Appendix A Solutions

Answers to selected questions are given here. For further exercises and solutions, see scipython.com.

Q2.2.5 This question illustrates the danger of “wildcard” imports: the value of the variable `e=2` is replaced by the definition of `e` in the `math` module. The expression `d ** e` therefore raises 8 to the power of $e = 2.71828\cdots$ instead of squaring it.

Q2.2.7 Using Python’s operators:

```
>>> a = 2
>>> b = 6
>>> 3 * (a**3*b - a*b**3) % 7
3
>>> a = 3
>>> b = 5
>>> 3 * (a**3*b - a*b**3) % 7
1
```

Q2.2.8 The thickness of the paper on the nth fold is $2^n t$, so we require $2^n t \geq d \Rightarrow n_{\min} = \lceil \log_2(d/t) \rceil$:

```
>>> d = 384400 * 1.e3        # distance to moon, m
>>> t = 1.e-4                # paper thickness, m
>>> math.log(d / t, 2)       # base-2 logarithm
41.805745474760016
```

Hence the paper must be folded 42 times to reach to the moon ($\lceil x \rceil$ denotes the *ceiling* of x: the smallest integer not less than x).

Q2.2.10 The `^` operator does not raise a number to another power (that is the `**` operator). It is the *bitwise xor* operator, and in binary `10^2` is `1010 xor 0010 = 1000`, which is `8` in decimal.

Q2.3.1 Slice the string `s='seehemewe'` as follows (other solutions are possible in some cases):

a. `s[:3]`
b. `s[3:5]`
c. `s[5:7]`
d. `s[7:]`
e. `s[3:6]`

f. `s[5:2:-1]`
g. `s[-2::-3]`

Q2.3.2 Simply slice the string backward and compare with the original:

```
>>> s = 'banana'
>>> s == s[::-1]
False
>>> s = 'deified'
>>> s == s[::-1]
True
```

Q2.3.5 This is not the correct way to test if the string `s` is equal to either `'ham'` or `'eggs'`. The expression `('eggs' or 'ham')` is a boolean one in which both arguments, being nonempty strings, evaluate to `True`. The expression short-circuits at the first `True` equivalent and this operand is returned (see Section 2.2.4): that is, `('eggs' or 'ham')` returns `'eggs'`. Because `s` is, indeed, the string `'eggs'` the equality comparison returns `True`. However, if the order of the operands is swapped, the boolean `or` again short-circuits at the first `True`-equivalent, which is now `'ham'` and returns it. The equality comparison with `s` fails, and the result is `False`.

There are two correct ways to test if `s` is one of two or more strings:

```
>>> s = 'eggs'
>>> s == 'ham' or s == 'eggs'
True
>>> s in ('ham', 'eggs')
True
```

(See Section 2.4.2 for more information about the syntax of the second statement.)

Q2.4.2 The problem is that `enumerate`, by default, returns the indexes and items of the array passed to it with the indexes starting at 0. The array passed to it is the slice `P[1:] = [5, 0, 2]` and so `enumerate` generates, in turn, the tuples `(0, 5)`, `(1, 0)` and `(2, 2)`. However, for our derivative we need the indexes into the original list, `P`, giving `(1, 5)`, `(2, 0)` and `(3, 2)`. There are two alternatives: pass the optional argument `start=1` to `enumerate` or add 1 to the default index:

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:], start=1):
...     dPdx.append(i*c)
>>> dPdx
[5, 0, 6]
```

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
...     dPdx.append((i+1)*c)
>>> dPdx
[5, 0, 6]
```

Q2.4.3 Here is one solution:

```
>>> scores = [87, 75, 75, 50, 32, 32]
>>> ranks = []
>>> for score in scores:
...     ranks.append(scores.index(score) + 1)
...
>>> ranks
[1, 2, 2, 4, 5, 5]
```

Q2.4.4 The following calculates π to 10 decimal places.

```
>>> import math

>>> pi = 0
>>> for k in range(20):
❶ ...     pi += pow(-3, -k) / (2*k+1)
...
>>> pi *= math.sqrt(12)
>>> print('pi = ', pi)
pi =  3.1415926535714034
>>> print('error = ', abs(pi - math.pi))
error =  1.8389734179891093e-11
```

❶ The built-in `pow(x, j)` is equivalent to `(x)**j`.

Q2.4.5 `any(x) and not all(x)` is `True` if at least one item in `x` is equivalent to `True` but not all of them:

```
>>> x1, x2, x3 = [False, False], [1, 2, 3, 4], [1, 2, 3, 0]
>>> any(x1) and not all(x1)
False
>>> any(x2) and not all(x2)
False
>>> any(x3) and not all(x3)
True
```

Q2.4.6 Recall that the `*` operator unpacks a tuple into a positional argument list to a function. So if `z = zip(a,b)` is the (iterator) sequence: `(a0,b0), (a1, b1), (a2, b2), ...`. Unpacking this sequence in the call `zip(*z)` is equivalent to calling `zip` with these tuples as arguments:

`zip((a0, b0), (a1, b1), (a2, b2), ...)`

`zip` takes the first and second items from each tuple in turn, reproducing the original sequences:

`(a0, a1, a2, ...), (b0, b1, b2, ...)`

Q2.4.7 Simply zip the lists of sunshine hours and month names together and reverse-sort the resulting list of tuples:

```
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
...           'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> sun = [44.7, 65.4, 101.7, 148.3, 170.9, 171.4,
...        176.7, 186.1, 133.9, 105.4, 59.6, 45.8]
>>> for s, m in sorted(zip(sun, months), reverse=True):
...     print('{}: {:.1f} hrs'.format(m, s))
...
```

```
Aug: 186.1 hrs
Jul: 176.7 hrs
Jun: 171.4 hrs
May: 170.9 hrs
Apr: 148.3 hrs
Sep: 133.9 hrs
Oct: 105.4 hrs
Mar: 101.7 hrs
Feb: 65.4 hrs
Nov: 59.6 hrs
Dec: 45.8 hrs
Jan: 44.7 hrs
```

Q2.5.1 To normalize a list:

```
>>> a = [2,4,10,6,8,4]
>>> amin, amax = min(a), max(a)
>>> for i, val in enumerate(a):
...     a[i] = (val-amin) / (amax-amin)
...
>>> a
[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]
```

Q2.5.2 The following code calculates Gauss's constant to 14 decimal places.

```
>>> import math
>>> tol = 1.e-14
>>> an, bn = 1., math.sqrt(2)
>>> while abs(an - bn) > tol:
...     an, bn = (an + bn) / 2, math.sqrt(an * bn)
...
>>> print('G = {:.14f}'.format(1/an))
G = 0.83462684167407
```

Q2.5.3 The following code produces the first 100 "fizzbuzz" numbers.

```
nmax = 100
for n in range(1, nmax+1):
    message = ''
    if not n % 3:
        message = 'fizz'
    if not n % 5:
        message += 'buzz'
❶   print(message or n)
```

❶ Note that if `n` is not divisible by either 3 or 5, `message` will be the empty string, which evaluates to `False` in this logical expression, so `n` is printed instead.

Q2.5.4 Here's one solution, using `stoich='C8H18'` as an example:

Listing A.1 The structural formula of a straight-chain alkane

```
# qn2-5-c-alkane-a.py

stoich = 'C8H18'

fragments = stoich.split('H')
nC = int(fragments[0][1:])
```

```
nH = int(fragments[1])
if nH != 2*nC + 2:
    print('{} is not an alkane!'.format(stoich))
else:
    print('H3C', end='')
    for i in range(nC-2):
        print('-CH2', end='')
    print('-CH3')
```

The output is:

```
H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

Q2.7.1 Only (b) and (f) behave as intended:

a. In the absence of an explicit `return` statement, the `line` function returns `None`. Because `None` cannot be joined into a string, an error occurs:

```
my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
...
TypeError: sequence item 2: expected str instance, NoneType found
```

b. This code works as intended.

c. The function `line` returns a string, as required, but is not called as `line()`: without the parentheses, `line` refers to the function object itself, which cannot be joined in a string, so an error occurs:

```
my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
...
TypeError: sequence item 2: expected str instance, function found
```

d. This code does not cause an error, but outputs a string representation of the function instead of the string returned when the function is called:

```
   56
  +44
<function line at 0x103d9e9e0>
  100
<function line at 0x103d9e9e0>
```

e. This code generates unwanted `None` output:

```
   56
  +44
-----
None
  100
-----
None
```

This happens because the statement `print(line())` calls the function `line`, which prints a line of hyphens but also prints its return value (which is `None` since it doesn't return anything else explicitly).

f. This code works as intended.

Q2.7.2 The problem is within the `add_interest` function:

```
def add_interest(balance, rate):
    balance += balance * rate / 100
```

This creates a new `float` object, `balance`, *local* to the function, which is independent of the original `balance` object. When the function exits, the local `balance` is destroyed and the original `balance` never updated. One fix would be to `return` the updated balance value from the function:

```
>>> balance = 100
>>> def add_interest(balance, rate):
...     balance += balance * rate / 100
...     return balance
...
>>> for year in range(4):
...     balance = add_interest(balance, 5)
...     print('Balance after year {}: ${:.2f}'.format(year+1, balance))
...
Balance after year 1: $105.00
Balance after year 2: $110.25
Balance after year 3: $115.76
Balance after year 4: $121.55
```

Q2.7.3 The problem is that the function `digit_sum` does not return the sum of the digits of `n` that it has calculated. In the absence of an explicit `return` statement, a Python function returns `None`, but `None` isn't an acceptable object to use in a modulus calculation and so a `TypeError` is raised.

The fix is simply to add `return dsum`:

```
def digit_sum(n):
    """ Find and return the sum of the digits of integer n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)
    return dsum

def is_harshad(n):
    return not n % digit_sum(n)
```

Now, as expected:

```
>>> is_harshad(21)
True
```

Q4.1.1 It is a good idea to keep the `try` block as small as possible to prevent exceptions that you do not want to catch being caught instead of the one you do. For example, in Example E4.5, suppose we read the file after opening it within the same `try` block:

```
try:
    fi = open(filename, 'r')
    lines = fi.readlines()
```

```
except IOError:
    ...
```

Now there are two errors that could give rise to an `IOError` Exception being raised: failure to open the file and failure to read its lines. The `except` clause is intended to handle the first case, but it will also be executed in the second case when it would be more appropriate to handle it differently (or leave it unhandled and stop program execution).

Q4.1.2 The point of `finally` in Example E4.5 is that statements in this block get executed *before* the function returns. If the line

```
print('    Done with file {}'.format(filename))
```

were moved to after the `try` block, it would not be executed if an `IOError` Exception is raised (because the function would have returned to its caller before this `print` statement is encountered.

Q4.2.1 This can easily be achieved with a `set`. Given the string, `s`:

```
set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
```

is `True` if it is a pangram. For example,

```
>>> s = 'The quick brown fox jumps over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
True
>>> s = 'The quick brown fox jumped over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
False
```

Q4.2.2 This function can be used to remove duplicates from an ordered list.

```
>>> def remove_dupes(l):
...     return sorted(set(l))
...
>>> remove_dupes([1,1,2,3,4,4,4,5,7,8,8,9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

Note that although sets don't have an order, they are iterable and can be passed to the `sorted()` built-in method (which returns a `list`).

Q4.2.3 From within the Python interpreter:

```
>>> set('hellohellohello')
{'h', 'o', 'l', 'e'}
>>> set(['hellohellohello'])
{'hellohellohello'}
>>> set(('hellohellohello'))
{'h', 'o', 'l', 'e'}
>>> set(('hellohellohello',))
{'hellohellohello'}
>>> set(('hello', 'hello', 'hello'))
{'hello'}
>>> set(('hello', ('hello', 'hello')))
{'hello', ('hello', 'hello')}
>>> set(('hello', ['hello', 'hello']))
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Note the difference between initializing a `set` with a `list` of objects and attempting to add a `list` as an object in a set.

Q4.2.4 Note that the statement

```
>>> a |= {2,3,4,5}
```

does not change the `frozenset` but rather creates a new one from the union of the old one and the `set` `{2,3,4,5}`. (In the same way, we have seen that for `int` object `i`, the assignment `i = i + 1` rebinds the label `i` to a new integer object with value `i+1` rather than changing the value of the immutable `int` object previously bound to `i`.)

Q4.3.1 The list comprehension

```
>>> flist = [lambda x, i=i: x**i for i in range(4)]
```

creates the same list of anonymous functions as that in Example E4.10.

Note that we need to pass each `i` into the `lambda` function explicitly or else Python's closure rules will lead to every `lambda` function being equivalent to `x**3` (3 being the final value of `i` in the loop).

Q4.3.2 The code snippet outputs the first `nmax+1` rows of Pascal's Triangle:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

In the list comprehension assignment,

```
x = [([0]+x)[i] + (x+[0])[i] for i in range(n+1)]
```

the elements of two lists are added. The two lists are formed from the list representing the previous row by, in the first case, adding a `0` to the beginning of the list, and in the second case, by adding a `0` to the end of the list. In this way, the sum is taken over by neighboring pairs of numbers, with the end numbers unchanged. For example, if `x` is `[1, 3, 3, 1]`, the next row is formed by summing the elements in the lists

```
[0, 1, 3, 3, 1]
[1, 3, 3, 1, 0]
```

which yields the required `[1, 4, 6, 4, 1]`.

Q4.3.3

a. Index the items of `a` using the elements of `b`:

```
>>> [a[x] for x in b]
['E', 'C', 'G', 'B', 'F', 'A', 'D']
```

b. Index the items of `a` using the sorted elements of `b`. In this case, the returned list is just (a copy of) `a`:

```
>>> [a[x] for x in sorted(b)]
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

c. Index the items of a using the elements of b indexed at the elements of b(!)

```
>>> [a[b[x]] for x in b]
['F', 'G', 'D', 'C', 'A', 'E', 'B']
```

d. Associate each element of b with the corresponding element of a in a sequence of tuples: `[(4, 'A'), (2, 'B'), (6, 'C'), ...]`, which is then sorted – this method is used to return the elements of a corresponding to the ordered elements of b.

```
>>> [x for (y,x) in sorted(zip(b,a))]
['F', 'D', 'B', 'G', 'A', 'E', 'C']
```

Q4.3.4 To return a sorted list of (*key*, *value*) pairs from a dictionary:

```
>>> d = {'five': 5, 'one': 1, 'four': 4, 'two': 2, 'three': 3}
>>> d
{'four': 4, 'one': 1, 'five': 5, 'two': 2, 'three': 3}
>>> sorted([(k, v) for k, v in d.items()])
[('five', 5), ('four', 4), ('one', 1), ('three', 3), ('two', 2)]
```

Note that sorting the list of (*key*, *value*) tuples requires that the keys all have data types that can be meaningfully ordered. This approach will not work, for example, if the keys are a mixture of integers and strings since (in Python 3) there is no defined order to sort these types into: a `TypeError: unorderable types: int() < str()` exception will be raised.

To sort by *value* we could sort a list of (*value*, *key*) tuples, but to keep the returned list as (*key*, *value*) pairs, use

```
>>> sorted([(k, v) for k, v in d.items()], key=lambda item: item[1])
[('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)]
```

The `key` argument to `sorted` specifies how to interpret each item in the list for ordering: here we want to order by the second entry `(item[1])` in each `(k, v)` tuple to order by value.

Q4.3.5 The following code encrypts (and decrypts) a telephone number held as a string using the "jump the 5" method.

```
''.join(['5987604321'[int(i)] if i.isdigit() else '-' for i in '555-867-5309'])
```

Q6.1.1 An `np.ndarray` is a NumPy class for representing multidimensional arrays in Python; in this book, we often refer to instances of this class simply as array objects. `np.array` is a function that constructs such objects from its arguments (usually a sequence).

Q6.1.2 To create a two-dimensional array, `array()` must be passed a *sequence of sequences* as a single argument: this call passes three sequence arguments instead. The correct call is

```
>>> np.array( ((1,0,0), (0,1,0), (0,0,1)) , dtype=float)
```

Q6.1.3 `np.array([0,0,0])` creates a one-dimensional array with three elements; `a = np.array([[0,0,0]])` creates a 1×3 two-dimensional array (i.e., `a[0]` is the one-dimensional array created in the first example).

Q6.1.4 Changing an array's type by setting `dtype` directly does not alter the data at the byte level, only how that data are interpreted as a number, string, and so on. As it happens, the byte-representations of zero are the same for integers (`int64`) and floats (`float64`), so the result of setting `dtype` is as expected. However, the 8-bytes representing `1.0` translate to the integer `4607182418800017408`. To convert the data type properly, use `astype()`, which returns a new array (with its own data):

```
In [x]: a = np.ones((3,))
In [x]: a
Out[x]: array([ 1.,  1.,  1.])

In [x]: a.astype('int')
In [x]: a
Out[x]: array([1, 1, 1])
```

Q6.1.5 Indexing and slicing a NumPy array:

a. `a[1,0,3]`
b. `a[0,2,:]` (or just `a[0,2]`)
c. `a[2,...]` (or `a[2,:,:]` or `a[2]`)
d. `a[:,1,:2]`
e. `a[2,:,:1:-1]` ("in the third block, for each row take the items in the middle two columns").
f. `a[:,::-1,0]` ("for each block, traverse the rows backward and take the item in the first column of each").
g. Defining the three 2×2 index arrays for the blocks, rows and columns locating our elements as follows:

```
ia = np.array([[0, 0], [2, 2]])
ja = np.array([[0, 0], [3, 3]])
ka = np.array([[0, 3], [0, 3]])
```

`a[ia,ja,ka]` returns the desired result.

Q6.1.6 For example,

```
In [a]: a = np.array([0, -1, 4.5, 0.5, -0.2, 1.1])
In [x]: a[abs(a)<=1]
Out[x]: array([ 0. , -1. ,  0.5, -0.2])
```

Q6.1.7 In the following code:

```
In [x]: a, b = -2.00231930436153, -2.0023193043615
In [x]: np.isclose(a, b, atol=1.e-14)
Out[x]: True
```

`np.isclose()` returns `True` because although the absolute difference between the two numbers is greater than 10^{-14}, it is (significantly) less than `rtol * abs(b)`, the contribution from the default *relative* difference. To obtain the expected behavior, set `rtol` to 0:

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14, rtol=0)
Out[x]: False
```

Q6.1.8 The different behavior here is due to the finite precision with which real numbers are stored: double-precision floating point numbers are only represented to the equivalent of about 15 decimal places and so the two numbers being compared here are the same to within this precision:

```
In [x]: 3.1415926535897932 - 3.141592653589793
Out[x]: 0.0
```

Q6.1.9 For example,

```
In [x]: N = 5
In [x]: Nsq = N**2
In [x]: np.allclose(np.sort(magic_square.flatten()),
                    np.linspace(1, Nsq, Nsq).astype(int))
Out[x]: True

In [x]: Nsum = N * (N**2 + 1) // 2
In [x]: np.allclose(np.sum(magic_square, axis=0), Nsum)
Out[x]: True

In [x]: np.allclose(np.sum(magic_square, axis=1), Nsum)
Out[x]: True

In [x]: n.allclose(np.diag(magic_square), Nsum)
Out[x]: True

❶ In [x]: n.allclose(np.diag(np.fliplr(magic_square)), Nsum)
Out[x]: True
```

❶ `np.fliplr` flips the array in the left/right direction. An alternative way to get this “other” diagonal is with `a.ravel()[N-1:-N+1:N-1]`.

Q6.1.10 The following statement will determine if a sequence `a` is increasing or not:

```
np.all(np.diff(a) > 0)
```

Q6.1.11 In the first case, a single object is created of the requested `dtype` and multiplied by a scalar (regular Python `int`). Python “upcasts” to return the result in `dtype` that can hold it:

```
In [x]: x = np.uint8(250)
In [x]: type(x*2)
Out[x]: numpy.int64
```

However, a `ndarray`, because it has a fixed byte size, cannot be upcast in the same way: its own `dtype` takes precedence over that of the scalar multiplying it, and so the multiplication is carried out modulo 256.

Compare this with the result of multiplying two scalars with the same `dtype`:

```
In [x]: np.uint8(250) * np.uint8(2)
Out[x]: 244             # (of type np.uint8)
```

(You may also see a warning: RuntimeWarning: overflow encountered in ubyte_scalars.)

Q6.4.1 The Polynomial deriv method returns a Polynomial object (in this case with a single term, the coefficient of x^0, equal to 18). This object is not equal to the integer object with value 18.

Q6.4.2 Using numpy.polynomial.Polynomial,

```
In [x]: p1 = Polynomial([-11,1,1])
In [x]: p2 = Polynomial([-7,1,1])
In [x]: p = p1**2 + p2**2
In [x]: dp = p.deriv()      # first derivative
In [x]: stationary_points = dp.roots()
In [x]: ddp = dp.deriv()    # second derivative
In [x]: minima = stationary_points[ddp(stationary_points) > 0]
In [x]: maxima = stationary_points[ddp(stationary_points) < 0]
In [x]: inflections = stationary_points[np.isclose(ddp(stationary_points),0)]
In [x]: print(np.array((minima, p(minima))).T)
[[-3.54138127  8.        ]
 [ 2.54138127  8.        ]]
In [x]: print(np.array((maxima, p(maxima))).T)
[[  -0.5   ,  179.125]]
In [x]: print(np.array((inflections, p(inflections))).T)
[]
```

That is, the function has two minima,

$$f(-3.54138127) = 8$$
$$f(2.54138127) = 8$$

one maximum,

$$f(-0.5) = 179.125$$

and no points of inflection / undulation.

Q6.5.1 Without overcomplicating things,

```
In [x]: pauli_matrices = np.array((
                                   ((0, 1), (1, 0)),
                                   ((0, -1j), (1j, 0)),
                                   ((1, 0), (0, -1))
                                  ))
In [x]: I2 = np.eye(2)
In [x]: for sigma in pauli_matrices:
   ...:     print(np.allclose(sigma.T.conj().dot(sigma), I2))
True
True
True
```

Q6.5.2 The following code fits the coefficients to the required quadratic equation. Note that this is a *linear* least squares fit even though the function is nonlinear in time because it is linear with respect to the coefficients.

```
# qn6-9-b-quadratic-fit-a.py
import numpy as np
import pylab
Polynomial = np.polynomial.Polynomial

x = np.array([1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2,
              384.7, 474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2,
              1369.4, 1535.1, 1710.3, 1894.9])
dt, n = 0.1, len(x)
tmax = dt * (n-1)
t = np.linspace(0, tmax, n)

A = np.vstack((np.ones(n), t, t**2)).T
coefs, resid, _, _ = np.linalg.lstsq(A, x)

# Initial position (cm) and speed (cm.s-1), acceleration due to gravity (m.s-2)
x0, v0, g = coefs[0], coefs[1], coefs[2] * 2 / 100

print('x0 = {:.2f} cm, v0 = {:.2f} cm.s-1, g = {:.2f} m.s-2'.format(x0, v0, g))

xfit = Polynomial(coefs)(t)
pylab.plot(t, x, 'ko')
pylab.plot(t, xfit, 'r')
pylab.xlabel('Time (sec)')
pylab.ylabel('Distance (cm)')
pylab.show()
```

The fitted function is shown in Figure A.1.

Q6.7.1 The first case,

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])      # (for example)
```

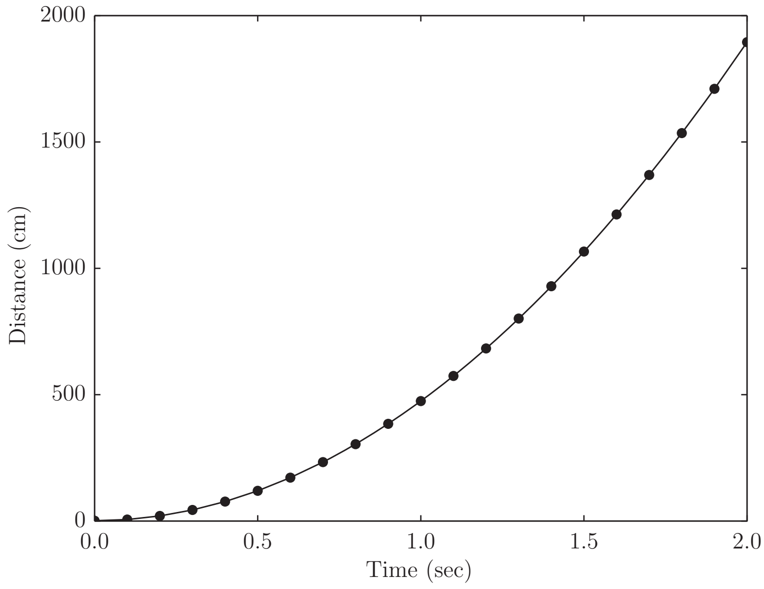

Figure A.1 Least squares fit to the function $x = x_0 + v_0 t + \frac{1}{2} g t^2$.

Samples randomly from the array `a` with replacement: for each item selected the probability of a 6 is $\frac{1}{3}$ and the probability of a 7 is $\frac{2}{3}$.

In the second case,

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])       # (for example)
```

the numbers are drawn from $[6, 7]$ uniformly, so the probabilities of each number being selected is $\frac{1}{2}$.

Q6.7.2 The function `np.random.randint` samples uniformly from the half-open interval, `[low, high)`, so to get the equivalent behavior to `np.random.random_integers` in Example E6.16 we need:

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b-a) * (np.random.randint(1, n+1, size=10) - 1) / (n-1)
Out[x]: array([ 0.5,  1.5,  0.5,  3.5,  1.5,  3.5,  2.5,  0.5,  1.5,  1.5])
```

Q6.7.3 The probability of winning is one in

$$\binom{75}{5}\binom{15}{1} = \frac{75\cdot 74\cdot 73\cdot 72\cdot 71}{1\cdot 2\cdot 3\cdot 4\cdot 5}\cdot 15 = 258890850$$

To pick five random numbers from 1–75 and one from 1–15:

```
In [x]: (sorted(np.random.choice(np.arange(1,76), 5, replace=False)),
         np.random.randint(15)+1)
([4, 21, 35, 36, 64], 14)
```

Q6.7.4 Here is a more general solution to the problem. Draw the distribution of misprints across the book from the binomial distribution using `np.random.binomial` and count up how many pages have more than q misprints on them. To compare with the Poisson distribution, for the number of misprints on a page, X, we must calculate $\mathrm{Pr}(X >= q) = 1 - \mathrm{Pr}(X < q) = 1 - (\mathrm{Pr}(X = 0) + \mathrm{Pr}(X = 1) + \cdots + \mathrm{Pr}(X = q - 1)$:

Listing A.2 Calculating the probability of q or more misprints on a given page of a book.

```
# qn6-7-d-misprints-a.py
import numpy as np

n, m = 500, 400
q = 2
ntrials = 100
errors_per_page = np.random.binomial(m, 1/n, (ntrials, n))
av_ge_q = np.sum(errors_per_page>=q) / n / ntrials
print('Probability of {} or more misprints on a given page'.format(q))
print('Result from {} trials using binomial distribution: {:.6f}'
            .format(ntrials, av_ge_q))

# Now calculate the same quantity using the Poisson approximation,
# Pr(X>=q) = 1 - exp(-lam)[1 + lam + lam^2/2! + ... + lam^(q-1}/(q-1)!]
lam = m/n
poisson = 1
term = 1
for k in range(1,q):
    term *= lam/k
    poisson += term
poisson = 1 - np.exp(-lam) * poisson
print('Result from Poisson distribution: {:.6f}'.format(poisson))
```

A sample output is

```
Probability of 2 or more misprints on a given page
Result from 100 trials using binomial distribution: 0.190200
Result from Poisson distribution: 0.191208
```

Q6.8.1 The two methods for calculating the DFT can be timed using the IPython `%timeit` magic function

```
In [x]: import numpy as np
In [x]: n = 512
In [x]: # Our input function is just random numbers
In [x]: f = np.random.rand(n)

In [x]: # Time the NumPy (Cooley-Tukey) DFT algorithm
In [x]: %timeit np.fft.fft(f)
100000 loops, best of 3: 13.1 us per loop

In [x]: # Now calculate the DFT by direct summation
In [x]: k = np.arange(n)
In [x]: m = k.reshape((n, 1))
In [x]: w = np.exp(-2j * np.pi * m * k / n)
In [x]: %timeit np.dot(w, f)
1000 loops, best of 3: 354 us per loop

In [x]: # Check the two methods produce the same result
In [x]: ftfast = np.fft.fft(f)
In [x]: ftslow = np.dot(w, f)
In [x]: np.allclose(ftfast, ftslow)
Out[x]: True
```

The Cooley-Tukey algorithm is found to be almost 30 times faster than the direct method. In fact, this algorithm can be shown to scale as $\mathcal{O}(n \log n)$ compared with $\mathcal{O}(n^2)$ for direct summation.

Q8.1.1 Simply change the line:

```
for rec in constants[-10:]:
```

to:

```
for rec in constants[constants['rel_unc'] > 0][:10]:
```

The most accurately known constant is the electron g-factor.

```
2.64693e-07 ppm: electron g factor = -2.00232
2.69687e-07 ppm: electron mag. mom. to Bohr magneton ratio = -1.00116
3.7956e-06 ppm: electron magn. moment to Bohr magneton ratio = -1.00116
4.96096e-06 ppm: atomic unit of time = 2.41888e-17 s
...
```

Q8.1.2 The calculation $N/V = p/k_{\mathrm{B}}T$ for the stated conditions can be done entirely with constants from `scipy.constants`:

```
In [x]: scipy.constants.atm / scipy.constants.k / scipy.constants.zero_Celsius
Out[x]: 2.686780501003883e+25
```

This is the *Loschmidt constant* which is defined by the 2010 CODATA standards and included in `scipy.constants` (see the documentation for details):

```
In [x]: from scipy import constants
In [x]: constants.value('Loschmidt constant (273.15 K, 101.325 kPa)')
Out[x]: 2.6867805e+25
```

Q8.2.1 By numerical integration, the result is seen to be 3:

```
In [x]: from scipy.integrate import quad
In [x]: import numpy as np
In [x]: func = lambda x: np.floor(x) - 2*np.floor(x/2)
In [x]: quad(func,0,6)
Out[x]: (2.999964948683555,0.0009520766614606472)
```

Q8.2.2 In the following we assume the following imports:

```
In [x]: import numpy as np
In [x]: from scipy.integrate import quad
```

a.

```
In [x]: f1 = lambda x: x**4 * (1 - x)**4/(1 + x**2)

In [x]: quad(f1, 0, 1)
Out[x]: (0.0012644892673496185, 1.1126990906558069e-14)

In [x]: 22/7 - np.pi
Out[x]: 0.0012644892673496777
```

b.

```
In [x]: f2 = lambda x: x**3/(np.exp(x) - 1)

In [x]: quad(f2, 0, np.inf)
Out[x]: (6.49393940226683, 2.628470028924825e-09)

In [x]: np.pi**4 / 15
Out[x]: 6.493939402266828
```

c.

```
In [x]: f3 = lambda x: x**-x

In [x]: quad(f3, 0, 1)
Out[x]: (1.2912859970626633, 3.668398917966442e-11)

In [x]: np.sum(n**-n for n in range(1,20))
Out[x]: 1.2912859970626636
```

d.

```
In [x]: from scipy.misc import factorial
In [x]: f4 = lambda x, p: np.log(1/x)**p
In [x]: for p in range(10):
   ...:     print(quad(f4, 0, 1, args=(p,))[0], factorial(p))
   ...:
1.0 1.0
0.9999999999999999 1.0
1.9999999999999991 2.0
6.000000000000064 6.0
```

```
24.000000000000014 24.0
119.9999999999327 120.0
719.9999999989705 720.0
5039.99999945767 5040.0
40320.00000363255 40320.0
362880.00027390465 362880.0
```

e.
```
In [x]: from scipy.special import i0
In [x]: z = np.linspace(0,2,100)
In [x]: y1 = i0(z)
In [x]: f5 = lambda theta, z: np.exp(z*np.cos(theta))
In [x]: y2 = np.array([quad(f5, 0, 2*np.pi, args=(zz,))[0] for zz in z])
In [x]: y2 /= 2 * np.pi
In [x]: np.max(abs(y2-y1))
Out[x]: 3.4796610037801656e-12
```

Q8.2.3 To estimate π by integration of the constant function $f(x,y) = 4$ over the quarter circle with unit radius in the quadrant $x > 0, y > 0$:

```
In [x]: from scipy.integrate import dblquad
In [x]: dblquad(lambda y, x: 4, 0, 1, lambda x: 0, lambda x: np.sqrt(1-x**2))
Out[x]: (3.1415926535897922, 3.533564552071766e-10)
```

Q8.2.4 The integral to be calculated is

$$\int_0^1 \int_0^{2\pi} r\,\mathrm{d}\theta\,\mathrm{d}r = \pi.$$

Note that the inner integral is over θ and the outer is over r. Therefore, the call to `dblquad` should call the function $f(r,\theta) = r$ as `lambda theta, r: r` (note the order of the arguments).

```
In [x]: dblquad(lambda theta, r: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (3.141592653589793, 3.487868498008632e-14)
```

Alternatively, swap the order of the integration:

```
dblquad(lambda r, theta: r, 0, 2*np.pi, lambda theta: 0, lambda theta: 1)
(3.141592653589793, 3.487868498008632e-14)
```

Q8.4.1 Rewrite the equation as

$$f(x) = x + 1 + (x-3)^{-3} = 0.$$

This function is readily plotted and the roots may be bracketed in $(-2, -0.5)$ and $(0, 2.99)$ (avoiding the singularity at $x = 3$).

```
In [x]: f = lambda x: x + 1 + (x-3)**-3
In [x]: brentq(f, -2, -0.5), brentq(f, 0, 2.99)
Out[x]: (-0.984188231211512, 2.3303684533047426)
```

Q8.4.2 Some examples of root-finding for which the Newton-Raphson algorithm fails and how to solve this.

a.
```
In [x]: newton(lambda x: x**3 - 5*x, 1, lambda x: 3*x**2 - 5)
...
RuntimeError: Failed to converge after 50 iterations, value is 1.0
```

The Newton-Raphson algorithm enters an endless cycle of values for x:

$$\begin{aligned} x_0 = 1: &\quad x_1 = x_0 - f(x_0)/f'(x_0) = -1 \\ x_1 = -1: &\quad x_2 = x_1 - f(x_1)/f'(x_1) = 1 \\ x_2 = 1: &\quad x_3 = x_2 - f(x_2)/f'(x_2) = -1 \\ \ldots & \end{aligned}$$

Alternative starting points converge correctly on a root. Even a very small displacement from $x = 0$ ensures convergence:

```
In [x]: newton(lambda x: x**3 - 5*x, 1.0001, lambda x: 3*x**2 - 5)
Out[x]: 2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 1.1, lambda x: 3*x**2 - 5)
Out[x]: -2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 0.5, lambda x: 3*x**2 - 5)
Out[x]: 0.0
```

b.
```
In [x]: f, fp = lambda x: x**3 - 3*x+1, lambda x: 3*x**2 - 3
In [x]: newton(f, 1, fp)
Out[x]: 1.0
In [x]: f(1.0)
Out[x]: -1
```

The algorithm converged, but not on a root! Unfortunately, the gradient of the function is zero at the chosen starting point and because of round-off error this has not led to a `ZeroDivisionError`. To find the roots, choose different starting points such that $f'(x_0) \neq 0$, or use a different method after bracketing the roots by inspection of a plot of the function:

```
In [x]: brentq(f, -0.5, 0.5), brentq(f, -2, -1.5), brentq(f, 1, 2)
Out[x]: (0.34729635533386066, -1.879385241571423, 1.532088886237956)
```

c. The function $f(x) = 2 - x^5$ has a flat plateau around $f(0) = 2$ and the small gradient here leads to slow convergence on the root:

```
In [x]: newton(f, 0.01, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

To find it using `newton` either move the starting point closer to the root, or increase the maximum number of iterations:

```
In [x]: newton(f, 0.01, fp, maxiter=100)
Out[x]: 1.148698354997035
```

d. This is another example of a function that generates an endless cycle of values from the Newton-Raphson method:

```
In [x]: f = lambda x: x**4 - 4.29 * x**2 - 5.29
In [x]: fp = lambda x: 4*x**3 - 8.58 * x
In [x]: newton(f, 0.8, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

Unlike the function in (a), the region $0.6 \leq x_0 \leq 1.1$ *attracts* this cyclic behavior, so one needs to initialize the algorithm outside this range to obtain the roots ± 2.3. For example,

```
In [x]: newton(f, 1.2, fp)
Out[x]: -2.3
```

Q8.4.3 In general, there are two (physically distinct) possible angles θ_0 corresponding to the projectile passing through the specified point, $(x_1, y_1) = (5, 15)$, on the way up or on the way down. These values are the roots in $(0, \pi/2)$ of the function

$$f(\theta_0; x_1, z_1) = x_1 \tan\theta_0 - \frac{gx_1^2}{2v_0^2\cos^2\theta_0} - z_1$$

After bracketing the roots with a rough plot of $f(\theta_0)$, we can use `brentq`:

```
In [x]: g = 9.81
In [x]: v0, x1, z1 = 25, 5, 15
In [x]: f = lambda theta0, x1, z1: x1 * np.tan(theta0) - g / 2\
                                  * (x1 / v0 / np.cos(theta0))**2 - z1
In [x]: th1 = brentq(f, 1, 1.4, args=(x1,z1))
In [x]: th2 = brentq(f, 1.5, 1.6, args=(x1,z1))
In [x]: np.degrees(th1), np.degrees(th2)
Out[x]: (74.172740936822834, 87.392310240255171)
```

That is, $\theta_0 = 74.2°$ or $\theta_0 = 87.4°$.

Q9.1.1 Let $x = 0.9999\cdots$. Then,

$$10x = 9.9999\cdots = 9 + x \ \Rightarrow\ 9x = 9 \ \Rightarrow\ x = 1.$$

Q9.1.2 This occurs because `math.pi` is only a (double-precision floating point) approximation to π, and the tangent of this approximate value happens to be negative:

```
In [x]: math.tan(math.pi)
Out[x]: -1.2246467991473532e-16
```

Taking the square root leads to the math domain error.

Q9.1.3 The problem, of course, is that the expression has been written using double-precision floating point numbers and the difference between the sum of the first two terms and the third is smaller than the precision of this representation. Using the exact representation in integer arithmetic,

```
In [x]: 844487**5 + 1288439**5
Out[x]: 3980245235185639013055619497406
In [x]: 1288439**5
Out[x]: 3980245235185639013290924656032
```

giving a difference of

```
In [x]: 844487**5 + 1288439**5 - 1318202**5
Out[x]: -235305158626
```

The finite precision of the floating point representation used, however, truncates the decimal places before this difference is apparent:

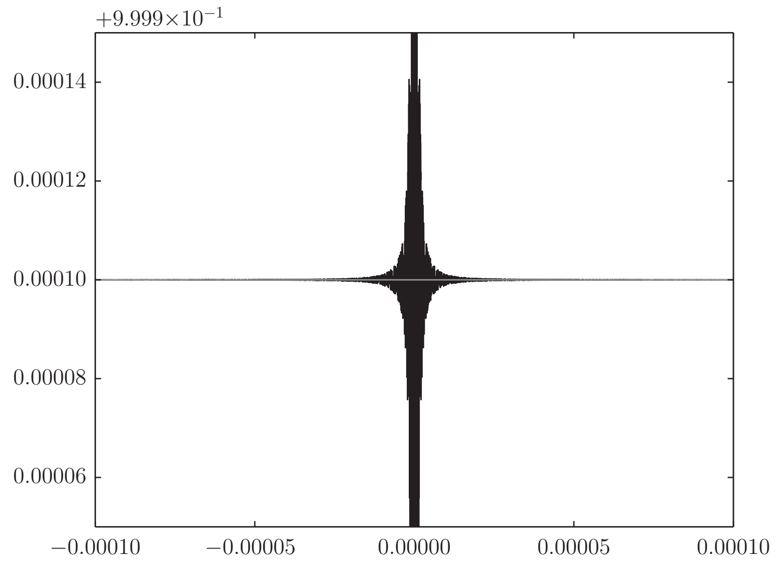

Figure A.2 A comparison of the numerical behavior of $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ close to $x = 0$.

```
In [x]: 844487.**5 + 1288439.**5
Out[x]: 3.980245235185639e+30
In [x]: 1318202.**5
Out[x]: 3.980245235185639e+30
```

This is an example of *catastrophic cancellation*.

Q9.1.4 The expression `1 - np.cos(x)**2` suffers from catastrophic cancellation close to `x=0` resulting in a dramatic loss of precision and wild oscillations in the plot of $f(x)$ (Figure A.2). Consider, for example, `x = 1.e-9`: in this case, the *difference* `1 - np.cos(x)**2` is indistinguishable from zero (at double precision) so `f(x)` returns 0. Conversely, `np.sin(x)**2` is indistinguishable from `x**2` and `g(x)` returns `1.0` correctly.

Listing A.3 A comparison of the numerical behavior of $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ close to $x = 0$.

```
# qn9-1-c-cos-sin-a.py

import numpy as np
import pylab

f = lambda x: (1 - np.cos(x)**2)/x**2
g = lambda x: (np.sin(x)/x)**2

x = np.linspace(-0.0001, 0.0001, 10000)

pylab.plot(x, f(x))
pylab.plot(x, g(x))
pylab.ylim(0.99995, 1.00005)
pylab.show()
```

Q9.1.5 We cannot compare with `==` because `nan` is not equal to itself. However, it is the *only* floating point number that is not equal to itself, so use `!=` instead:

```
In [x]: c = 0 * 1.e1000     # 0 * inf is nan
In [x]: c != c
Out[x]: True                # c isn't equal to itself, so must be nan
```