```
    # 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

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

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

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

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 $f(x)$.

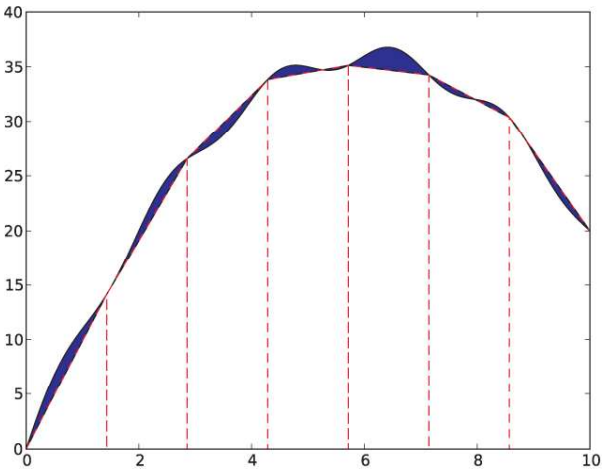
b) Use the approximation (3.9) to compute the following integrals: $\int_0^\pi \cos x \, dx$, $\int_0^\pi \sin x \, dx$, and $\int_0^{\pi/2} \sin x \, dx$, In each case, write out the error, i.e., the difference between the exact integral and the approximation (3.9). Make rough sketches of the trapezoid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

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

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).

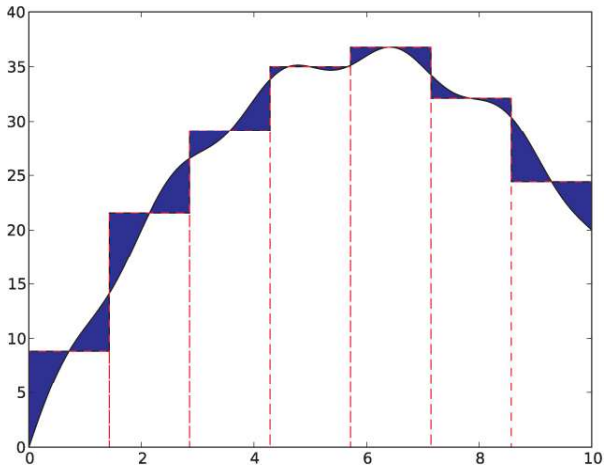
Hint. Obviously, the Trapezoidal method integrates linear functions exactly for any n. Another more surprising result is that the method is also exact for, e.g., $\int_0^{2\pi} \cos x \, dx$ for any n.
Filename: `trapezint.py`.

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

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

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`.

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

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) 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

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

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

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1, \tag{3.20}$$

with the special cases

$$1! = 1, \quad 0! = 1\,. \tag{3.21}$$

For example, $4! = 4{\cdot}3{\cdot}2{\cdot}1 = 24$, and $2! = 2{\cdot}1 = 2$. 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/`