the acceleration a of the surface by the formula

$$a_i \approx (\eta_{i+1} - 2\eta_i + \eta_{i-1})/h^2, \quad i = 1, \ldots, n-1.$$

Plot a versus the time values in a separate plot.

e) If we have a noisy signal η_i, where $i = 0, \ldots, n$ counts time levels, the noise can be reduced by computing a new signal where the value at a point is a weighted average of the values at that point and the neighboring points at each side. More precisely, given the signal η_i, $i = 0, \ldots, n$, we compute a filtered (averaged) signal with values $\eta_i^{(1)}$ by the formula

$$\eta_i^{(1)} = \frac{1}{4}(\eta_{i+1} + 2\eta_i + \eta_{i-1}), \quad i = 1, \ldots, n-1, \ \eta_0^{(1)} = \eta_0, \ \eta_n^{(1)} = \eta_n \,. \tag{8.21}$$

Make a function `filter` that takes the η_i values in an array `eta` as input and returns the filtered $\eta_i^{(1)}$ values in an array.

f) Let $\eta_i^{(k)}$ be the signal arising by applying the `filtered` function k times to the same signal. Make a plot with curves η_i and the filtered $\eta_i^{(k)}$ values for $k = 1, 10, 100$. Make similar plots for the velocity and acceleration where these are made from both the original, measured η data and the filtered data. Discuss the results.
Filename: `labstunami.py`.

Exercise 8.42: Differentiate noisy signals

The purpose of this exercise is to look into numerical differentiation of time series signals that contain measurement errors. This insight might be helpful when analyzing the noise in real data from a laboratory experiment in Exercise 8.41.

a) Compute a signal

$$\bar{\eta}_i = A\sin(\frac{2\pi}{T}t_i), \quad t_i = i\frac{T}{40}, \ i = 0, \ldots, 200 \,.$$

Display $\bar{\eta}_i$ versus time t_i in a plot. Choose $A = 1$ and $T = 2\pi$. Store the $\bar{\eta}$ values in an array `etabar`.

[3] `http://tinyurl.com/pwyasaa/random/gauge.dat`

b) Compute a signal with random noise E_i,

$$\eta_i = \bar{\eta}_i + E_i,$$

E_i is drawn from the normal distribution with mean zero and standard deviation $\sigma = 0.04A$. Plot this η_i signal as circles in the same plot as $\bar{\eta}_i$. Store the E_i in an array `E` for later use.

c) Compute the first derivative of $\bar{\eta}_i$ by the formula

$$\frac{\bar{\eta}_{i+1} - \bar{\eta}_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$

and store the values in an array `detabar`. Display the graph.

d) Compute the first derivative of the error term by the formula

$$\frac{E_{i+1} - E_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$

and store the values in an array `dE`. Calculate the mean and the standard deviation of `dE`.

e) Plot `detabar` and `detabar + dE`. Use the result of the standard deviation calculations to explain the qualitative features of the graphs.

f) The second derivative of a time signal η_i can be computed by

$$\frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \ldots, n-1\,.$$

Use this formula on the `etabar` data and save the result in `d2etabar`. Also apply the formula to the `E` data and save the result in `d2E`. Plot `d2etabar` and `d2etabar + d2E`. Compute the standard deviation of `d2E` and compare with the standard deviation of `dE` and `E`. Discuss the plot in light of these standard deviations.
Filename: `sine_noise.py`.

Exercise 8.43: Model noise in a time signal

We assume that the measured data can be modeled as a smooth time signal $\bar{\eta}(t)$ plus a random variation $E(t)$. Computing the velocity of $\eta = \bar{\eta} + E$ results in a smooth velocity from the $\bar{\eta}$ term and a noisy signal from the E term.

a) We can estimate the level of noise in the first derivative of E as follows. The random numbers $E(t_i)$ are assumed to be independent and normally distributed with mean zero and standard deviation σ. It can then be shown that

$$\frac{E_{i+1} - E_{i-1}}{2h}$$

produces numbers that come from a normal distribution with mean zero and standard deviation $2^{-1/2}h^{-1}\sigma$. How much is the original noise, reflected by σ, magnified when we use this numerical approximation of the velocity?

b) The fraction

$$\frac{E_{i+1} - 2E_i + E_{i-1}}{h^2}$$

will also generate numbers from a normal distribution with mean zero, but this time with standard deviation $2h^{-2}\sigma$. Find out how much the noise is magnified in the computed acceleration signal.

c) The numbers in the `gauge.dat` file in Exercise 8.41 are given with 5 digits. This is no certain indication of the accuracy of the measurements, but as a test we may assume σ is of the order 10^{-4}. Check if the visual results for the velocity and acceleration are consistent with the standard deviation of the noise in these signals as modeled above.

Exercise 8.44: Speed up Markov chain mutation

The functions `transition` and `mutate_via_markov_chain` from Section 8.3.4 were made for being easy to read and understand. Upon closer inspection, we realize that the `transition` function constructs the `interval_limits` every time a random transition is to be computed, and we want to run a large number of transitions. By merging the two functions, pre-computing interval limits for each `from_base`, and adding a loop over `N` mutations, one can reduce the computation of interval limits to a minimum. Perform such an efficiency enhancement. Measure the CPU time of this new function versus the `mutate_via_markov_chain` function for 1 million mutations. Filename: `markov_chain_mutation2.py`.

We can also drop the redirection of standard input to a file, and instead run an interactive session in IPython or the terminal window:

Terminal

```
demo_ReadInput.py ReadFileInput
n = 101
filename = myfunction_data_file.dat
^D
{'a': 0,
 'b': 1,
 'filename': 'myfunction_data_file.dat',
 'formula': 'x+1',
 'n': 101}
```

Note that `Ctrl+d` is needed to end the interactive session with the user and continue program execution.

Command-line arguments can also be specified:

Terminal

```
demo_ReadInput.py ReadCommandLine \
          --a -1 --b 1 --formula "sin(x) + cos(x)"
{'a': -1, 'b': 1, 'filename': 'tmp.dat',
 'formula': 'sin(x) + cos(x)', 'n': 2}
```

Finally, we can run the program with a GUI,

Terminal

```
demo_ReadInput.py GUI
{'a': -1, 'b': 10, 'filename': 'tmp.dat',
 'formula': 'x+1', 'n': 2}
```

The GUI is shown in Figure 9.13.

Fortunately, it is now quite obvious how to apply the `ReadInput` hierarchy of classes in your own programs to simplify input. Especially in applications with a large number of parameters one can initially define these in a dictionary and then automatically create quite comprehensive user interfaces where the user can specify only some subset of the parameters (if the default values for the rest of the parameters are suitable).

9.7 Exercises

Exercise 9.1: Demonstrate the magic of inheritance

Consider class `Line` from Section 9.1.1 and a subclass `Parabola0` defined as

```python
class Parabola0(Line):
    pass
```

That is, class `Parabola0` does not have any own code, but it inherits from class `Line`. Demonstrate in a program or interactive session, using `dir` and looking at the `__dict__` object, (see Section 7.5.6) that an instance of class `Parabola0` contains everything (i.e., all attributes and methods) that an instance of class `Line` contains. Filename: `dir_subclass.py`.

Exercise 9.2: Make polynomial subclasses of parabolas

The task in this exercise is to make a class `Cubic` for cubic functions

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

with a call operator and a `table` method as in classes `Line` and `Parabola` from Section 9.1. Implement class `Cubic` by inheriting from class `Parabola`, and call up functionality in class `Parabola` in the same way as class `Parabola` calls up functionality in class `Line`.

Make a similar class `Poly4` for 4-th degree polynomials

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

by inheriting from class `Cubic`. Insert `print` statements in all the `__call__` to help following the program flow. Evaluate cubic and a 4-th degree polynomial at a point, and observe the printouts from all the superclasses. Filename: `Cubic_Poly4.py`.

Remarks. This exercise follows the idea from Section 9.1 where more complex polynomials are subclasses of simpler ones. Conceptually, a cubic polynomial *is not* a parabola, so many programmers will not accept class `Cubic` as a subclass of `Parabola`; it should be the other way around, and Exercise 9.2 follows that approach. Nevertheless, one can use inheritance solely for sharing code and not for expressing that a subclass `is a` kind of the superclass. For code sharing it is natural to start with the simplest polynomial as superclass and add terms to the inherited data structure as we make subclasses for higher degree polynomials.

Exercise 9.3: Implement a class for a function as a subclass

Implement a class for the function $f(x) = A\sin(wx) + ax^2 + bx + c$. The class should have a call operator for evaluating the function for some argument x, and a constructor that takes the function parameters `A`, `w`, `a`, `b`, and `c` as arguments. Also a `table` method as in classes `Line` and `Parabola` should be present. Implement the class by deriving it from class `Parabola` and call up functionality already implemented in class `Parabola` whenever possible. Filename: `sin_plus_quadratic.py`.

Exercise 9.4: Create an alternative class hierarchy for polynomials

Let class `Polynomial` from Section 7.3.7 be a superclass and implement class `Parabola` as a subclass. The constructor in class `Parabola` should take the three coefficients in the parabola as separate arguments. Try to reuse as much code as possible from the superclass in the subclass. Implement class `Line` as a subclass specialization of class `Parabola`.

Which class design do you prefer, class `Line` as a subclass of `Parabola` and `Polynomial`, or `Line` as a superclass with extensions in subclasses? (See also remark in Exercise 9.2.) Filename: `Polynomial_hier.py`.

Exercise 9.5: Make circle a subclass of an ellipse

Section 7.2.3 presents class `Circle`. Make a similar class `Ellipse` for representing an ellipse. Then create a new class `Circle` that is a subclass of `Ellipse`. Filename: `Ellipse_Circle.py`.

Exercise 9.6: Make super- and subclass for a point

A point (x, y) in the plane can be represented by a class:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

We can extend the `Point` class to also contain the representation of the point in polar coordinates. To this end, create a subclass `PolarPoint` whose constructor takes the polar representation of a point, (r, θ), as arguments. Store r and θ as attributes and call the superclass constructor with the corresponding x and y values (recall the relations $x = r\cos\theta$ and $y = r\sin\theta$ between Cartesian and polar coordinates). Add a `__str__` method in class `PolarPoint` which prints out r, θ, x, and y. Verify the implementation by initializing three points and printing these points. Filename: `PolarPoint.py`.

Exercise 9.7: Modify a function class by subclassing

Consider a class `F` implementing the function $f(t; a, b) = e^{-at}\sin(bt)$:

```
class F:
    def __init__(self, a, b):
        self.a, self.b = a, b
    def __call__(self, t):
        return exp(-self.a*t)*sin(self.b*t)
```

We now want to study how the function $f(t; a, b)$ varies with the parameter b, given t and a. Mathematically, this means that we want to compute $g(b; t, a) = f(t; a, b)$. Write a subclass `Fb` of `F` with a new `__call__` method for evaluating $g(b; t, a)$. Do not reimplement the formula, but call the `__call__` method in the superclass to evaluate $f(t; a, b)$. The `Fs` should work as follows:

```
f = Fs(t=2, a=4.5)
print f(3)  # b=3
```

Hint. Before calling `__call__` in the superclass, the attribute `b` in the superclass must be set to the right value.
Filename: `Fb.py`.

Exercise 9.8: Explore the accuracy of difference formulas

The purpose of this exercise is to investigate the accuracy of the `Backward1`, `Forward1`, `Forward3`, `Central2`, `Central4`, `Central6` methods for the function

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}} .$$

Compute the errors in the approximations for $x = 0, 0.9$ and $\mu = 1, 0.01$. Illustrate in a plot how the $v(x)$ function looks like for these two μ values.

Hint. Modify the `src/oo/Diff2_examples.py` program which produces tables of errors of difference approximations as discussed at the end of Section 9.2.4.
Filename: `boundary_layer_derivative.py`.

Exercise 9.9: Implement a subclass

Make a subclass `Sine1` of class `FuncWithDerivatives` from Section 9.1.6 for the $\sin x$ function. Implement the function only, and rely on the inherited `df` and `ddf` methods for computing the derivatives. Make another subclass `Sine2` for $\sin x$ where you also implement the `df` and `ddf` methods using analytical expressions for the derivatives. Compare `Sine1` and `Sine2` for computing the first- and second-order derivatives of $\sin x$ at two x points. Filename: `Sine12.py`.

Exercise 9.10: Make classes for numerical differentiation

Carry out Exercise 7.16. Find the common code in the classes `Derivative`, `Backward`, and `Central`. Move this code to a superclass, and let the three mentioned classes be subclasses of this superclass. Compare the resulting code with the hierarchy shown in Section 9.2.1. Filename: `numdiff_classes.py`.

Exercise 9.11: Implement a new subclass for differentiation

A one-sided, three-point, second-order accurate formula for differentiating a function $f(x)$ has the form

$$f'(x) \approx \frac{f(x-2h) - 4f(x-h) + 3f(x)}{2h}\,. \tag{9.17}$$

Implement this formula in a subclass `Backward2` of class `Diff` from Section 9.2. Compare `Backward2` with `Backward1` for $g(t) = e^{-t}$ for $t = 0$ and $h = 2^{-k}$ for $k = 0, 1, \ldots, 14$ (write out the errors in $g'(t)$). Filename: `Backward2.py`.

Exercise 9.12: Understand if a class can be used recursively

Suppose you want to compute $f''(x)$ of some mathematical function $f(x)$, and that you apply some class from Section 9.2 twice, e.g.,

```
ddf = Central2(Central2(f))
```

Will this work?

Hint. Follow the program flow, and find out what the resulting formula will be. Then see if this formula coincides with a formula you know for approximating $f''(x)$ (actually, to recover the well-known formula with an h parameter, you would use $h/2$ in the nested calls to `Central2`).

Exercise 9.13: Represent people by a class hierarchy

Classes are often used to model objects in the real world. We may represent the data about a person in a program by a class `Person`, containing the person's name, address, phone number, date of birth, and nationality. A method `__str__` may print the person's data. Implement such a class `Person`.

A worker is a person with a job. In a program, a worker is naturally represented as class `Worker` derived from class `Person`, because a worker *is* a person, i.e., we have an is-a relationship. Class `Worker` extends class `Person` with additional data, say name of company, company address, and job phone number. The print functionality must be modified accordingly. Implement this `Worker` class.

A scientist is a special kind of a worker. Class `Scientist` may therefore be derived from class `Worker`. Add data about the scientific discipline (physics, chemistry, mathematics, computer science, ...). One may also add the type of scientist: theoretical, experimental, or computational. The value of such a type attribute should not be restricted to just one category, since a scientist may be classified as, e.g., both experimental and computational (i.e., you can represent the value as a list or tuple). Implement class `Scientist`.

Researcher, postdoc, and professor are special cases of a scientist. One can either create classes for these job positions, or one may add an attribute (`position`) for this information in class `Scientist`. We adopt the former strategy. When, e.g., a researcher is represented by a class `Researcher`, no extra data or methods are needed. In Python we can create such an empty class by writing `pass` (the empty statement) as the class body:

```
class Researcher(Scientist):
    pass
```

Finally, make a demo program where you create and print instances of classes `Person`, `Worker`, `Scientist`, `Researcher`, `Postdoc`, and `Professor`. Print out the attribute contents of each instance (use the `dir` function).

Remark. An alternative design is to introduce a class `Teacher` as a special case of `Worker` and let `Professor` be both a `Teacher` and `Scientist`, which is natural. This implies that class `Professor` has two superclasses, `Teacher` and `Scientist`, or equivalently, class `Professor` inherits from two superclasses. This is known as *multiple inheritance* and technically achieved as follows in Python:

```
class Professor(Teacher, Scientist):
    pass
```

It is a continuous debate in computer science whether multiple inheritance is a good idea or not. One obvious problem in the present example is that class `Professor` inherits two names, one via `Teacher` and one via `Scientist` (both these classes inherit from `Person`). Filename: `Person.py`.

Exercise 9.14: Add a new class in a class hierarchy

Add the Monte Carlo integration method from Section 8.5.2 as a subclass in the `Integrator` hierarchy explained in Section 9.3. Import the superclass `Integrator` from the `integrate` module in the file with the new integration class. Test the Monte Carlo integration class in a case with known analytical solution. Filename: `MCint_class.py`.

Exercise 9.15: Compute convergence rates of numerical integration methods

Numerical integration methods can compute “any“ integral $\int_a^b f(x)dx$, but the result is not exact. The methods have a parameter n, closely related to the number of evaluations of the function f, that can be increased to achieve more accurate results. In this exercise we want to explore the relation between the error E in the numerical approximation to the integral and n. Different numerical methods have different relations.

The relations are of the form

$$E = Cn^r,$$

where and C and $r < 0$ are constants to be determined. That is, r is the most important of these parameters, because if Simpson's method has a more negative r than the Trapezoidal method, it means that increasing n in Simpson's method reduces the error more effectively than increasing n in the Trapezoidal method.

One can estimate r from numerical experiments. For a chosen $f(x)$, where the exact value of $\int_a^b f(x)dx$ is available, one computes the numerical approximation for $N + 1$ values of n: $n_0 < n_1 < \cdots < n_N$ and finds the corresponding errors $E_0, E_1, \ldots, E_N$ (the difference between the exact value and the value produced by the numerical method).

One way to estimate r goes as follows. For two successive experiments we have

$$E_i = Cn_i^r\,.$$

and

$$E_{i+1} = Cn_{i+1}^r$$

Divide the first equation by the second to eliminate C, and then take the logarithm to solve for r:

$$r = \frac{\ln(E_i/E_{i+1})}{\ln(n_i/n_{i+1})}\,.$$

We can compute r for all pairs of two successive experiments. Say r_i is the r value found from experiment i and $i + 1$,

$$r_i = \frac{\ln(E_i/E_{i+1})}{\ln(n_i/n_{i+1})}, \quad i = 0, 1, \ldots, N - 1\,.$$

Usually, the last value, r_{N-1}, is the best approximation to the true r value. Knowing r, we can compute C as $E_i n_i^{-r}$ for any i.

Use the method above to estimate r and C for the Midpoint method, the Trapezoidal method, and Simpson's method. Make your own choice of integral problem: $f(x)$, a, and b. Let the parameter n be the number of function evaluations in each method, and run the experiments with $n = 2^k + 1$ for $k = 2, \ldots, 11$. The `Integrator` hierarchy from Section 9.3 has all the requested methods implemented. Filename: `integrators_convergence.py`.

Exercise 9.16: Add common functionality in a class hierarchy

Suppose you want to use classes in the `Integrator` hierarchy from Section 9.3 to calculate integrals of the form

$$F(x) = \int_a^x f(t)dt\,.$$

Such functions $F(x)$ can be efficiently computed by the method from Exercise 7.22. Implement this computation of $F(x)$ in an additional method in the superclass `Integrator`. Test that the implementation is correct for $f(x) = 2x - 3$ for all the implemented integration methods (the Midpoint, Trapezoidal and Gauss-Legendre methods, as well as Simpson's rule, integrate a linear function exactly). Filename: `integrate_efficient.py`.

Exercise 9.17: Make a class hierarchy for root finding

Given a general nonlinear equation $f(x) = 0$, we want to implement classes for solving such an equation, and organize the classes in a class hierarchy. Make classes for three methods: Newton's method (Section A.1.10), the Bisection method (Section 4.10.2), and the Secant method (Exercise A.10).

It is not obvious how such a hierarchy should be organized. One idea is to let the superclass store the $f(x)$ function and its derivative $f'(x)$ (if provided - if not, use a finite difference approximation for $f'(x)$). A method

```
def solve(start_values=[0], max_iter=100, tolerance=1E-6):
    ...
```

in the superclass can implement a general iteration loop. The `start_values` argument is a list of starting values for the algorithm in question: one point for Newton, two for Secant, and an interval $[a, b]$ containing a root for Bisection. Let `solve` define a list `self.x` holding all the computed approximations. The initial value of `self.x` is simply `start_values`. For the Bisection method, one can use the convention $a, b, c =$ `self.x[-3:]`, where $[a, b]$ represents the most recently computed interval and c is its midpoint. The `solve` method can return an approximate root x, the corresponding $f(x)$ value, a boolean indicator that is `True` if $|f(x)|$ is less than the `tolerance` parameter, and a list of all the approximations and their f values (i.e., a list of $(x, f(x))$ tuples).

Do Exercise A.11 using the new class hierarchy. Filename: `Rootfinders.py`.

Exercise 9.18: Make a calculus calculator class

Given a function $f(x)$ defined on a domain $[a, b]$, the purpose of many mathematical exercises is to sketch the function curve $y = f(x)$, compute the derivative $f'(x)$, find local and global extreme points, and compute the integral $\int_a^b f(x)dx$. Make a class `CalculusCalculator` which can perform all these actions for any function $f(x)$ using numerical differentiation and integration, and the method explained in Exercise 7.33. for finding extrema.

Here is an interactive session with the class where we analyze $f(x) = x^2e^{-0.2x}\sin(2\pi x)$ on $[0, 6]$ with a grid (set of x coordinates) of 700 points:

```
>>> from CalculusCalculator import *
>>> def f(x):
...     return x**2*exp(-0.2*x)*sin(2*pi*x)
...
>>> c = CalculusCalculator(f, 0, 6, resolution=700)
>>> c.plot()                # plot f
>>> c.plot_derivative()     # plot f'
>>> c.extreme_points()

All minima: 0.8052, 1.7736, 2.7636, 3.7584, 4.7556, 5.754, 0
All maxima: 0.3624, 1.284, 2.2668, 3.2604, 4.2564, 5.2548, 6
Global minimum: 5.754
Global maximum: 5.2548

>>> c.integral
-1.7353776102348935
>>> c.df(2.51)    # c.df(x) is the derivative of f
-24.056988888465636
>>> c.set_differentiation_method(Central4)
>>> c.df(2.51)
-24.056988832723189
>>> c.set_integration_method(Simpson)  # more accurate integration
>>> c.integral
-1.7353857856973565
```

Design the class such that the above session can be carried out.

Hint. Use classes from the `Diff` and `Integrator` hierarchies (Sections 9.2 and 9.3) for numerical differentiation and integration (with, e.g., `Central2` and `Trapezoidal` as default methods for differentiation and integration). The method `set_differentiation_method` takes a subclass name in the `Diff` hierarchy as argument, and makes an attribute `df` that holds a subclass instance for computing derivatives. With `set_integration_method` we can similarly set the integration method as a subclass name in the `Integrator` hierarchy, and then compute the integral $\int_a^b f(x)dx$ and store the value in the attribute `integral`. The

`extreme_points` method performs a `print` on a `MinMax` instance, which is stored as an attribute in the calculator class.
Filename: `CalculusCalculator.py`.

Exercise 9.19: Compute inverse functions

Extend class `CalculusCalculator` from Exercise 9.18 to offer computations of inverse functions.

Hint. A numerical way of computing inverse functions is explained in Section A.1.11. Other, perhaps more attractive methods are described in Exercises E.17-E.20.
Filename: `CalculusCalculator2.py`.

Exercise 9.20: Make line drawing of a person; program

A very simple sketch of a human being can be made of a circle for the head, two lines for the arms, one vertical line, a triangle, or a rectangle for the torso, and two lines for the legs. Make such a drawing in a program, utilizing appropriate classes in the `Shape` hierarchy. Filename: `draw_person.py`.

Exercise 9.21: Make line drawing of a person; class

Use the code from Exercise 9.20 to make a subclass of `Shape` that draws a person. Supply the following arguments to the constructor: the center point of the head and the radius R of the head. Let the arms and the torso be of length $4R$, and the legs of length $6R$. The angle between the legs can be fixed (say 30 degrees), while the angle of the arms relative to the torso can be an argument to the constructor with a suitable default value. Filename: `Person.py`.

Exercise 9.22: Animate a person with waving hands

Make a subclass of the class from Exercise 9.21 where the constructor can take an argument describing the angle between the arms and the torso. Use this new class to animate a person who waves her/his hands.
Filename: `waving_person.py`.

```
    data = concatenate(tones)
    write(data, filename)
    data = read(filename)
    play(filename)

if __name__ == '__main__':
    try:
        seqtype = sys.argv[1]
        N = int(sys.argv[2])
    except IndexError:
        print 'Usage: %s oscillations|logistic N' % sys.argv[0]
        sys.exit(1)
    make_sound(N, seqtype)
```

This code should be quite easy to read at the present stage in the book. However, there is one statement that deserves a comment:

```
    x = eval(seqtype)(N)
```

The `seqtype` argument reflects the type of sequence and is a string that the user provides on the command line. The values of the string equal the function names `oscillations` and `logistic`. With `eval(seqtype)` we turn the string into a function name. For example, if `seqtype` is `'logistic'`, performing an `eval(seqtype)(N)` is the same as if we had written `logistic(N)`. This technique allows the user of the program to choose a function call inside the code. Without `eval` we would need to explicitly test on values:

```
if seqtype == 'logistic':
    x = logistic(N)
elif seqtype == 'oscillations':
    x = oscillations(N)
```

This is not much extra code to write in the present example, but if we have a large number of functions generating sequences, we can save a lot of boring if-else code by using the `eval` construction.

The next step, as a reader who have understood the problem and the implementation above, is to run the program for two cases: the `oscillations` sequence with $N = 40$ and the `logistic` sequence with $N = 100$. By altering the q parameter to lower values, you get other sounds, typically quite boring sounds for non-oscillating logistic growth ($q < 1$). You can also experiment with other transformations of the form (A.46), e.g., increasing the frequency variation from 200 to 400.

A.3 Exercises

Exercise A.1: Determine the limit of a sequence

a) Write a Python function for computing and returning the sequence

$$a_n = \frac{7 + 1/(n+1)}{3 - 1/(n+1)^2}, \quad n = 0, 2, \ldots, N\,.$$

Write out the sequence for $N = 100$. Find the exact limit as $N \to \infty$ and compare with a_N.

b) Write a Python function `limit(seq)` that takes a sequence of numbers as input, stored in a list or array `seq`, and returns the limit of the sequence, if it exists, otherwise `None` is returned. Test the `limit` function on the sequence in a) and on the divergent sequence $b_n = n$.

Hint. One possible quite strict test for determining if a sequence $(a_n)_{n=0}^N$ has a limit is to check

$$|a_n| - |a_{n+1}| < |a_{n-1}| - |a_n|,$$

for $n = 1, \ldots, N-1$.

c) Write a Python function for computing and returning the sequence

$$D_n = \frac{\sin(2^{-n})}{2^{-n}}, \quad n = 0, \ldots, N\,.$$

Call `limit` from b) to determine the limit of the sequence (for a sufficiently large N).

d) Given the sequence

$$D_n = \frac{f(x+h) - f(x)}{h}, \quad h = 2^{-n}, \tag{A.47}$$

make a function `D(f, x, N)` that takes a function $f(x)$, a value x, and the number N of terms in the sequence as arguments, and returns the sequence D_n for $n = 0, 1, \ldots, N$. Make a call to the `D` function with $f(x) = \sin x$, $x = 0$, and $N = 80$. Find the limit with aid of the `limit` function above. Plot the evolution of the computed D_n values, using small circles for the data points.

e) Make another call to `D` where $x = \pi$, let the `limit` function analyze the sequence, and plot this sequence in a separate figure. What would be your expected limit?

f) Explain why the computations for $x = \pi$ go wrong for large N.

Hint. Print out the numerator and denominator in D_n.
Filename: `sequence_limits.py`.

Exercise A.2: Compute π via sequences

The following sequences all converge to π:

$$
\begin{aligned}
(a_n)_{n=1}^{\infty}, &\quad a_n = 4\sum_{k=1}^{n}\frac{(-1)^{k+1}}{2k-1},\\
(b_n)_{n=1}^{\infty}, &\quad b_n = \left(6\sum_{k=1}^{n}k^{-2}\right)^{1/2},\\
(c_n)_{n=1}^{\infty}, &\quad c_n = \left(90\sum_{k=1}^{n}k^{-4}\right)^{1/4},\\
(d_n)_{n=1}^{\infty}, &\quad d_n = \frac{6}{\sqrt{3}}\sum_{k=0}^{n}\frac{(-1)^k}{3^k(2k+1)},\\
(e_n)_{n=1}^{\infty}, &\quad e_n = 16\sum_{k=0}^{n}\frac{(-1)^k}{5^{2k+1}(2k+1)} - 4\sum_{k=0}^{n}\frac{(-1)^k}{239^{2k+1}(2k+1)}\,.
\end{aligned}
$$

Make a function for each sequence that returns an array with the elements in the sequence. Plot all the sequences, and find the one that converges fastest toward the limit π. Filename: `pi_sequences.py`.

Exercise A.3: Reduce memory usage of difference equations

Consider the program `growth_years.py` from Section A.1.1. Since x_n depends on x_{n-1} only, we do not need to store all the $N+1$ x_n values. We actually only need to store x_n and its previous value x_{n-1}. Modify the program to use two variables and not an array for the entire sequence. Also avoid the `index_set` list and use an integer counter for n and a `while` loop instead. Write the sequence to file such that it can be visualized later. Filename: `growth_years_efficient.py`.

Exercise A.4: Compute the development of a loan

Solve (A.16)-(A.17) in a Python function. Filename: `loan.py`.

Exercise A.5: Solve a system of difference equations

Solve (A.32)-(A.33) in a Python function and plot the x_n sequence. Filename: `fortune_and_inflation1.py`.

Exercise A.6: Modify a model for fortune development

In the model (A.32)-(A.33) the new fortune is the old one, plus the interest, minus the consumption. During year n, x_n is normally also reduced with t percent tax on the earnings $x_{n-1} - x_{n-2}$ in year $n-1$.

a) Extend the model with an appropriate tax term, implement the model, and demonstrate in a plot the effect of tax ($t = 27$) versus no tax ($t = 0$).

b) Suppose you expect to live for N years and can accept that the fortune x_n vanishes after N years. Choose some appropriate values for p, q, I, and t, and experiment with the program to find how large the initial c_0 can be in this case.
Filename: `fortune_and_inflation2.py`.

Exercise A.7: Change index in a difference equation

A mathematically equivalent equation to (A.5) is

$$x_{i+1} = x_i + \frac{p}{100}x_i, \tag{A.48}$$

since the name of the index can be chosen arbitrarily. Suppose someone has made the following program for solving (A.48):

```
from scitools.std import *
x0 = 100                # initial amount
p = 5                   # interest rate
N = 4                   # number of years
index_set = range(N+1)
x = zeros(len(index_set))

# Compute solution
x[0] = x0
for i in index_set[1:]:
    x[i+1] = x[i] + (p/100.0)*x[i]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

This program does not work. Make a correct version, but keep the difference equations in its present form with the indices `i+1` and `i`. Filename: `growth1_index_ip1.py`.

Exercise A.8: Construct time points from dates

A certain quantity p (which may be an interest rate) is piecewise constant and undergoes changes at some specific dates, e.g.,

$$p \text{ changes to } \begin{cases} 4.5 & \text{on Jan 4, 2019} \\ 4.75 & \text{on March 21, 2019} \\ 6.0 & \text{on April 1, 2019} \\ 5.0 & \text{on June 30, 2019} \\ 4.5 & \text{on Nov 1, 2019} \\ 2.0 & \text{on April 1, 2020} \end{cases} \tag{A.49}$$

Given a start date d_1 and an end date d_2, fill an array `p` with the right p values, where the array index counts days. Use the `datetime` module to compute the number of days between dates. Filename: `dates2days.py`.

Exercise A.9: Visualize the convergence of Newton's method

Let $x_0, x_1, \ldots, x_N$ be the sequence of roots generated by Newton's method applied to a nonlinear algebraic equation $f(x) = 0$ (see Section A.1.10). In this exercise, the purpose is to plot the sequences $(x_n)_{n=0}^N$ and $(|f(x_n)|)_{n=0}^N$ such that we can understand how Newton's method converges or diverges.

a) Make a general function

```
Newton_plot(f, x, dfdx, xmin, xmax, epsilon=1E-7)
```

for this purpose. The arguments `f` and `dfdx` are Python functions representing the $f(x)$ function in the equation and its derivative $f'(x)$, respectively. Newton's method is run until $|f(x_N)| \leq \epsilon$, and the ϵ value is available as the `epsilon` argument. The `Newton_plot` function should make three separate plots of $f(x)$, $(x_n)_{n=0}^N$, and $(|f(x_n)|)_{n=0}^N$ on the screen and also save these plots to PNG files. The relevant x interval for plotting of $f(x)$ is given by the arguments `xmin` and `xmax`. Because of the potentially wide scale of values that $|f(x_n)|$ may exhibit, it may be wise to use a logarithmic scale on the y axis.

Hint. You can save quite some coding by calling the improved `Newton` function from Section A.1.10, which is available in the module file `Newton.py`.

b) Demonstrate the function on the equation $x^6 \sin \pi x = 0$, with $\epsilon = 10^{-13}$. Try different starting values for Newton's method: $x_0 = -2.6, -1.2, 1.5, 1.7, 0.6$. Compare the results with the exact solutions $x = \ldots, -2 - 1, 0, 1, 2, \ldots$.

c) Use the `Newton_plot` function to explore the impact of the starting point x_0 when solving the following nonlinear algebraic equations:

$$\sin x = 0, \tag{A.50}$$

$$x = \sin x, \tag{A.51}$$

$$x^5 = \sin x, \tag{A.52}$$

$$x^4 \sin x = 0, \tag{A.53}$$

$$x^4 = 16, \tag{A.54}$$

$$x^{10} = 1, \tag{A.55}$$

$$\tanh x = 0 \,.\, \tanh x \qquad = x^{10} \,. \tag{A.56}$$

Hint. Such an experimental investigation is conveniently recorded in an IPython notebook. See Section H.1.9 for a quick introduction to notebooks.
Filename: `Newton2.py`.

Exercise A.10: Implement the secant method

Newton's method (A.34) for solving $f(x) = 0$ requires the derivative of the function $f(x)$. Sometimes this is difficult or inconvenient. The derivative can be approximated using the last two approximations to the root, x_{n-2} and x_{n-1}:

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}\,.$$

Using this approximation in (A.34) leads to the Secant method:

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}, \quad x_0, x_1 \text{ given}\,. \tag{A.57}$$

Here $n = 2, 3, \ldots$. Make a program that applies the Secant method to solve $x^5 = \sin x$. Filename: `Secant.py`.

Exercise A.11: Test different methods for root finding

Make a program for solving $f(x) = 0$ by Newton's method (Section A.1.10), the Bisection method (Section 4.10.2), and the Secant method (Exercise A.10). For each method, the sequence of root approximations should be written out (nicely formatted) on the screen. Read $f(x)$, $f'(x)$, a, b, x_0, and x_1 from the command line. Newton's method starts with x_0, the Bisection method starts with the interval $[a, b]$, whereas the Secant method starts with x_0 and x_1.

Run the program for each of the equations listed in Exercise A.9d. You should first plot the $f(x)$ functions so you know how to choose x_0, x_1, a, and b in each case. Filename: `root_finder_examples.py`.

Exercise A.12: Make difference equations for the Midpoint rule

Use the ideas of Section A.1.7 to make a similar system of difference equations and corresponding implementation for the Midpoint integration rule:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(a - \frac{1}{2}h + ih),$$

where $h = (b - a)/n$ and n counts the number of function evaluations (i.e., rectangles that approximate the area under the curve). Filename: `diffeq_midpoint.py`.

Exercise A.13: Compute the arc length of a curve

Sometimes one wants to measure the length of a curve $y = f(x)$ for $x \in [a, b]$. The arc length from $f(a)$ to some point $f(x)$ is denoted by $s(x)$ and defined through an integral

$$s(x) = \int_a^x \sqrt{1 + [f'(\xi)]^2} d\xi\,. \qquad \text{(A.58)}$$

We can compute $s(x)$ via difference equations as explained in Section A.1.7.

a) Make a Python function `arclength(f, a, b, n)` that returns an array `s` with $s(x)$ values for n uniformly spaced coordinates x in $[a, b]$. Here `f(x)` is the Python implementation of the function that defines the curve we want to compute the arc length of.

b) How can you verify that the `arclength` function works correctly? Construct test case(s) and write corresponding test functions for automating the tests.

Hint. Check the implementation for curves with known arc length, e.g., a semi-circle and a straight line.

c) Apply the function to

$$f(x) = \int_{-2}^x = \frac{1}{\sqrt{2\pi}} e^{-4t^2} dt, \quad x \in [-2, 2]\,.$$

Compute $s(x)$ and plot it together with $f(x)$.
Filename: `arclength.py`.

Exercise A.14: Find difference equations for computing $\sin x$

The purpose of this exercise is to derive and implement difference equations for computing a Taylor polynomial approximation to $\sin x$:

$$\sin x \approx S(x; n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}\,. \qquad \text{(A.59)}$$

To compute $S(x; n)$ efficiently, write the sum as $S(x; n) = \sum_{j=0}^{n} a_j$, and derive a relation between two consecutive terms in the series:

$$a_j = -\frac{x^2}{(2j+1)2j} a_{j-1}\,. \qquad \text{(A.60)}$$

Introduce $s_j = S(x; j-1)$ and a_j as the two sequences to compute. We have $s_0 = 0$ and $a_0 = x$.

a) Formulate the two difference equations for s_j and a_j.

Hint. Section A.1.8 explains how this task and the associated programming can be solved for the Taylor polynomial approximation of e^x.

b) Implement the system of difference equations in a function `sin_Taylor(x, n)`, which returns s_{n+1} and $|a_{n+1}|$. The latter is the first neglected term in the sum (since $s_{n+1} = \sum_{j=0}^{n} a_j$) and may act as a rough measure of the size of the error in the Taylor polynomial approximation.

c) Verify the implementation by computing the difference equations for $n = 2$ by hand (or in a separate program) and comparing with the output from the `sin_Taylor` function. Automate this comparison in a test function.

d) Make a table of s_n for various x and n values to verify that the accuracy of a Taylor polynomial improves as n increases and x decreases. Be aware of the fact that `sine_Taylor(x, n)` can give extremely inaccurate approximations to $\sin x$ if x is not sufficiently small and n sufficiently large.
Filename: `sin_Taylor_series_diffeq.py`.

Exercise A.15: Find difference equations for computing $\cos x$

Solve Exercise A.14 for the Taylor polynomial approximation to $\cos x$. (The relevant expression for the Taylor series is easily found in a mathematics textbook or by searching on the Internet.) Filename: `cos_Taylor_series_diffeq.py`.

Exercise A.16: Make a guitar-like sound

Given start values $x_0, x_1, \ldots, x_p$, the following difference equation is known to create guitar-like sound:

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}), \quad n = p+1, \ldots, N\,. \tag{A.61}$$

With a sampling rate r, the frequency of this sound is given by r/p. Make a program with a function `solve(x, p)` which returns the solution array `x` of (A.61). To initialize the array `x[0:p+1]` we look at two methods, which can be implemented in two alternative functions:

- $x_0 = 1$, $x_1 = x_2 = \cdots = x_p = 0$
- $x_0, \ldots, x_p$ are uniformly distributed random numbers in $[-1, 1]$

Import `max_amplitude`, `write`, and `play` from the `scitools.sound` module. Choose a sampling rate r and set $p = r/440$ to create a 440 Hz tone (A). Create an array `x1` of zeros with length $3r$ such that the tone will last for 3 seconds. Initialize `x1` according to method 1 above and solve (A.61). Multiply the `x1` array by `max_amplitude`. Repeat this process for an array `x2` of length $2r$, but use method 2 for the initial values and choose p such that the tone is 392 Hz (G). Concatenate `x1` and `x2`, call `write` and then `play` to play the sound. As you will experience, this sound is amazingly similar to the sound of a guitar string, first playing A for 3 seconds and then playing G for 2 seconds.

The method (A.61) is called the Karplus-Strong algorithm and was discovered in 1979 by a researcher, Kevin Karplus, and his student Alexander Strong, at Stanford University. Filename: `guitar_sound.py`.

Exercise A.17: Damp the bass in a sound file

Given a sequence $x_0, \ldots, x_{N-1}$, the following *filter* transforms the sequence to a new sequence $y_0, \ldots, y_{N-1}$:

$$y_n = \begin{cases} x_n, & n = 0 \\ -\frac{1}{4}(x_{n-1} - 2x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.62}$$

If x_n represents sound, y_n is the same sound but with the bass damped. Load some sound file, e.g.,

```
x = scitools.sound.Nothing_Else_Matters()
# or
x = scitools.sound.Ja_vi_elsker()
```

to get a sound sequence. Apply the filter (A.62) and play the resulting sound. Plot the first 300 values in the x_n and y_n signals to see graphically what the filter does with the signal. Filename: `damp_bass.py`.

Exercise A.18: Damp the treble in a sound file

Solve Exercise A.17 to get some experience with coding a filter and trying it out on a sound. The purpose of this exercise is to explore some other filters that reduce the treble instead of the bass. Smoothing the sound signal will in general damp the treble, and smoothing is typically obtained by letting the values in the new filtered sound sequence be an average of the neighboring values in the original sequence.

The simplest smoothing filter can apply a standard average of three neighboring values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{3}(x_{n-1} + x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.63}$$

Two other filters put less emphasis on the surrounding values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.64}$$

$$y_n = \begin{cases} x_n, & n = 0,1 \\ \frac{1}{16}(x_{n-2} + 4x_{n-1} + 6x_n + 4x_{n+1} + x_{n+2}), & 2 \leq n \leq N-3 \\ x_n, & n = N-2, N-1 \end{cases} \tag{A.65}$$

Apply all these three filters to a sound file and listen to the result. Plot the first 300 values in the x_n and y_n signals for each of the three filters to see graphically what the filter does with the signal. Filename: `damp_treble.py`.

Exercise A.19: Demonstrate oscillatory solutions of (A.13)

a) Write a program to solve the difference equation (A.13):

$$y_n = y_{n-1} + q y_{n-1} (1 - y_{n-1}), \quad n = 0, \ldots, N\,.$$

Read the input parameters y_0, q, and N from the command line. The variables and the equation are explained in Section A.1.5.

b) Equation (A.13) has the solution $y_n = 1$ as $n \to \infty$. Demonstrate, by running the program, that this is the case when $y_0 = 0.3$, $q = 1$, and $N = 50$.

c) For larger q values, y_n does not approach a constant limit, but y_n oscillates instead around the limiting value. Such oscillations are sometimes observed in wildlife populations. Demonstrate oscillatory solutions when q is changed to 2 and 3.

d) It could happen that y_n stabilizes at a constant level for larger N. Demonstrate that this is not the case by running the program with $N = 1000$.
Filename: `growth_logistic2.py`.

Exercise A.20: Automate computer experiments

It is tedious to run a program like the one from Exercise A.19 repeatedly for a wide range of input parameters. A better approach is to let the computer do the manual work. Modify the program from Exercise A.19 such that the computation of y_n and the plot is made in a function. Let the title in the plot contain the parameters y_0 and q (N is easily visible from the x axis). Also let the name of the plot file reflect the values of y_0, q, and N. Then make loops over y_0 and q to perform the following more comprehensive set of experiments:

- $y_0 = 0.01, 0.3$
- $q = 0.1, 1, 1.5, 1.8, 2, 2.5, 3$
- $N = 50$

How does the initial condition (the value y_0) seem to influence the solution?

Hint. If you do no want to get a lot of plots on the screen, which must be killed, drop the call to `show()` in Matplotlib or use `show=False` as argument to `plot` in SciTools.
Filename: `growth_logistic3.py`.

Exercise A.21: Generate an HTML report

Extend the program made in Exercise A.20 with a report containing all the plots. The report can be written in HTML and displayed by a web browser. The plots must then be generated in PNG format. The source of the HTML file will typically look as follows:

```
<html>
<body>
<p><img src="tmp_y0_0.01_q_0.1_N_50.png">
<p><img src="tmp_y0_0.01_q_1_N_50.png">
<p><img src="tmp_y0_0.01_q_1.5_N_50.png">
<p><img src="tmp_y0_0.01_q_1.8_N_50.png">
...
<p><img src="tmp_y0_0.01_q_3_N_1000.png">
</html>
</body>
```

Let the program write out the HTML text to a file. You may let the function making the plots return the name of the plot file such that this string can be inserted in the HTML file. Filename: `growth_logistic4.py`.

Exercise A.22: Use a class to archive and report experiments

The purpose of this exercise is to make the program from Exercise A.21 more flexible by creating a Python class that runs and archives all the experiments (provided you know how to program with Python classes). Here is a sketch of the class:

```
class GrowthLogistic:
    def __init__(self, show_plot_on_screen=False):
        self.experiments = []
        self.show_plot_on_screen = show_plot_on_screen
        self.remove_plot_files()

    def run_one(self, y0, q, N):
        """Run one experiment."""
        # Compute y[n] in a loop...
        plotfile = 'tmp_y0_%g_q_%g_N_%d.png' % (y0, q, N)
        self.experiments.append({'y0': y0, 'q': q, 'N': N,
                                 'mean': mean(y[20:]),
                                 'y': y, 'plotfile': plotfile})
                               # Make plot...

                           def run_many(self, y0_list, q_list, N):
                               """Run many experiments."""
                               for q in q_list:
                                   for y0 in y0_list:
                                       self.run_one(y0, q, N)

                           def remove_plot_files(self):
                               """Remove plot files with names tmp_y0*.png."""
                               import os, glob
                               for plotfile in glob.glob('tmp_y0*.png'):
                                   os.remove(plotfile)

                           def report(self, filename='tmp.html'):
                               """
                               Generate an HTML report with plots of all
                               experiments generated so far.
                               """
                               # Open file and write HTML header...
                               for e in self.experiments:
                                   html.write('<p><img src="%s">\n' % e['plotfile'])
                               # Write HTML footer and close file...
```

Each time the `run_one` method is called, data about the current experiment is stored in the `experiments` list. Note that `experiments` contains a list of dictionaries. When desired, we can call the `report` method to collect all the plots made so far in an HTML report. A typical use of the class goes as follows:

```
N = 50
g = GrowthLogistic()
g.run_many(y0_list=[0.01, 0.3],
           q_list=[0.1, 1, 1.5, 1.8] + [2, 2.5, 3], N=N)
g.run_one(y0=0.01, q=3, N=1000)
g.report()
```

Make a complete implementation of class `GrowthLogistic` and test it with the small program above. The program file should be constructed as a module. Filename: `growth_logistic5.py`.

Exercise A.23: Explore logistic growth interactively

Class `GrowthLogistic` from Exercise A.22 is very well suited for interactive exploration. Here is a possible sample session for illustration:

```
>>> from growth_logistic5 import GrowthLogistic
>>> g = GrowthLogistic(show_plot_on_screen=True)
>>> q = 3
>>> g.run_one(0.01, q, 100)
>>> y = g.experiments[-1]['y']
>>> max(y)
1.3326056469620293
>>> min(y)
0.0029091569028512065
```

Extend this session with an investigation of the oscillations in the solution y_n. For this purpose, make a function for computing the local maximum values y_n and the corresponding indices where these local maximum values occur. We can say that y_i is a local maximum value if

$$y_{i-1} < y_i > y_{i+1} \,.$$

Plot the sequence of local maximum values in a new plot. If $I_0, I_1, I_2, \ldots$ constitute the set of increasing indices corresponding to the local maximum values, we can define the periods of the oscillations as $I_1 - I_0$, $I_2 - I_1$, and so forth. Plot the length of the periods in a separate plot. Repeat this investigation for $q = 2.5$. Filename: `GrowthLogistic_interactive.py`.

Exercise A.24: Simulate the price of wheat

The demand for wheat in year t is given by

$$D_t = ap_t + b,$$

where $a < 0$, $b > 0$, and p_t is the price of wheat. Let the supply of wheat be

$$S_t = Ap_{t-1} + B + \ln(1 + p_{t-1}),$$

where A and B are given constants. We assume that the price p_t adjusts such that all the produced wheat is sold. That is, $D_t = S_t$.

a) For $A = 1$, $a = -3, b = 5, B = 0$, find from numerical computations, a stable price such that the production of wheat from year to year is constant. That is, find p such that $ap + b = Ap + B + \ln(1 + p)$.

b) Assume that in a very dry year the production of wheat is much less than planned. Given that price this year, p_0, is 4.5 and $D_t = S_t$, compute in a program how the prices $p_1, p_2, \ldots, p_N$ develop. This implies solving the difference equation

$$ap_t + b = Ap_{t-1} + B + \ln(1 + p_{t-1}) \,.$$

From the p_t values, compute S_t and plot the points (p_t, S_t) for $t = 0, 1, 2, \ldots, N$. How do the prices move when $N \to \infty$?
Filename: `wheat.py`.

is necessary to derive what the additional terms are, but implementing the terms is trivial in our program above (do Exercise E.39).

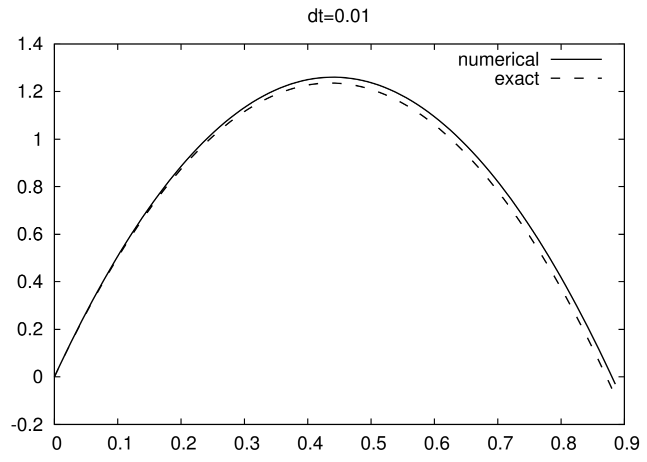


Fig. E.8 The trajectory of a ball solved as a system of four ODEs by the Forward Euler method.

E.3.9 Further developments of ODESolver

The `ODESolver` hierarchy is a simplified prototype version of a more professional Python package for solving ODEs called Odespy. This package features a range of simple and sophisticated methods for solving scalar ODEs and systems of ODEs. Some of the solvers are implemented in Python, while others call up well-known ODE software in Fortran. Like the `ODESolver` hierarchy, Odespy offers a unified interface to the different numerical methods, which means that the user can specify the ODE problem as a function `f(u,t)` and send this function to all solvers. This feature makes it easy to switch between solvers to test a wide collection of numerical methods for a problem.

Odespy can be downloaded from `http://hplgit.github.com/odespy`. It is installed by the usual `python setup.py install` command.

E.4 Exercises

Exercise E.1: Solve a simple ODE with function-based code

This exercise aims to solve the ODE problem $u - 10u' = 0$, $u(0) = 0.2$, for $t \in [0, 20]$.

a) Identify the mathematical function $f(u,t)$ in the generic ODE form $u' = f(u,t)$.

b) Implement the $f(u,t)$ function in a Python function.

c) Use the `ForwardEuler` function from Section E.1.3 to compute a numerical solution of the ODE problem. Use a time step $\Delta t = 5$.

d) Plot the numerical solution and the exact solution $u(t) = 0.2e^{-0.1t}$.

e) Save the numerical solution to file. Decide upon a suitable file format.

f) Perform simulations for smaller Δt values and demonstrate visually that the numerical solution approaches the exact solution.
Filename: `simple_ODE_func.py`.

Exercise E.2: Solve a simple ODE with class-based code

Perform Exercise E.1, but solve the ODE using the `ForwardEuler` class from Section E.1.7. Implement the right-hand side function f as a class too. Filename: `simple_ODE_class.py`.

Exercise E.3: Solve a simple ODE with the ODEsolver hierarchy

Redo Exercise E.1, but use the `ForwardEuler` class in the `ODESolver` hierarchy from Section E.3. Filename: `simple_ODE_class_ODESolver.py`.

Exercise E.4: Solve an ODE specified on the command line

We want to make a program `odesolver_cml.py` which accepts an ODE problem to be specified on the command line. The command-line arguments are `f u0 dt T`, where `f` is the right-hand side $f(u,t)$ specified as a string formula, `u0` is the initial condition, `dt` is the time step, and `T` is the final time of the simulation. A fifth optional argument can be given to specify the name of the numerical solution method (set any method of your choice as default value). A curve plot of the solution versus time should be produced and stored in a file `plot.png`.

Hint 1. Use `StringFunction` from `scitools.std` for convenient conversion of a formula on the command line to a Python function.

Hint 2. Use the `ODESolver` hierarchy to solve the ODE and let the fifth command-line argument be the class name in the `ODESolver` hierarchy.
Filename: `odesolver_cml.py`.

Exercise E.5: Implement a numerical method for ODEs

Implement the numerical method (E.36)-(E.37). How can you verify that the implementation is correct? Filename: `Heuns_method.py`.

Exercise E.6: Solve an ODE for emptying a tank

A cylindrical tank of radius R is filled with water to a height h_0. By opening a valve of radius r at the bottom of the tank, water flows out, and the height of water, $h(t)$, decreases with time. We can derive an ODE that governs the height function $h(t)$.

Mass conservation of water requires that the reduction in height balances the outflow. In a time interval Δt, the height is reduced by Δh, which corresponds to a water volume of $\pi R^2 \Delta h$. The water leaving the tank in the same interval of time equals $\pi r^2 v \Delta t$, where v is the outflow velocity. It can be shown (from what is known as Bernoulli's equation) [16, 27] that

$$v(t) = \sqrt{2gh(t) + h'(t)^2},$$

where g is the acceleration of gravity. Note that $\Delta h > 0$ implies an increase in h, which means that $-\pi R^2 \Delta h$ is the corresponding decrease in volume that must balance the outflow loss of volume $\pi r^2 v \Delta t$. Elimination of v and taking the limit $\Delta t \to 0$ lead to the ODE

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \left(1 - \left(\frac{r}{R}\right)^4\right)^{-1/2} \sqrt{2gh}\,.$$

For practical applications $r \ll R$ so that $1 - (r/R)^4 \approx 1$ is a reasonable approximation, since other approximations are done as well: friction is neglected in the derivation, and we are going to solve the ODE by approximate numerical methods. The final ODE then becomes

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \sqrt{2gh}\,. \tag{E.61}$$

The initial condition follows from the initial height of water, h_0, in the tank: $h(0) = h_0$.

Solve (E.61) by a numerical method of your choice in a program. Set $r = 1$ cm, $R = 20$ cm, $g = 9.81$ m/s^2, and $h_0 = 1$ m. Use a time step of 10 seconds. Plot the solution, and experiment to see what a proper time interval for the simulation is. Make sure to test for $h < 0$ so that you do not apply the square root function to negative numbers. Can you find an analytical solution of the problem to compare the numerical solution with? Filename: `tank_ODE.py`.

Exercise E.7: Solve an ODE for the arc length

Given a curve $y = f(x)$, the length of the curve from $x = x_0$ to some point x is given by the function $s(x)$, which solves the problem

$$\frac{ds}{dx} = \sqrt{1 + [f'(x)]^2}, \quad s(x_0) = 0\,. \tag{E.62}$$

Since s does not enter the right-hand side, (E.62) can immediately be integrated from x_0 to x (see Exercise A.13). However, we shall solve (E.62) as an ODE. Use the Forward Euler method and compute the length of a straight line (for easy verification) and a parabola: $f(x) = \frac{1}{2}x + 1$, $x \in [0, 2]$; $f(x) = x^2$, $x \in [0, 2]$. Filename: `arclength_ODE.py`.

Exercise E.8: Simulate a falling or rising body in a fluid

A body moving vertically through a fluid (liquid or gas) is subject to three different types of forces:

- the gravity force $F_g = -mg$, where m is the mass of the body and g is the acceleration of gravity;
- the drag force $F_d = -\frac{1}{2}C_D\varrho A|v|v$, where C_D is a dimensionless drag coefficient depending on the body's shape, ϱ is the density of the fluid, A is the cross-sectional area (produced by a cut plane, perpendicular to the motion, through the thickest part of the body), and v is the velocity;
- the uplift or buoyancy ("Archimedes") force $F_b = \varrho g V$, where V is the volume of the body.

(Roughly speaking, the F_d formula is suitable for medium to high velocities, while for very small velocities, or very small bodies, F_d is proportional to the velocity, not the velocity squared, see [27].)

Newton's second law applied to the body says that the sum of the listed forces must equal the mass of the body times its acceleration a:

$$F_g + F_d + F_b = ma,$$

which gives

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho g V = ma\,.$$

The unknowns here are v and a, i.e., we have two unknowns but only one equation. From kinematics in physics we know that the acceleration is the time derivative of the velocity: $a = dv/dt$. This is our second equation. We can easily eliminate a and get a single differential equation for v:

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho g V = m\frac{dv}{dt}\,.$$

A small rewrite of this equation is handy: we express m as $\varrho_b V$, where ϱ_b is the density of the body, and we isolate dv/dt on the left-hand side,

$$\frac{dv}{dt} = -g\left(1 - \frac{\varrho}{\varrho_b}\right) - \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,. \tag{E.63}$$

This differential equation must be accompanied by an initial condition: $v(0) = V_0$.

a) Make a program for solving (E.63) numerically, using any numerical method of your choice.

Hint. It is not strictly necessary, but it is an elegant Python solution to implement the right-hand side of (E.63) in the `__call__` method of a class where the parameters g, ϱ, ϱ_b, C_D, A, and V are attributes.

b) To verify the program, assume a heavy body in air such that the F_b force can be neglected, and assume a small velocity such that the air resistance F_d can also be neglected. Mathematically, setting $\varrho = 0$ removes both these terms from the equation. The solution is then $v(t) = y'(t) = v_0 - gt$. Observe through experiments that the linear solution is exactly reproduced by the numerical solution regardless of the value of Δt. (Note that if you use the Forward Euler method, the method can become unstable for large Δt, see Section E.3.5. and time steps above the critical limit for stability cannot be used to reproduce the linear solution.) Write a test function for automatically checking that the numerical solution is $u_k = v_0 - gk\Delta t$ in this test case.

c) Make a function for plotting the forces F_g, F_b, and F_d as functions of t. Seeing the relative importance of the forces as time develops gives an increased understanding of how the different forces contribute to changing the velocity.

d) Simulate a skydiver in free fall before the parachute opens. We set the density of the human body as $\varrho_b = 1003$ kg/m^3 and the mass as $m = 80$ kg, implying $V = m/\varrho_b = 0.08$ m^3. We can base the cross-sectional area A the assumption of a circular cross section of diameter 50 cm, giving $A = \pi R^2 = 0.9$ m^2. The density of air decreases with height, and we here use the value 0.79 kg/m^3 which is relevant for about 5000 m height. The C_D coefficient can be set as 0.6. Start with $v_0 = 0$.

e) A ball with the size of a soccer ball is placed in deep water, and we seek to compute its motion upwards. Contrary to the former example, where the buoyancy force F_b is very small, F_b is now the driving force, and the gravity force F_g is small. Set $A = \pi a^2$ with $a = 11$ cm. The mass of the ball is 0.43 kg, the density of water is 1000 kg/m^3, and C_D is 0.2. Start with $v_0 = 0$.
Filename: `body_in_fluid.py`.

Exercise E.9: Verify the limit of a solution as time grows

The solution of (E.63) often tends to a constant velocity, called the terminal velocity. This happens when the sum of the forces, i.e., the right-hand side in (E.63), vanishes.

a) Compute the formula for the terminal velocity by hand.

b) Solve the ODE using class `ODESolver` and call the `solve` method with a `terminate` function that terminates the computations when a constant velocity is reached, that is, when $|v(t_n) - v(t_{n-1})| \leq \epsilon$, where ϵ is a small number.

c) Run a series of Δt values and make a graph of the terminal velocity as a function of Δt for the two cases in Exercise E.8 d) and e). Indicate the exact terminal velocity in the plot by a horizontal line.
Filename: `body_in_fluid_termvel.py`.

Exercise E.10: Scale the logistic equation

Consider the logistic model (E.5):

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = U_0\,.$$

This problem involves three input parameters: U_0, R, and α. Learning how u varies with U_0, R, and α requires much experimentation where we vary all three parameters and observe the solution. A much more effective approach is to *scale* the problem. By this technique the solution depends only on one parameter: U_0/R. This exercise tells how the scaling is done.

The idea of scaling is to introduce *dimensionless* versions of the independent and dependent variables:

$$v = \frac{u}{u_c}, \quad \tau = \frac{t}{t_c},$$

where u_c and t_c are characteristic sizes of u and t, respectively, such that the dimensionless variables v and τ are of approximately unit size. Since we know that $u \to R$ as $t \to \infty$, R can be taken as the characteristic size of u.

Insert $u = Rv$ and $t = t_c\tau$ in the governing ODE and choose $t_c = 1/\alpha$. Show that the ODE for the new function $v(\tau)$ becomes

$$\frac{dv}{d\tau} = v(1 - v), \quad v(0) = v_0\,. \tag{E.64}$$

We see that the three parameters U_0, R, and α have disappeared from the ODE problem, and only one parameter $v_0 = U_0/R$ is involved.

Show that if $v(\tau)$ is computed, one can recover $u(t)$ by

$$u(t) = Rv(\alpha t)\,. \tag{E.65}$$

Geometrically, the transformation from v to u is just a stretching of the two axis in the coordinate system.

Make a program `logistic_scaled.py` where you compute $v(\tau)$, given $v_0 = 0.05$, and then you use (E.65) to plot $u(t)$ for $R = 100, 500, 1000$ and $\alpha = 1$ in one figure, and $u(t)$ for $\alpha = 1, 5, 10$ and $R = 1000$ in another figure. Note how effectively you can generate $u(t)$ without needing to solve an ODE problem, and also note how varying R and α impacts the graph of $u(t)$. Filename: `logistic_scaled.py`.

Exercise E.11: Compute logistic growth with time-varying carrying capacity

Use classes `Problem2` and `AutoSolver` from Section E.3.6 to study logistic growth when the carrying capacity of the environment, R, changes periodically with time: $R = 500$ for $it_s \leq t < (i+1)t_s$ and $R = 200$ for $(i+1)t_s \leq t < (i+2)t_s$, with $i = 0, 2, 4, 6, \ldots$. Use the same data as in Section E.3.6, and find some relevant sizes of the period of variation, t_s, to experiment with. Filename: `seasonal_logistic_growth.py`.

Exercise E.12: Solve an ODE until constant solution

Newton's law of cooling,

$$\frac{dT}{dt} = -h(T - T_s) \tag{E.66}$$

can be used to see how the temperature T of an object changes because of heat exchange with the surroundings, which have a temperature T_s. The parameter h, with unit s^{-1} is an experimental constant (heat transfer coefficient) telling how efficient the heat exchange with the surroundings is. For example, (E.66) may model the cooling of a hot pizza taken out of the oven. The problem with applying (E.66) is that h must be measured. Suppose we have measured T at $t = 0$ and t_1. We can use a rough Forward Euler approximation of (E.66) with one time step of length t_1,

$$\frac{T(t_1) - T(0)}{t_1} = -h(T(0) - T_s),$$

to make the estimate

$$h = \frac{T(t_1) - T(0)}{t_1(T_s - T(0))}\,. \tag{E.67}$$

a) The temperature of a pizza is 200 C at $t = 0$, when it is taken out of the oven, and 180 C after 50 seconds, in a room with temperature 20 C. Find an estimate of h from the formula above.

b) Solve (E.66) numerically by a method of your choice to find the evolution of the temperature of the pizza. Plot the solution.

Hint. You may solve the ODE the way you like, but the `solve` method in the classes in the `ODESolver` hierarchy accepts an optional `terminate` function that can be used to terminate the solution process when T is sufficiently close to T_s. Reading the first part of Section E.3.6 may be useful.
Filename: `pizza_cooling1.py`.

Exercise E.13: Use a problem class to hold data about an ODE

We address the same physical problem as in Exercise E.12, but we will now provide a class-based implementation for the user's code.

a) Make a class `Problem` containing the parameters h, T_s, $T(0)$, and Δt as attributes. A method `estimate_h` should take t_1 and $T(t_1)$ as arguments, compute h, and assign it to `self.h`. The right-hand side of the ODE can be implemented in a `__call__` method. If you use a class from the `ODESolver` hierarchy to solve the ODE, include the `terminate` function as a method in class `Problem`.

b) Implement a test function `test_Problem()` for testing that class `Problem` works. It is up to you to define how to test the class.

c) What are the advantages and disadvantages with class `Problem` compared to using plain functions (in your view)?

d) We now want to run experiments with different values of some parameters: $T_s = 15, 22, 30$ C and $T(0) = 250, 200$ C. For each $T(0)$, plot T for the three T_s values.

Hint. The typical elegant Python way to solve such a problem goes as follows. Write a function `solve(problem)` that takes a `Problem` object with name `problem` as argument and performs what it takes to solve one case (i.e., `solve` must solve the ODE and plot the solution). A dictionary can for each T_0 value hold a list of the cases to be plotted together. Then we loop over the problems in the dictionary of lists and call `solve` for each problem:

```
# Given h and dt
problems = {T_0: [Problem(h, T_s, T_0, dt)
                  for T_s in 15, 22, 30] for T_0 in 250, 200}
for T_0 in problems:
    hold('off')
    for problem in problems[T_0]:
        solve(problem)
        hold('on')
    savefig('T0_%g'.pdf % T_0)
```

When you become familiar with such code, and appreciate it, you can call yourself a professional programmer - with a deep knowledge of how lists, dictionaries, and classes can play elegantly together to conduct scientific experiments. In the present case we perform only a few experiments that could also have been done by six separate calls to the solver functionality, but in large-scale scientific and engineering investigations with hundreds of parameter combinations, the above code is still the same, only the generation of the `Problem` instances becomes more involved.
Filename: `pizza_cooling2.py`.

Exercise E.14: Derive and solve a scaled ODE problem

Use the scaling approach outlined in Exercise E.10 to "scale away" the parameters in the ODE in Exercise E.12. That is, introduce a new unknown $u = (T - T_s)/(T(0) - T_s)$ and a new time scale $\tau = th$. Find the ODE and the initial condition that governs the $u(\tau)$ function. Make a program that computes $u(\tau)$ until $|u| < 0.001$. Store the discrete u and τ values in a file `u_tau.dat` if that file is not already present (you can use `os.path.isfile(f)` to test if a file with name `f` exists). Create a function `T(u, tau, h, T0, Ts)` that loads the u and τ data from the `u_tau.dat` file and returns two arrays with T and t values, corresponding to the computed arrays for u and τ. Plot T versus t. Give the parameters h, T_s, and $T(0)$ on the command line. Note that this program is supposed to solve the ODE once and then recover any $T(t)$ solution by a simple scaling of the single $u(\tau)$ solution. Filename: `pizza_cooling3.py`.

Exercise E.15: Clean up a file to make it a module

The `ForwardEuler_func.py` file is not well organized to be used as a module, say for doing

```
>>> from ForwardEuler_func import ForwardEuler
>>> u, t = ForwardEuler(lambda u, t: -u**2, U0=1, T=5, n=30)
```

The reason is that this `import` statement will execute a main program in the `ForwardEuler_func.py` file, involving plotting of the solution in an example. Also, the verification tests are run, which in more complicated problems could take considerable time and thus make the `import` statement hang until the tests are done.

Go through the file and modify it such that it becomes a module where no code is executed unless the module file is run as a program. Filename: `ForwardEuler_func2.py`.

Exercise E.16: Simulate radioactive decay

The equation $u' = -au$ is a relevant model for radioactive decay, where $u(t)$ is the fraction of particles that remains in the radioactive substance at time t. The parameter a is the inverse of the so-called mean lifetime of the substance. The initial condition is $u(0) = 1$.

a) Introduce a class `Decay` to hold information about the physical problem: the parameter a and a `__call__` method for computing the right-hand side $-au$ of the ODE.

b) Initialize an instance of class `Decay` with $a = \ln(2)/5600$ 1/y. The unit 1/y means one divided by year, so time is here measured in years, and the particular value of a corresponds to the Carbon-14 radioactive isotope whose decay is used extensively in dating organic material that is tens of thousands of years old.

c) Solve $u' = -au$ with a time step of 500 y, and simulate the radioactive decay for $T = 20,000$ y. Plot the solution. Write out the final $u(T)$ value and compare it with the exact value e^{-aT}.
Filename: `radioactive_decay.py`.

Exercise E.17: Compute inverse functions by solving an ODE

The inverse function g of some function $f(x)$ takes the value of $f(x)$ back to x again: $g(f(x)) = x$. The common technique to compute inverse functions is to set $y = f(x)$ and solve with respect to x. The formula on the right-hand side is then the desired inverse function $g(y)$. Section A.1.11 makes use of such an approach, where $y - f(x) = 0$ is solved numerically with respect to x for different discrete values of y.

We can formulate a general procedure for computing inverse functions from an ODE problem. If we differentiate $y = f(x)$ with respect to y, we get $1 = f'(x)\frac{dx}{dy}$ by the chain rule. The inverse function we seek is $x(y)$, but this function then fulfills the ODE

$$x'(y) = \frac{1}{f'(x)}\,. \tag{E.68}$$

That y is the independent coordinate and x a function of y can be a somewhat confusing notation, so we might introduce u for x and t for y:

$$u'(t) = \frac{1}{f'(u)}\,.$$

The initial condition is $x(0) = x_r$ where x_r solves the equation $f(x_r) = 0$ ($x(0)$ implies $y = 0$ and then from $y = f(x)$ it follows that $f(x(0)) = 0$).

Make a program that can use the described method to compute the inverse function of $f(x)$, given x_r. Use any numerical method of your choice for solving the ODE problem. Verify the implementation for

$f(x) = 2x$. Apply the method for $f(x) = \sqrt{x}$ and plot $f(x)$ together with its inverse function. Filename: `inverse_ODE.py`.

Exercise E.18: Make a class for computing inverse functions

The method for computing inverse functions described in Exercise E.17 is very general. The purpose now is to use this general approach to make a more reusable utility, here called `Inverse`, for computing the inverse of some Python function `f(x)` on some interval `I=[a,b]`. The utility can be used as follows to calculate the inverse of $\sin x$ on $I = [0, \pi/2]$:

```
def f(x):
    return sin(x)

# Compute the inverse of f
inverse = Inverse(f, x0=0, I=[0, pi/2], resolution=100)
x, y = Inverse.compute()

plot(y, x, 'r-',
     x, f(x), 'b-',
     y, asin(y), 'go')
legend(['computed inverse', 'f(x)', 'exact inverse'])
```

Here, `x0` is the value of `x` at 0, or in general at the left point of the interval: `I[0]`. The parameter `resolution` tells how many equally sized intervals Δy we use in the numerical integration of the ODE. A default choice of 1000 can be used if it is not given by the user.

Write class `Inverse` and put it in a module. Include a test function `test_Inverse()` in the module for verifying that class `Inverse` reproduces the exact solution in the test problem $f(x) = 2x$. Filename: `Inverse1.py`.

Exercise E.19: Add functionality to a class

Extend the module in Exercise E.18 such that the value of $x(0)$ (`x0` in class `Inverse`'s constructor) does not need to be provided by the user.

Hint. Class `Inverse` can compute a value of $x(0)$ as the root of $f(x) = 0$. You may use the Bisection method from Section 4.10.2, Newton's method from Section A.1.10, or the Secant method from Exercise A.10 to solve $f(x) = 0$. Class `Inverse` should figure out a suitable initial interval for the Bisection method or start values for the Newton or Secant methods. Computing $f(x)$ for x at many points and examining these may help in solving $f(x) = 0$ without any input from the user.
Filename: `Inverse2.py`.

Exercise E.20: Compute inverse functions by interpolation

Instead of solving an ODE for computing the inverse function $g(y)$ of some function $f(x)$, as explained in Exercise E.17, one may use a simpler approach based on ideas from Section E.1.5. Say we compute discrete values of x and $f(x)$, stored in the arrays `x` and `y`. Doing a `plot(x, y)` shows $y = f(x)$ as a function of x, and doing `plot(y, x)` shows x as a function of y, i.e., we can trivially plot the inverse function $g(y)$ (!).

However, if we want the inverse function of $f(x)$ as some Python function `g(y)` that we can call for any `y`, we can use the tool `wrap2callable` from Section E.1.5 to turn the discrete inverse function, described by the arrays `y` (independent coordinate) and `x` (dependent coordinate), into a continuous function `g(y)`:

```
from scitools.std import wrap2callable
g = wrap2callable((y, x))

y = 0.5
print g(y)
```

The `g(y)` function applies linear interpolation in each interval between the points in the `y` array.

Implement this method in a program. Verify the implementation for $f(x) = 2x$, $x \in [0, 4]$, and apply the method to $f(x) = \sin x$ for $x \in [0, \pi/2]$. Filename: `inverse_wrap2callable.py`.

Exercise E.21: Code the 4th-order Runge-Kutta method; function

Use the file `ForwardEuler_func.py` from Section E.1.3 as starting point for implementing the famous and widely used 4th-order Runge-Kutta method (E.41)-(E.45). Use the test function involving a linear $u(t)$ for verifying the implementation. Exercise E.23 suggests an application of the code. Filename: `RK4_func.py`.

Exercise E.22: Code the 4th-order Runge-Kutta method; class

Carry out the steps in Exercise E.21, but base the implementation on the file `ForwardEuler.py` from Section E.1.7. Filename: `RK4_class.py`.

Exercise E.23: Compare ODE methods

Investigate the accuracy of the 4th-order Runge-Kutta method and the Forward Euler scheme for solving the (challenging) ODE problem

$$\frac{dy}{dx} = \frac{1}{2(y-1)}, \quad y(0) = 1 + \sqrt{\epsilon}, \quad x \in [0,4], \tag{E.69}$$

where ϵ is a small number, say $\epsilon = 0.001$. Start with four steps in $[0,4]$ and reduce the step size repeatedly by a factor of two until you find the solutions sufficiently accurate. Make a plot of the numerical solutions along with the exact solution $y(x) = 1 + \sqrt{x+\epsilon}$ for each step size. Filename: `yx_ODE_FE_vs_RK4.py`.

Exercise E.24: Code a test function for systems of ODEs

The `ForwardEuler_func.py` file from Section E.1.3 does not contain any test function for verifying the implementation. We can use the fact that linear functions of time will be exactly reproduced by most numerical methods for ODEs. A simple system of two ODEs with linear solutions $v(t) = 2 + 3t$ and $w(t) = 3 + 4t$ is

$$v' = 2 + (3 + 4t - w)^3, \tag{E.70}$$

$$\tag{E.71}$$

$$w' = 3 + (2 + 3t - v)^4 \tag{E.72}$$

Write a test function `test_ForwardEuler()` for comparing the numerical solution of this system with the exact solution. Filename: `ForwardEuler_sys_func2.py`.

Exercise E.25: Code Heun's method for ODE systems; function

Use the file `ForwardEuler_sys_func.py` from Section E.2.3 as starting point for implementing Heun's method (E.36)-(E.37) for systems of ODEs. Verify the solution using the test function suggested in Exercise E.24. Filename: `Heun_sys_func.py`.

Exercise E.26: Code Heun's method for ODE systems; class

Carry out the steps in Exercise E.25, but make a class implementation based on the file `ForwardEuler_sys.py` from Section E.2.4. Filename: `Heun_sys_class.py`.

Exercise E.27: Implement and test the Leapfrog method

a) Implement the Leapfrog method specified in formula (E.35) from Section E.3.1 in a subclass of `ODESolver`. Place the code in a separate module file `Leapfrog.py`.

b) Make a test function for verifying the implementation.

Hint. Use the fact that the method will exactly produce a linear u, see Section E.3.4.

c) Make a movie that shows how the Leapfrog method, the Forward Euler method, and the 4th-order Runge-Kutta method converge to the exact solution as Δt is reduced. Use the model problem $u' = u$, $u(0) = 1$, $t \in [0, 8]$, with $n = 2^k$ intervals, $k = 1, 2 \ldots, 14$. Place the movie generation in a function.

d) Repeat c) for the model problem $u' = -u$, $u(0) = 1$, $t \in [0, 5]$, with $n = 2^k$ intervals, $k = 1, 2 \ldots, 14$. In the movie, start with the finest resolution and reduce n until $n = 2$. The lessons learned is that Leapfrog can give completely wrong, oscillating solutions if the time step is not small enough.
Filename: `Leapfrog.py`.

Exercise E.28: Implement and test an Adams-Bashforth method

Do Exercise E.27 with the 3rd-order Adams-Bashforth method (E.46).
Filename: `AdamBashforth3.py`.

Exercise E.29: Solve two coupled ODEs for radioactive decay

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a mean lifetime τ_A, while substance B decay to form type A nuclei with a mean lifetime τ_B. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$u'_A = u_B/\tau_B - u_A/\tau_A, \tag{E.73}$$

$$u'_B = u_A/\tau_A - u_B/\tau_B, \tag{E.74}$$

with $u_A(0) = u_B(0) = 1$.

a) Introduce a problem class, which holds the parameters τ_A and τ_B and offers a `__call__` method to compute the right-hand side vector of the ODE system, i.e., $(u_B/\tau_B - u_A/\tau_A, u_A/\tau_A - u_B/\tau_B)$.

b) Solve for u_A and u_B using a subclass in the `ODESolver` hierarchy and the parameter choices $\tau_A = 8$ minutes, $\tau_B = 40$ minutes, and $\Delta t = 10$ seconds.

c) Plot u_A and u_B against time measured in minutes.

d) From the ODE system it follows that the ratio $u_A/u_B \rightarrow \tau_A/\tau_B$ as $t \rightarrow \infty$ (assuming $u'_A = u'_B = 0$ in the limit $t \rightarrow \infty$). Extend the problem class with a test method for checking that two given solutions u_A and u_B fulfill this requirement. Verify that this is indeed the case with the computed solutions in b).
Filename: `radioactive_decay2.py`.

Exercise E.30: Implement a 2nd-order Runge-Kutta method; function

Implement the 2nd-order Runge-Kutta method specified in formula (E.38). Use a plain function `RungeKutta2` of the type shown in Section E.1.2 for the Forward Euler method. Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Demonstrate that the numerical solution approaches the exact solution as Δt is reduced. Filename: `RungeKutta2_func.py`.

Exercise E.31: Implement a 2nd-order Runge-Kutta method; class

Make a new subclass `RungeKutta2` in the `ODESolver` hierarchy from Section E.3 for solving ordinary differential equations with the 2nd-order Runge-Kutta method specified in formula (E.38). Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution as a function of time. Place the `RungeKutta2` class and the test problem in a separate module (where the superclass `ODESolver` is imported from the `ODESolver` module). Call the test problem from the test block in the module file. How can you verify that the implementation is correct? Filename: `RungeKutta2.py`.

Exercise E.32: Code the iterated midpoint method; function

a) Implement the numerical method (E.47)-(E.48) as a function

```
iterated_Midpoint_method(f, U0, T, n, N)
```

where `f` is a Python implementation of $f(u, t)$, `U0` is the initial condition $u(0) = U_0$, `T` is the final time of the simulation, `n` is the number of time steps, and `N` is the parameter N in the method (E.47). The `iterated_Midpoint_method` should return two arrays: $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$.

Hint. You may want to build the function on the software described in Section E.1.3.

b) To verify the implementation, calculate by hand u_1 and u_2 when $N = 2$ for the ODE $u' = -2u$, $u(0) = 1$, with $\Delta t = 1/4$. Compare your hand calculations with the results of the program. Make a test function `test_iterated_Midpoint_method()` for automatically comparing the hand calculations with the output of the function in a).

c) Consider the ODE problem $u' = -2(t-4)u$, $u(0) = e^{-16}$, $t \in [0, 8]$, with exact solution $u = e^{-(t-4)^2}$. Write a function for comparing the numerical and exact solution in a plot. Enable setting of Δt and N from the command line and use the function to study the behavior of the numerical solution as you vary Δt and N. Start with $\Delta t = 0.5$ and $N = 1$. Continue with reducing Δt and increasing N.
Filename: `MidpointIter_func.py`.

Exercise E.33: Code the iterated midpoint method; class

The purpose of this exercise is to implement the numerical method (E.47)-(E.48) in a class `MidpointIter`, like the `ForwardEuler` class from Section E.1.7. Also make a test function `test_MidpointIter()` where you apply the verification technique from Exercise E.32b. Filename: `MidpointIter_class.py`.

Exercise E.34: Make a subclass for the iterated midpoint method

Implement the numerical method (E.47)-(E.48) in a subclass in the `ODESolver` hierarchy. The code should reside in a separate file where the `ODESolver` class is imported. One can either fix N or introduce an ϵ and iterate until the change in $|v_q - v_{q-1}|$ is less than ϵ. Allow the constructor to take both N and ϵ as arguments. Compute a new v_q as long as $q \leq N$ and $|v_q - v_{q-1}| > \epsilon$. Let $N = 20$ and $\epsilon = 10^{-6}$ by default. Store N as an attribute such that the user's code can access what N was in the last computation. Also write a test function for verifying the implementation. Filename: `MidpointIter.py`.

Exercise E.35: Compare the accuracy of various methods for ODEs

We want to see how various numerical methods treat the following ODE problem:

$$u' = -2(t-4)u, \quad u(0) = e^{-16}, \quad t \in (0, 10] .$$

The exact solution is a Gaussian function: $u(t) = e^{-(t-4)^2}$. Compare the Forward Euler method with other methods of your choice in the same plot. Relevant methods are the 4th-order Runge-Kutta method (found in the `ODESolver.py` hierarchy) and methods from Exercises E.5, E.21, E.22, E.25, E.26, E.27, E.28 E.31, or E.34. Put the value of Δt in the title of the plot. Perform experiments with $\Delta t = 0.3, 0.25, 0.1, 0.05, 0.01, 0.001$ and report how the various methods behave. Filename: `methods4gaussian.py`.

Exercise E.36: Animate how various methods for ODEs converge

Make a movie for illustrating how three selected numerical methods converge to the exact solution for the problem described in Exercise E.35 as Δt is reduced. Start with $\Delta t = 1$, fix the y axis in $[-0.1, 1.1]$, and reduce Δt by a quite small factor, say 1.5, between each frame in the movie. The movie must last until all methods have their curves visually on top of the exact solution. Filename: `animate_methods4gaussian.py`.

Exercise E.37: Study convergence of numerical methods for ODEs

The approximation error when solving an ODE numerically is usually of the form $C\Delta t^r$, where C and r are constants that can be estimated from numerical experiments. The constant r, called the *convergence rate*, is of particular interest. Halving Δt halves the error if $r = 1$, but if $r = 3$, halving Δt reduces the error by a factor of 8.

Exercise 9.15 describes a method for estimating r from two consecutive experiments. Make a function

```
ODE_convergence(f, U0, u_e, method, dt=[])
```

that returns a series of estimated r values corresponding to a series of Δt values given as the `dt` list. The argument `f` is a Python implementation of $f(u, t)$ in the ODE $u' = f(u, t)$. The initial condition is $u(0) = U_0$, where U_0 is given as the `U0` argument, `u_e` is the exact solution $u_e(t)$ of the ODE, and `method` is the name of a class in the `ODESolver` hierarchy. The error between the exact solution u_e and the computed solution $u_0, u_1, \ldots, u_n$ can be defined as

$$e = \left(\Delta t \sum_{i=0}^{n} (u_e(t_i) - u_i)^2 \right)^{1/2} .$$

Call the `ODE_convergence` function for some numerical methods and print the estimated r values for each method. Make your own choice of the ODE problem and the collection of numerical methods. Filename: `ODE_convergence.py`.

Exercise E.38: Find a body's position along with its velocity

In Exercise E.8 we compute the velocity $v(t)$. The position of the body, $y(t)$, is related to the velocity by $y'(t) = v(t)$. Extend the program from Exercise E.8 to solve the system

$$\begin{aligned}\frac{dy}{dt} &= v,\\ \frac{dv}{dt} &= -g\left(1-\frac{\varrho}{\varrho_b}\right) - -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,.\end{aligned}$$

Filename: `body_in_fluid2.py`.

Exercise E.39: Add the effect of air resistance on a ball

The differential equations governing the horizontal and vertical motion of a ball subject to gravity and air resistance read

$$\frac{d^2x}{dt^2} = -\frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2+\left(\frac{dy}{dt}\right)^2}\frac{dx}{dt}, \tag{E.75}$$

$$\frac{d^2y}{dt^2} = -g-\frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2+\left(\frac{dy}{dt}\right)^2}\frac{dy}{dt}, \tag{E.76}$$

where (x, y) is the position of the ball (x is a horizontal measure and y is a vertical measure), g is the acceleration of gravity, $C_D = 0.2$ is a drag coefficient, $\bar{\varrho}$ is the ratio of the density of air and the ball, and a is the radius of the ball.

Let the initial condition be $x = y = 0$ (start position in the origin) and

$$dx/dt = v_0\cos\theta, \quad dy/dt = v_0\sin\theta,$$

where v_0 is the magnitude of the initial velocity and θ is the angle the velocity makes with the horizontal.

a) Express the two second-order equations above as a system of four first-order equations with four initial conditions.

b) Implement the right-hand side in a problem class where the physical parameters C_D, $\bar{\varrho}$, a, v_0, and θ are stored along with the initial conditions.

You may also want to add a `terminate` method in this class for checking when the ball hits the ground and then terminate the solution process.

c) Simulate a hard football kick where $v_0 = 120$ km/h and θ is 30 degrees. Take the density of the ball as 0.017 hg/m^3 and the radius as 11 cm. Solve the ODE system for $C_D = 0$ (no air resistance) and $C_D = 0.2$, and plot y as a function of x in both cases to illustrate the effect of air resistance. Make sure you express all units in kg, m, s, and radians.
Filename: `kick2D.py`.

Exercise E.40: Solve an ODE system for an electric circuit

An electric circuit with a resistor, a capacitor, an inductor, and a voltage source can be described by the ODE

$$L\frac{dI}{dt} + RI + \frac{Q}{C} = E(t), \tag{E.77}$$

where LdI/dt is the voltage drop across the inductor, I is the current (measured in amperes, A), L is the inductance (measured in henrys, H), R is the resistance (measured in ohms, $\varOmega$), Q is the charge on the capacitor (measured in coulombs, C), C is the capacitance (measured in farads, F), $E(t)$ is the time-variable voltage source (measured in volts, V), and t is time (measured in seconds, s). There is a relation between I and Q:

$$\frac{dQ}{dt} = I\,. \tag{E.78}$$

Equations (E.77)-(E.78) is a system two ODEs. Solve these for $L = 1$ H, $E(t) = 2\sin\omega t$ V, $\omega^2 = 3.5$ s^{-2}, $C = 0.25$ C, $R = 0.2$ $\varOmega$, $I(0) = 1$ A, and $Q(0) = 1C$. Use the Forward Euler scheme with $\varDelta t = 2\pi/(60\omega)$. The solution will, after some time, oscillate with the same period as $E(t)$, a period of $2\pi/\omega$. Simulate 10 periods. Filename: `electric_circuit.py`.

Remarks. It turns out that the Forward Euler scheme overestimates the amplitudes of the oscillations. The more accurate 4th-order Runge-Kutta method is much better for this type of differential equation model.

Exercise E.41: Simulate the spreading of a disease by a SIR model

We shall in this exercise model epidemiological diseases such as measles or swine flu. Suppose we have three categories of people: susceptibles (S) who can get the disease, infected (I) who have developed the disease and who can infect susceptibles, and recovered (R) who have recovered from the disease and become immune. Let $S(t)$, $I(t)$, and $R(t)$ be the number of people in category S, I, and R, respectively. We have that $S + I + R = N$, where N is the size of the population, assumed constant here for simplicity.

When people mix in the population there are SI possible pairs of susceptibles and infected, and a certain fraction βSI per time interval meets with the result that the infected "successfully" infect the susceptible. During a time interval Δt, $\beta SI\Delta t$ get infected and move from the S to the I category:

$$S(t + \Delta t) = S(t) - \beta SI\Delta t\,.$$

We divide by Δt and let $\Delta \to 0$ to get the differential equation

$$S'(t) = -\beta SI\,. \qquad \text{(E.79)}$$

A fraction νI of the infected will per time unit recover from the disease. In a time Δt, $\nu I\Delta t$ recover and move from the I to the R category. The quantity $1/\nu$ typically reflects the duration of the disease. In the same time interval, $\beta SI\Delta t$ come from the S to the I category. The accounting for the I category therefore becomes

$$I(t + \Delta t) = I(t) + \beta SI\Delta t - \nu I\Delta t,$$

which in the limit $\Delta t \to \infty$ becomes the differential equation

$$I'(t) = \beta SI - \nu I\,. \qquad \text{(E.80)}$$

Finally, the R category gets contributions from the I category:

$$R(t + \Delta t) = R(t) + \nu I\Delta t\,.$$

The corresponding ODE for R reads

$$R'(t) = \nu I\,. \qquad \text{(E.81)}$$

In case the recovered do not become immune, we do not need the recovered category, since the recovered go directly out of the I category to the S category again. This gives a contribution νI to the equation for S and we end up with the S-I system (C.31)-(C.32) from Section C.5.

The system (E.79)-(E.81) is known as a SIR model in epidemiology (which is the name of the scientific field studying the spreading of epidemic diseases).

Make a function for solving the differential equations in the SIR model by any numerical method of your choice. Make a separate function for visualizing $S(t)$, $I(t)$, and $R(t)$ in the same plot.

Adding the equations shows that $S' + I' + R' = 0$, which means that $S + I + R$ must be constant. Perform a test at each time level for checking that $S + I + R$ equals $S_0 + I_0 + R_0$ within some small tolerance. If a subclass of `ODESolver` is used to solve the ODE system, the test can be implemented as a user-specified `terminate` function that is called by the `solve` method a every time level (simply return `True` for termination if $S + I + R$ is not sufficiently constant).

A specific population has 1500 susceptibles and one infected. We are interested in how the disease develops. Set $S(0) = 1500$, $I(0) = 1$, and $R(0) = 0$. Choose $\nu = 0.1$, $\Delta t = 0.5$, and $t \in [0, 60]$. Time t here counts days. Visualize first how the disease develops when $\beta = 0.0005$. Certain precautions, like staying inside, will reduce β. Try $\beta = 0.0001$ and comment from the plot how a reduction in β influences $S(t)$. (Put the comment as a multi-line string in the bottom of the program file.) Filename: `SIR.py`.

Exercise E.42: Introduce problem and solver classes in the SIR model

The parameters ν and β in the SIR model in Exercise E.41 can be constants or functions of time. Now we shall make an implementation of the $f(u, t)$ function specifying the ODE system such that ν and β can be given as either a constant or a Python function. Introduce a class for $f(u, t)$, with the following code sketch:

```
class ProblemSIR:
    def __init__(self, nu, beta, S0, I0, R0, T):
        """
        nu, beta: parameters in the ODE system
        S0, I0, R0: initial values
        T: simulation for t in [0,T]
        """
        if isinstance(nu, (float,int)):  # number?
            self.nu = lambda t: nu       # wrap as function
        elif callable(nu):
            self.nu = nu

        # same for beta and self.beta
        ...

        # store the other parameters

    def __call__(self, u, t):
        """Right-hand side function of the ODE system."""
        S, I, R = u
        return [-self.beta(t)*S*I,    # S equation
                ...,                  # I equation
                -self.nu(t)*I]        # R equation

# Example:
problem = ProblemSIR(beta=lambda t: 0.0005 if t <= 12 else 0.0001,
                     nu=0.1, S0=1500, I0=1, R0=0, T=60)
solver = ODESolver.ForwardEuler(problem)
```

Write the complete code for class `ProblemSIR` based on the sketch of ideas above. The ν parameter is usually not varying with time as $1/\nu$ is a characteristic size of the period a person is sick, but introduction of new medicine during the disease might change the picture such that time dependence becomes relevant.

We can also make a class `SolverSIR` for solving the problem (see Section E.3.6 for similar examples):

```
class SolverSIR:
    def __init__(self, problem, dt):
        self.problem, self.dt = problem, dt

    def solve(self, method=ODESolver.RungeKutta4):
        self.solver = method(self.problem)
        ic = [self.problem.S0, self.problem.I0, self.problem.R0]
        self.solver.set_initial_condition(ic)
        n = int(round(self.problem.T/float(self.dt)))
        t = np.linspace(0, self.problem.T, n+1)
        u, self.t = self.solver.solve(t)
        self.S, self.I, self.R = u[:,0], u[:,1], u[:,2]

    def plot(self):
        # plot S(t), I(t), and R(t)
```

After the breakout of a disease, authorities often start campaigns for decreasing the spreading of the disease. Suppose a massive campaign telling people to wash their hands more frequently is launched, with the effect that β is significantly reduced after a some days. For the specific case simulated in Exercise E.41, let

$$\beta(t) = \begin{cases} 0.0005, \, 0 \leq t \leq 12, \\ 0.0001, \, t > 12 \end{cases}$$

Simulate this scenario with the `Problem` and `Solver` classes. Report the maximum number of infected people and compare it to the case where $\beta(t) = 0.0005$. Filename: `SIR_class.py`.

Exercise E.43: Introduce vaccination in a SIR model

We shall now extend the SIR model in Exercise E.41 with a vaccination[2] program. If a fraction p of the susceptibles per time unit is being vaccinated, and we say that the vaccination is 100 percent effective, $pS\Delta t$ individuals will be removed from the S category in a time interval Δt. We place the vaccinated people in a new category V. The equations for S and V becomes

$$S' = -\beta SI - pS, \tag{E.82}$$
$$V' = pS\,. \tag{E.83}$$

The equations for I and R are not affected. The initial condition for V can be taken as $V(0) = 0$. The resulting model is named SIRV.

[2] `https://www.youtube.com/watch?v=s_6QW9sNPEY`

Try the same parameters as in Exercise E.41 in combination with $p = 0.1$ and compute the evolution of $S(t)$, $I(t)$, $R(t)$, and $V(t)$. Comment on the effect of vaccination on the maximum number of infected. Filename: `SIRV.py`.

Exercise E.44: Introduce a vaccination campaign in a SIR model

Let the vaccination campaign in Exercise E.43 start 6 days after the outbreak of the disease and let it last for 10 days,

$$p(t) = \begin{cases} 0.1, & 6 \leq t \leq 15, \\ 0, & \text{otherwise} \end{cases}$$

Plot the corresponding solutions $S(t)$, $I(t)$, $R(t)$, and $V(t)$. (It is clearly advantageous to have the SIRV model implemented as an extension to the classes in Exercise E.42.) Filename: `SIRV_varying_p.py`.

Exercise E.45: Find an optimal vaccination period

Let the vaccination campaign in Exercise E.44 last for V_T days:

$$p(t) = \begin{cases} 0.1, & 6 \leq t \leq 6 + V_T, \\ 0, & \text{otherwise} \end{cases}$$

Compute the maximum number of infected people, $\max_t I(t)$, as a function of $V_T \in [0, 31]$, by running the model for $V_T = 0, 1, 2 \ldots, 31$. Plot this function. Determine from the plot the optimal V_T, i.e., the smallest vaccination period V_T such that increasing V_T has negligible effect on the maximum number of infected people. Filename: `SIRV_optimal_duration.py`.

Exercise E.46: Simulate human-zombie interaction

Suppose the human population is attacked by zombies. This is quite a common happening in movies, and the "zombification" of humans acts much like the spreading of a disease. Let us make a differential equation model, inspired by the SIR model from Exercise E.41, to simulate how humans and zombies interact.

We introduce four categories of individuals:

1. S: susceptible humans who can become zombies.
2. I: infected humans, being bitten by zombies.
3. Z: zombies.
4. R: removed individuals, either conquered zombies or dead humans.

The corresponding functions counting how many individuals we have in each category are named $S(t)$, $I(t)$, $Z(t)$, and $R(t)$, respectively.

The type of zombies considered here is inspired by the standard for modern zombies set by the classic movie *The Night of the Living Dead*, by George A. Romero from 1968. Only a small extension of the SIR model is necessary to model the effect of human-zombie interaction mathematically. A fraction of the human susceptibles is getting bitten by zombies and moves to the infected category. A fraction of the infected is then turned into zombies. On the other hand, humans can conquer zombies.

Now we shall precisely set up all the dynamic features of the human-zombie populations we aim to model. Changes in the S category are due to three effects:

1. Susceptibles are infected by zombies, modeled by a term $-\Delta t\beta SZ$, similar to the S-I interaction in the SIR model.
2. Susceptibles die naturally or get killed and therefore enter the removed category. If the probability that one susceptible dies during a unit time interval is δ_S, the total expected number of deaths in a time interval Δt becomes $\Delta t\delta_S S$.
3. We also allow new humans to enter the area with zombies, as this effect may be necessary to successfully run a war on zombies. The number of new individuals in the S category arriving per time unit is denoted by Σ, giving an increase in $S(t)$ by $\Delta t\Sigma$ during a time Δt.

We could also add newborns to the S category, but we simply skip this effect since it will not be significant over time scales of a few days.

The balance of the S category is then

$$S' = \Sigma - \beta SZ - \delta_S S,$$

in the limit $\Delta t \to 0$.

The infected category gets a contribution $\Delta t\beta SZ$ from the S category, but loses individuals to the Z and R category. That is, some infected are turned into zombies, while others die. Movies reveal that infected may commit suicide or that others (susceptibles) may kill them. Let δ_I be the probability of being killed in a unit time interval. During time Δt, a total of $\delta_I\Delta tI$ will die and hence be transferred to the removed category. The probability that a single infected is turned into a zombie during a unit time interval is denoted by ρ, so that a total of $\Delta t\rho I$ individuals are lost from the I to the Z category in time Δt. The accounting in the I category becomes

$$I' = \beta SZ - \rho I - \delta_I I\,.$$

The zombie category gains $-\Delta t\rho I$ individuals from the I category. We disregard the effect that any removed individual can turn into a zombie again, as we consider that effect as pure magic beyond reasonable behavior, at least according to what is observed in the Romero movie tradition. A fundamental feature in zombie movies is that humans can conquer zombies. Here we consider zombie killing in a "man-to-man" human-zombie fight. This interaction resembles the nature of zombification (or the susceptible-infective interaction in the SIR model) and can be modeled by a loss $-\alpha SZ$ for some parameter α with an interpretation similar to that of β. The equation for Z then becomes

$$Z' = \rho I - \alpha SZ\,.$$

The accounting in the R category consists of a gain δS of natural deaths from the S category, a gain δI from the I category, and a gain αSZ from defeated zombies:

$$R' = \delta_S S + \delta_I I + \alpha SZ\,.$$

The complete SIZR model for human-zombie interaction can be summarized as

$$\begin{aligned}
S' &= \Sigma - \beta SZ - \delta_S S, && \text{(E.84)}\\
I' &= \beta SZ - \rho I - \delta_I I, && \text{(E.85)}\\
Z' &= \rho I - \alpha SZ, && \text{(E.86)}\\
R' &= \delta_S S + \delta_I I + \alpha SZ\,. && \text{(E.87)}
\end{aligned}$$

The interpretations of the parameters are as follows:

- Σ: the number of new humans brought into the zombified area per unit time.
- β: the probability that a theoretically possible human-zombie pair actually meets physically, during a unit time interval, with the result that the human is infected.
- δ_S: the probability that a susceptible human is killed or dies, in a unit time interval.
- δ_I: the probability that an infected human is killed or dies, in a unit time interval.
- ρ: the probability that an infected human is turned into a zombie, during a unit time interval.
- α: the probability that, during a unit time interval, a theoretically possible human-zombie pair fights and the human kills the zombie.

Note that probabilities per unit time do not necessarily lie in the interval $[0, 1]$. The real probability, lying between 0 and 1, arises after multiplication by the time interval of interest.

Implement the SIZR model with a `Problem` and `Solver` class as explained in Exercise E.42, allowing parameters to vary in time. The time variation is essential to make a realistic model that can mimic what happens in movies.

Test the implementation with the following data: $\beta = 0.0012$, $\alpha = 0.0016$, $\delta_I = 0.014$, $\Sigma = 2$, $\rho = 1$, $S(0) = 10$, $Z(0) = 100$, $I(0)$, $R(0) = 0$, and simulation time $T = 24$ hours. All other parameters can be set to zero. These values are estimated from the hysterical phase of the movie *The Night of the Living Dead*. The time unit is hours. Plot the S, I, Z, and R quantities. Filename: `SIZR.py`.

Exercise E.47: Simulate a zombie movie

The movie *The Night of the Living Dead* has three phases:

1. The initial phase, lasting for (say) 4 hours, where two humans meet one zombie and one of the humans get infected. A rough (and uncertain) estimation of parameters in this phase, taking into account dynamics not shown in the movie, yet necessary to establish a more realistic evolution of the S and Z categories later in the movie, is $\Sigma = 20$, $\beta = 0.03$, $\rho = 1$, $S(0) = 60$, and $Z(0) = 1$. All other parameters are taken as zero when not specified.
2. The hysterical phase, when the zombie treat is evident. This phase lasts for 24 hours, and relevant parameters can be taken as $\beta = 0.0012$, $\alpha = 0.0016$, $\delta_I = 0.014$, $\Sigma = 2$, $\rho = 1$.
3. The counter attack by humans, estimated to last for 5 hours, with parameters $\alpha = 0.006$, $\beta = 0$ (humans no longer get infected), $\delta_S = 0.0067$, $\rho = 1$.

Use the program from Exercise E.46 to simulate all three phases of the movie.

Hint. It becomes necessary to work with piecewise constant functions in time. These can be hardcoded for each special case, our one can employ a ready-made tool for such functions (actually developed in Exercise 3.26):

```
from scitools.std import PiecewiseConstant

# Define f(t) as 1.5 in [0,3], 0.1 in [3,4] and 1 in [4,7]
f = PiecewiseConstant(domain=[0, 7],
                      data=[(0, 1.5), (3, 0.1), (4, 1)])
```

Filename: `Night_of_the_Living_Dead.py`.

Exercise E.48: Simulate a war on zombies

A war on zombies can be implemented through large-scale effective attacks. A possible model is to increase α in the SIZR model from Exercise E.46 by some additional amount $\omega(t)$, where $\omega(t)$ varies in time to model strong attacks at $m+1$ distinct points of time $T_0 < T_1 < \cdots < T_m$. Around these t values we want ω to have a large value, while in between the attacks ω is small. One possible mathematical function with this behavior is a sum of Gaussian functions:

$$\omega(t) = a \sum_{i=0}^{m} \exp\left(-\frac{1}{2}\left(\frac{t-T_i}{\sigma}\right)^2\right), \tag{E.88}$$

where a measures the strength of the attacks (the maximum value of $\omega(t)$) and σ measures the length of the attacks, which should be much less than the time between the points of attack: typically, 4σ measures the length of an attack, and we must have $4\sigma \ll T_i - T_{i-1}$ for $i = 1, \ldots, m$. We should choose a significantly larger than α to make the attacks in the war on zombies much stronger than the usual "man-to-man" killing of zombies.

Modify the model and the implementation from Exercise E.46 to include a war on zombies. We start out with 50 humans and 3 zombies and $\beta = 0.03$. This leads to rapid zombification. Assume that there are some small resistances against zombies from the humans, $\alpha = 0.2\beta$, throughout the simulations. In addition, the humans implement three strong attacks, $a = 50\alpha$, at 5, 10, and 18 hours after the zombification starts. The attacks last for about 2 hours ($\sigma = 0.5$). Set $\delta_S = \Delta_I = \Sigma = 0$, $\beta = 0.03$, and $\rho = 1$, simulate for $T = 20$ hours, and see if the war on zombies modeled by the suggested $\omega(t)$ is sufficient to save mankind. Filename: `war_on_zombies.py`.

Exercise E.49: Explore predator-prey population interactions

Suppose we have two species in an environment: a predator and a prey. How will the two populations interact and change with time? A system of ordinary differential equations can give insight into this question. Let $x(t)$ and $y(t)$ be the size of the prey and the predator populations, respectively. In the absence of a predator, the population of the prey will follow the ODE derived in Section C.2:

$$\frac{dx}{dt} = rx,$$

with $r > 0$, assuming there are enough resources for exponential growth. Similarly, in the absence of prey, the predator population will just experience a death rate $m > 0$:

$$\frac{dy}{dt} = -my\,.$$

In the presence of the predator, the prey population will experience a reduction in the growth proportional to xy. The number of interactions (meetings) between x and y numbers of animals is xy, and in a certain fraction of these interactions the predator eats the prey. The predator population will correspondingly experience a growth in the population because of the xy interactions with the prey population. The adjusted growth of both populations can now be expressed as

$$\frac{dx}{dt} = rx - axy, \tag{E.89}$$

$$\frac{dy}{dt} = -my + bxy, \tag{E.90}$$

for positive constants r, m, a, and b. Solve this system and plot $x(t)$ and $y(t)$ for $r = m = 1$, $a = 0.3$, $b = 0.2$, $x(0) = 1$, and $y(0) = 1$, $t \in [0, 20]$. Try to explain the dynamics of the population growth you observe. Experiment with other values of a and b. Filename: `predator_prey.py`.

Exercise E.50: Formulate a 2nd-order ODE as a system

In this and subsequent exercises we shall deal with the following second-order ordinary differential equation with two initial conditions:

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad t > 0, \quad u(0) = U_0,\ \dot{u}(0) = V_0\,. \tag{E.91}$$

The notation $\dot{u}$ and $\ddot{u}$ means $u'(t)$ and $u''(t)$, respectively. Write (E.91) as a system of two first-order differential equations. Also set up the initial condition for this system.

Physical applications. Equation (E.91) has a wide range of applications throughout science and engineering. A primary application is damped spring systems in, e.g., cars and bicycles: u is the vertical displacement of the spring system attached to a wheel; $\dot{u}$ is then the corresponding velocity; $F(t)$ resembles a bumpy road; $s(u)$ represents the force from the spring; and $f(\dot{u})$ models the damping force (friction) in the spring system. For this particular application f and s will normally be linear functions of their arguments: $f(\dot{u}) = \beta\dot{u}$ and $s(u) = ku$, where k is a spring constant and β some parameter describing viscous damping.

Equation (E.91) can also be used to describe the motions of a moored ship or oil platform in waves: the moorings act as a nonlinear spring $s(u)$; $F(t)$ represents environmental excitation from waves, wind, and current; $f(\dot{u})$ models damping of the motion; and u is the one-dimensional displacement of the ship or platform.

Oscillations of a pendulum can be described by (E.91): u is the angle the pendulum makes with the vertical; $s(u) = (mg/L)\sin(u)$, where L is the length of the pendulum, m is the mass, and g is the acceleration of gravity; $f(\dot{u}) = \beta|\dot{u}|\dot{u}$ models air resistance (with β being some suitable constant, see Exercises 1.11 and E.54); and $F(t)$ might be some motion of the top point of the pendulum.

Another application is electric circuits with $u(t)$ as the charge, $m = L$ as the inductance, $f(\dot{u}) = R\dot{u}$ as the voltage drop across a resistor R, $s(u) = u/C$ as the voltage drop across a capacitor C, and $F(t)$ as an electromotive force (supplied by a battery or generator).

Furthermore, Equation (E.91) can act as a (very) simplified model of many other oscillating systems: aircraft wings, lasers, loudspeakers, microphones, tuning forks, guitar strings, ultrasound imaging, voice, tides, the El Ni no phenomenon, climate changes – to mention some.

We remark that (E.91) is a possibly nonlinear generalization of Equation (D.8) explained in Section D.1.3. The case in Appendix D corresponds to the special choice of $f(\dot{u})$ proportional to the velocity $\dot{u}$, $s(u)$ proportional to the displacement u, and $F(t)$ as the acceleration $\ddot{w}$ of the plate and the action of the gravity force.

Exercise E.51: Solve $\ddot{u} + u = 0$

Make a function

```
def rhs(u, t):
    ...
```

for returning a list with two elements with the two right-hand side expressions in the first-order differential equation system from Exercise E.50. As usual, the `u` argument is an array or list with the two solution components `u[0]` and `u[1]` at some time `t`. Inside `rhs`, assume that you have access to three global Python functions `friction(dudt)`, `spring(u)`, and `external(t)` for evaluating $f(\dot{u})$, $s(u)$, and $F(t)$, respectively.

Test the `rhs` function in combination with the functions $f(\dot{u}) = 0$, $F(t) = 0$, $s(u) = u$, and the choice $m = 1$. The differential equation then reads $\ddot{u} + u = 0$. With initial conditions $u(0) = 1$ and $\dot{u}(0) = 0$, one can show that the solution is given by $u(t) = \cos(t)$. Apply three numerical methods: the 4th-order Runge-Kutta method and the Forward Euler method from the `ODESolver` module developed in Section E.3, as well as the 2nd-order Runge-Kutta method developed in Exercise E.31. Use a time step $\Delta t = \pi/20$.

Plot $u(t)$ and $\dot{u}(t)$ versus t together with the exact solutions. Also make a plot of $\dot{u}$ versus u (`plot(u[:,0], u[:,1])` if `u` is the array returned from the solver's `solve` method). In the latter case, the exact plot should be a circle because the points on the curve are $(\cos t, \sin t)$, which all lie on a circle as t is varied. Observe that the ForwardEuler method results in a spiral and investigate how the spiral develops as Δt is reduced.

The kinetic energy K of the motion is given by $\frac{1}{2}m\dot{u}^2$, and the potential energy P (stored in the spring) is given by the work done by the spring force: $P = \int_0^u s(v)dv = \frac{1}{2}u^2$. Make a plot with K and P as functions of time for both the 4th-order Runge-Kutta method and the Forward Euler method, for the same physical problem described above. In this test case, the sum of the kinetic and potential energy should be constant. Compute this constant analytically and plot it together with the sum $K + P$ as calculated by the 4th-order Runge-Kutta method and the Forward Euler method. Filename: `oscillator_v1.py`.

Exercise E.52: Make a tool for analyzing oscillatory solutions

The solution $u(t)$ of the equation (E.91) often exhibits an oscillatory behavior (for the test problem in Exercise E.51 we have that $u(t) = \cos t$). It is then of interest to find the wavelength of the oscillations. The purpose of this exercise is to find and visualize the distance between peaks in a numerical representation of a continuous function.

Given an array $(y_0, \ldots, y_{n-1})$ representing a function $y(t)$ sampled at various points $t_0, \ldots, t_{n-1}$, a local maximum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} < y_k > y_{k+1}$. Similarly, a local minimum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} > y_k < y_{k+1}$. By iterating over the $y_1, \ldots, y_{n-2}$ values and making the two tests, one can collect local maxima and minima as (t_k, y_k) pairs. Make a function `minmax(t, y)` which returns two lists, `minima` and `maxima`, where each list holds pairs (2-tuples) of t and y values of local minima or maxima. Ensure that the t value increases from one pair to the next. The arguments `t` and `y` in `minmax` hold the coordinates $t_0, \ldots, t_{n-1}$ and $y_0, \ldots, y_{n-1}$, respectively.

Make another function `wavelength(peaks)` which takes a list `peaks` of 2-tuples with t and y values for local minima or maxima as argument and returns an array of distances between consecutive t values, i.e., the distances between the peaks. These distances reflect the local wavelength of the computed y function. More precisely, the first element in the returned array is `peaks[1][0]-peaks[0][0]`, the next element is `peaks[2][0]-peaks[1][0]`, and so forth.

Test the `minmax` and `wavelength` functions on y values generated by $y = e^{t/4}\cos(2t)$ and $y = e^{-t/4}\cos(t^2/5)$ for $t \in [0, 4\pi]$. Plot the $y(t)$ curve in each case, and mark the local minima and maxima computed by `minmax` with circles and boxes, respectively. Make a separate plot with the array returned from the `wavelength` function (just plot the array against its indices - the point is to see if the wavelength varies or not). Plot only the wavelengths corresponding to maxima.

Make a module with the `minmax` and `wavelength` function, and let the test block perform the tests specified above. Filename: `wavelength.py`.

Exercise E.53: Implement problem, solver, and visualizer classes

The user-chosen functions f, s, and F in Exercise E.51 must be coded with particular names. It is then difficult to have several functions for $s(u)$ and experiment with these. A much more flexible code arises if we adopt the ideas of a problem and a solver class as explained in Section E.3.6. Specifically, we shall here make use of class `Problem3` in Section E.3.6 to store information about $f(\dot{u})$, $s(u)$, $F(t)$, $u(0)$, $\dot{u}(0)$, m, T, and the exact solution (if available). The solver class can store parameters related to the numerical quality of the solution, i.e., Δt and the name of the solver class in the `ODESolver` hierarchy. In addition we will make a visualizer class for producing plots of various kinds.

We want all parameters to be set on the command line, but also have sensible default values. As in Section E.3.6, the `argparse` module is used to read data from the command line. Class `Problem` can be sketched as follows:

```
class Problem:
    def define_command_line_arguments(self, parser):
        """Add arguments to parser (argparse.ArgumentParser)."""

        parser.add_argument(
            '--friction', type=func_dudt, default='0',
            help='friction function f(dudt)',
            metavar='<function expression>')
        parser.add_argument(
            '--spring', type=func_u, default='u',
            help='spring function s(u)',
            metavar='<function expression>')
        parser.add_argument(
            '--external', type=func_t, default='0',
            help='external force function F(t)',
            metavar='<function expression>')
        parser.add_argument(
            '--u_exact', type=func_t_vec, default='0',
            help='exact solution u(t) (0 or None: now known)',
            metavar='<function expression>')
        parser.add_argument(
            '--m', type=evalcmlarg, default=1.0, help='mass',
            type=float, metavar='mass')
        ...
        return parser

    def set(self, args):
        """Initialize parameters from the command line."""
        self.friction = args.friction
        self.spring = args.spring
        self.m = args.m
        ...

    def __call__(self, u, t):
        """Define the right-hand side in the ODE system."""
        m, f, s, F = \
           self.m, self.friction, self.spring, self.external
        ...
```

Several functions are specified as the `type` argument to `parser.add_argument` for turning strings into proper objects, in particular `StringFunction` objects with different independent variables:

```
def evalcmlarg(text):
    return eval(text)

def func_dudt(text):
    return StringFunction(text, independent_variable='dudt')

def func_u(text):
    return StringFunction(text, independent_variable='u')

def func_t(text):
    return StringFunction(text, independent_variable='t')

def func_t_vec(text):
    if text == 'None' or text == '0':
        return None
    else:
        f = StringFunction(text, independent_variable='t')
        f.vectorize(globals())
        return f
```

The use of `evalcmlarg` is essential: this function runs the strings from the command line through `eval`, which means that we can use mathematical formulas like `-T '4*pi'`.

Class `Solver` is relatively much shorter than class `Problem`:

```
class Solver:
    def __init__(self, problem):
        self.problem = problem

    def define_command_line_arguments(self, parser):
        """Add arguments to parser (argparse.ArgumentParser)."""
        # add --dt and --method
        ...
        return parser

    def set(self, args):
        self.dt = args.dt
        self.n = int(round(self.problem.T/self.dt))
        self.solver = eval(args.method)

    def solve(self):
        self.solver = self.method(self.problem)
        ic = [self.problem.initial_u, self.problem.initial_dudt]
        self.solver.set_initial_condition(ic)
        time_points = linspace(0, self.problem.T, self.n+1)
        self.u, self.t = self.solver.solve(time_points)
```

The `Visualizer` class holds references to a `Problem` and `Solver` instance and creates plots. The user can specify plots in an interactive dialog in the terminal window. Inside a loop, the user is repeatedly asked to specify a plot until the user responds with `quit`. The specification of a plot can be one of the words `u`, `dudt`, `dudt-u`, `K`, and `wavelength` which means a plot of $u(t)$ versus t, $\dot{u}(t)$ versus t, $\dot{u}$ versus u, K $(= \frac{1}{2}m\dot{u}^2$, kinetic energy) versus t, and u's wavelength versus its indices, respectively. The wavelength can be computed from the local maxima of u as explained in Exercise E.52.

A sketch of class `Visualizer` is given next:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem = problem
        self.solver = solver

    def visualize(self):
        t = self.solver.t   # short form
        u, dudt = self.solver.u[:,0], self.solver.u[:,1]

        # Tag all plots with numerical and physical input values
        title = 'solver=%s, dt=%g, m=%g' % \
               (self.solver.method, self.solver.dt, self.problem.m)
        # Can easily get the formula for friction, spring and force
        # if these are string formulas.
        if isinstance(self.problem.friction, StringFunction):
            title += ' f=%s' % str(self.problem.friction)
        if isinstance(self.problem.spring, StringFunction):
            title += ' s=%s' % str(self.problem.spring)
        if isinstance(self.problem.external, StringFunction):
            title += ' F=%s' % str(self.problem.external)

        # Let the user interactively specify what
        # to be plotted
        plot_type = ''
        while plot_type != 'quit':
            plot_type = raw_input('Specify a plot: ')
            figure()
            if plot_type == 'u':
                # Plot u vs t
                if self.problem.u_exact is not None:
                    hold('on')
                    # Plot self.problem.u_exact vs t
                show()
                savefig('tmp_u.pdf')
            elif plot_type == 'dudt':
            ...
            elif plot_type == 'dudt-u':
            ...
            elif plot_type == 'K':
            ...
            elif plot_type == 'wavelength':
            ...
```

Make a complete implementation of the three proposed classes. Also make a `main` function that (i) creates a problem, solver, and visualizer, (ii) calls the functions to define command-line arguments in the problem and solver classes, (iii) reads the command line, (iv) passes on the command-line parser object to the problem and solver classes, (v) calls the solver, and (vi) calls the visualizer's `visualize` method to create plots. Collect the classes and functions in a module `oscillator`, which has a call to `main` in the test block.

The first task from Exercises E.51 can now be run as

```
Terminal
oscillator.py --method ForwardEuler --u_exact "cos(t)" \
              --dt "pi/20" --T "5*pi"
```

The other tasks from Exercises E.51 can be tested similarly.

Explore some of the possibilities of specifying several functions on the command line:

```
Terminal
oscillator.py --method RungeKutta4 --friction "0.1*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "40*pi" --m 10

oscillator.py --method RungeKutta4 --friction "0.8*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "120*pi" --m 50
```

Filename: `oscillator.py`.

Exercise E.54: Use classes for flexible choices of models

Some typical choices of $f(\dot{u})$, $s(u)$, and $F(t)$ in (E.91) are listed below:

- Linear friction force (low velocities): $f(\dot{u}) = 6\pi\mu R\dot{u}$ (Stokes drag), where R is the radius of a spherical approximation to the body's geometry, and μ is the viscosity of the surrounding fluid.
- Quadratic friction force (high velocities): $f(\dot{u}) = \frac{1}{2}C_D\varrho A|\dot{u}|\dot{u}$. See Exercise 1.11 for explanation of the symbols.
- Linear spring force: $s(u) = ku$, where k is a spring constant.
- Sinusoidal spring force: $s(u) = k\sin u$, where k is a constant.
- Cubic spring force: $s(u) = k(u - \frac{1}{6}u^3)$, where k is a spring constant.
- Sinusoidal external force: $F(t) = F_0 + A\sin\omega t$, where F_0 is the mean value of the force, A is the amplitude, and ω is the frequency.
- Bump force: $F(t) = H(t - t_1)(1 - H(t - t_2))F_0$, where $H(t)$ is the Heaviside function ($H = 0$ for $x < 0$ and $H = 1$ for $x \geq 0$), t_1 and t_2 are two given time points, and F_0 is the size of the force. This $F(t)$ is zero for $t < t_1$ and $t > t_2$, and F_0 for $t \in [t_1, t_2]$.
- Random force 1: $F(t) = F_0 + A \cdot U(t; B)$, where F_0 and A are constants, and $U(t; B)$ denotes a function whose value at time t is random and uniformly distributed in the interval $[-B, B]$.
- Random force 2: $F(t) = F_0 + A \cdot N(t; \mu, \sigma)$, where F_0 and A are constants, and $N(t; \mu, \sigma)$ denotes a function whose value at time t is random, Gaussian distributed number with mean μ and standard deviation σ.

Make a module `functions` where each of the choices above are implemented as a class with a `__call__` special method. Also add a class `Zero` for a function whose value is always zero. It is natural that the parameters in a function are set as arguments to the constructor. The different classes for spring functions can all have a common base class holding the k parameter as attribute. Filename: `functions.py`.

Exercise E.55: Apply software for oscillating systems

The purpose of this exercise is to demonstrate the use of the classes from Exercise E.54 to solve problems described by (E.91).

With a lot of models for $f(\dot{u})$, $s(u)$, and $F(t)$ available as classes in `functions.py`, the initialization of `self.friction`, `self.spring`, etc., from the command line does not work, because we assume simple string formulas on the command line. Now we want to write things like `-spring 'LinearSpring(1.0)'`. There is a quite simple remedy: replace all the special conversion functions to `StringFunction` objects by `evalcmlarg` in the `type` specifications in the `parser.add_argument` calls. If a `from functions import *` is also performed in the `oscillator.py` file, a simple `eval` will turn strings like `'LinearSpring(1.0)'` into living objects.

However, we shall here follow a simpler approach, namely dropping initializing parameters on the command line and instead set them directly in the code. Here is an example:

```
problem = Problem()
problem.m = 1.0
k = 1.2
problem.spring = CubicSpring(k)
problem.friction = Zero()
problem.T = 8*pi/sqrt(k)
...
```

This is the simplest way of making use of the objects in the `functions` module.

Note that the `set` method in classes `Solver` and `Visualizer` is unaffected by the new objects from the `functions` module, so flexible initialization via command-line arguments works as before for `-dt`, `-method`, and `plot`. One may also dare to call the `set` method in the problem object to set parameters like `m`, `initial_u`, etc., or one can choose the safer approach of not calling `set` but initialize all attributes explicitly in the user's code.

Make a new file say `oscillator_test.py` where you import class `Problem`, `Solver`, and `Visualizer`, plus all classes from the `functions` module. Provide a `main1` function for solving the following problem: $m = 1$, $u(0) = 1$, $\dot{u}(0) = 0$, no friction (use class `Zero`), no external forcing (class `Zero`), a linear spring $s(u) = u$, $\Delta t = \pi/20$, $T = 8\pi$, and exact $u(t) = \cos(t)$. Use the Forward Euler method.

Then make another function `main2` for the case with $m = 5$, $u(0) = 1$, $\dot{u}(0) = 0$, linear friction $f(\dot{u}) = 0.1\dot{u}$, $s(u) = u$, $F(t) = \sin(\frac{1}{2}t)$, $\Delta t = \pi/80$, $T = 60\pi$, and no knowledge of an exact solution. Use the 4-th order Runge-Kutta method.

Let a test block use the first command-line argument to indicate a call to `main1` or `main2`. Filename: `oscillator_test.py`.

Exercise E.56: Model the economy of fishing

A population of fish is governed by the differential equation

$$\frac{dx}{dt} = \frac{1}{10}x\left(1 - \frac{x}{100}\right) - h, \quad x(0) = 500, \tag{E.92}$$

where $x(t)$ is the size of the population at time t and h is the harvest.

a) Assume $h = 0$. Find an exact solution for $x(t)$. For which value of t is $\frac{dx}{dt}$ largest? For which value of t is $\frac{1}{x}\frac{dx}{dt}$ largest?

b) Solve the differential equation (E.92) by the Forward Euler method. Plot the numerical and exact solution in the same plot.

c) Suppose the harvest h depends on the fishers' efforts, E, in the following way: $h = qxE$, with q as a constant. Set $q = 0.1$ and assume E is constant. Show the effect of E on $x(t)$ by plotting several curves, corresponding to different E values, in the same figure.

d) The fishers' total revenue is given by $\pi = ph - \frac{c}{2}E^2$, where p is a constant. In the literature about the economy of fisheries, one is often interested in how a fishery will develop in the case the harvest is not regulated. Then new fishers will appear as long as there is money to earn ($\pi > 0$). It can (for simplicity) be reasonable to model the dependence of E on π as

$$\frac{dE}{dt} = \gamma\pi, \tag{E.93}$$

where γ is a constant. Solve the system of differential equations for $x(t)$ and $E(t)$ by the 4th-order Runge-Kutta method, and plot the curve with points $(x(t), E(t))$ in the two cases $\gamma = 1/2$ and $\gamma \to \infty$. Choose $c = 0.3$, $p = 10$, $E(0) = 0.5$, and $T = 1$.
Filename: `fishery.py`.