Answers to Questions

Hints may be found on page 221.

Chapter 1 (Starting Off)

1

The expression `17` is a number and so is a value already – there is no work to do. The expression `1 + 2 * 3 + 4` will evaluate to the value `11`, since multiplication has higher precedence than addition. The expression `400 > 200` evaluates to the boolean `True` since this is result of the comparison operator `>` on the operands `400` and `200`. Similarly, `1 != 1` evaluates to `False`. The expression `True or False` evaluates to `True` since one of the operands is true. Similarly, `True and False` evaluates to `False` since one of the operands is false. The expression `'%'` is a string and is already a value.

2

The expression evaluates to `11`. The programmer seems to be under the impression that spacing affects precedence. It does not, and so this use of space is misleading.

3

The `%` operator is of higher precedence than the `+` operator. So `1 + 2 % 3` and `1 + (2 % 3)` are the same expression, evaluating to `1 + 2` which is `3`, but `(1 + 2) % 3` is the same as `3 % 3`, which is `0`.

4

The comparison operator `<` considers the words in dictionary order, so `'bacon' < 'eggs'`. The uppercase letters are all "smaller" than the lowercase characters, so for example `'Bacon' < 'Bacon'` evaluates to `True`. For booleans, `False` is considered "less than" `True`.

5

The first one is, of course entirely as expected:

```
Python
>>> 1 + 2
3
```

It turns out that the + operator we have been using on numbers to add them can be used on strings to concatenate them:

```
Python
>>> 'one' + 'two'
'onetwo'
```

However, it will not work to mix the two types:

```
Python
>>> 1 + 'two'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The $*$ operator can also be used on a string and a number, to concatenate the string multiple times:

```
Python
>>> 3 * '1'
'111'
>>> '1' * 3
'111'
>>> print('1' * 3)
111
```

In the last line, we remember the difference between a string and printing a string. When it is (ab)used as a number, `True` has the value 1, whereas `False` has the value 0:

```
Python
>>> True + 1
2
>>> False + 1
1
```

Did you notice the `f` before the quotation mark in the last example? This is a *format string*, which we will discuss in chapter 6:

```
Python
>>> print(f'One and two is {1 + 2} and that is all.')
One and two is 3 and that is all.
```

The part between the curly braces `{}` has been evaluated and then printed.

Chapter 2 (Names and Functions)

1

We include the **`return`** keyword to make sure the result is returned to us:

```
Python
>>> def times_ten(x):
...     return x * 10
...
>>> times_ten(50)
500
```

2

This function will have two arguments. We use the **`and`** operator, together with the inequality operator `!=`.

```
Python
>>> def both_non_zero(a, b):
...     return a != 0 and b != 0
...
>>> both_non_zero(1, 2)
True
>>> both_non_zero(1, 0)
False
```

3

This is a simple function with three arguments. We remember to use **`return`**, of course:

```
Python
>>> def volume(w, h, d):
...    return w * h * d
...
>>> volume(10, 20, 30)
6000
```

We can now write our `volume_ten_deep` function:

```
Python
>>> def volume_ten_deep(w, h):
...     return volume(w, h, 10)
>>> volume_ten_deep(5, 6)
300
```

Notice that we need **`return`** here too: the **`return`** in `volume` will not suffice.

4

If a lower case character in the range `'a'`…`'z'` is not a vowel, it must be a consonant. So we can reuse the `is_vowel` function we wrote earlier, and negate its result using **`not`**:

```
Python
>>> def is_consonant(s):
...     return not is_vowel(s)
...
>>> is_consonant('r')
True
>>> is_consonant('e')
False
```

5

We could simply return 0 for a negative argument. The factorial of 0 is 1, so we can change that too, and say our new function finds the factorial of any non-negative number:

```
Python
>>> def factorial(x):
...     if x < 0:
...         return 0
...     elif x == 0:
...         return 1
...     else:
...         return x * factorial(x - 1)
...
>>> factorial(-1)
0
```

6

We can use a recursive function:

```
Python
>>> def sum_nums(n):
...     if n == 1:
...         return 1
...     else:
...         return n + sum_nums(n - 1)
...
>>> sum_nums(10)
55
```

There is a direct mathematical formula too. We use the integer division operator `//`, which we have not yet seen:

```
Python
>>> def sum_nums(n):
...     return (n * (n + 1)) // 2
>>> sum_nums(10)
55
```

Can you see why?

7

A number to the power of 0 is 1. A number to the power of 1 is itself. Otherwise, the answer is the current n multiplied by n^{x-1}.

```
Python
>>> def power(x, n):
...     if n == 0:
...         return 1
...     else:
...         if n == 1:
...             return x
...         else:
...             return x * power(x, n - 1)
...
>>> power(2, 5)
32
```

Notice that we had to put one **`if`** and **`else`** inside another here. The indentation helps to show the structure. Remembering that Python allows us to compress this using the **`elif`** keyword:

```
Python
>>> def power(x, n):
...     if n == 0:
...         return 1
...     elif n == 1:
...         return x
...     else:
...         return x * power(x, n - 1)
...
>>> power(2, 5)
32
```

This is easier to read, partly because all the **`return`** keywords line up. In fact, we can remove the case for `n = 1` since `power(x, 1)` will reduce to `x * power(x, 0)` which is just `x`.

8

We test each number less than the given number for divisibility using the `%` modulus operator we learned about in chapter 1, increasing the test divisor by one each time. If it is divisible, we print it.

```
Python
>>> def factors(n, x):
...  if x % n == 0: print(n)
...  if n < x: factors(n + 1, x)
>>> factors(1, 12)
1
2
3
4
6
12
```

We can clean the solution up by wrapping it in another function which supplies the starting point of `1`:

```
Python
>>> def factors_simple(x):
...   factors(1, x)
>>> factors_simple(12)
1
2
3
4
6
12
```

Chapter 3 (Again and Again)

1

We set the step to -1. We must be careful with the start and stop points. We set the stop point to 0 so that we stop at 1 (i.e. before we get to 0):

```
def print_down_from(n):
    for x in range(n, 0, -1):
        print(x)
```

2

We change the calculation of `column_width` to take `n * (n - 1)` as the item with maximum width rather than `n * n`:

```
def times_table(n):
    column_width = len(str(n * (n - 1))) + 1
    for y in range(1, n + 1):
        for x in range(1, n + 1):
            print(x * y, end=' ' * (column_width - len(str(x * y))))
        print('')
```

Here is the new result for a table of size ten:

```
Python
>>> times_table(10)
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10 12 14 16 18 20
3  6  9  12 15 18 21 24 27 30
4  8  12 16 20 24 28 32 36 40
5  10 15 20 25 30 35 40 45 50
6  12 18 24 30 36 42 48 54 60
7  14 21 28 35 42 49 56 63 70
8  16 24 32 40 48 56 64 72 80
9  18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

It is still possible to have excess space in some columns with this method:

```
Python
>>> times_table(4)
1  2  3  4
2  4  6  8
3  6  9  12
4  8  12 16
```

Can you fix this?

3

We use a **for** loop to check each letter in the string, adding one to a local variable each time we see a space:

```
def count_spaces(s):
    c = 0
    for x in s:
        if x == ' ':
            c = c + 1
    return c
```

We use **return** to make sure the final count is the result of the function. We can use the **+=** operator to shorten the common operation of adding to a variable:

```
def count_spaces(s):
    c = 0
    for x in s:
        if x == ' ':
            c += 1
    return c
```

This works for other operators too.

4

This is a classic problem. It sounds easy, and yet requires several changes. We calculate the length of the string with `len`, so we know how many letters there are left to go. Then, in the **`for`** loop itself, we deduct one from that count each time, and print the space only if the count indicates we are not on the last letter.

```
def print_spaced(s):
    l = len(s)
    for x in s:
        print(x, end='')
        l = l - 1
        if l > 0:
            print(' ', end='')
```

5

A very simple **`while`** loop is required:

```
def sentence_checker():
    text = 'Jackdaws love my sphinx of Quartz'
    print(text)
    while input() != text:
        print('Incorrect! Try again...')
    print('Correct!')
```

6

We supply the prompt directly to the `input` function. We must add the newline `\n` because, unlike `print`, `input` does not move to the next line after printing the prompt.

```
def ask_for_password():
  entered = ''
  while entered != 'please':
    entered = input('Please enter the password\n')
```

We can now remove the `entered` variable, but we must use **`pass`**, otherwise the **`while`** statement would be ungrammatical in Python:

```
def ask_for_password():
    while input('Please enter the password\n') != 'please':
        pass
```

7

We need three variables: `target` to hold the secret number between 1 and 100, `guess` to hold the current guess, and `tries` to count the number of tries.

```
import random

def guessing_game():
    target = random.randint(1, 100)
    guess = int(input('Guess a number between 1 and 100\n'))
    tries = 1
    while guess != target:
        tries = tries + 1
        if guess < target:
            guess = int(input('Higher!\n'))
        elif guess > target:
            guess = int(input('Lower!\n'))
    print('Correct! You took ' + str(tries) + ' guesses.')
```

Inside the **`while`** loop, we add one to the number of tries, and keep going until the correct answer is guessed. Then we print the final message with the number of tries. Notice that we have use an **`if`** and an **`elif`** but no **`else`**. Can you simplify the conditional construct further?

8

We use one giant **`if`** construct (in the next chapter we shall discuss better ways to do this). The function `print_morse_letter` prints a single code for the given letter, followed by three spaces:

```
def print_morse_letter(l):
    if l == 'A': print('. -', end='   ')
    elif l == 'B': print('- . . .', end='   ')
    elif l == 'C': print('- . - .', end='   ')
    elif l == 'D': print('- . .', end='   ')
    elif l == 'E': print('.', end='   ')
    elif l == 'F': print('. . - .', end='   ')
    elif l == 'G': print('- - .', end='   ')
    elif l == 'H': print('. . . .', end='   ')
    elif l == 'I': print('. .', end='   ')
    elif l == 'J': print('. - - -', end='   ')
    elif l == 'K': print('- . -', end='   ')
    elif l == 'L': print('. - . .', end='   ')
    elif l == 'M': print('- -', end='   ')
    elif l == 'N': print('- .', end='   ')
    elif l == 'O': print('- - -', end='   ')
    elif l == 'P': print('. - - .', end='   ')
    elif l == 'Q': print('- -. -', end='   ')
    elif l == 'R': print('- . -', end='   ')
    elif l == 'S': print('. . .', end='   ')
    elif l == 'T': print('-', end='   ')
    elif l == 'U': print('. . -', end='   ')
    elif l == 'V': print('. . . -', end='   ')
    elif l == 'W': print('. - -', end='   ')
    elif l == 'X': print('- . . -', end='   ')
    elif l == 'Y': print('- . - -', end='   ')
    elif l == 'Z': print('- - . .', end='   ')
    elif l == '1': print('. - - - -', end='   ')
    elif l == '2': print('. . - - -', end='   ')
    elif l == '3': print('. . . - -', end='   ')
    elif l == '4': print('. . . . -', end='   ')
    elif l == '5': print('. . . . .', end='   ')
    elif l == '6': print('- . . . .', end='   ')
    elif l == '7': print('- - . . .', end='   ')
    elif l == '8': print('- - - . .', end='   ')
    elif l == '9': print('- - - - .', end='   ')
    elif l == '0': print('- - - - -', end='   ')
    else: print('bad letter')
```

Now the main function, to print a whole string, uses `print_morse_letter` for each letter which is not a space. For spaces it prints an extra four spaces in the output to add to the three following the previous letter.

```
def print_morse(s):
    for l in s:
        if l == ' ': print('    ', end='')
        else: print_morse_letter(l)
    print('')
```

This implementation has two problems: (1) it prints an extra three spaces at the end of any message not ending with a space; and (2) a space at the beginning of a message will have only four spaces in the output not seven. Can you fix them?

Chapter 4 (Making Lists)

1

The `first` function is simple: we just return the element at index 0. To return the last element we must use the `len` function to calculate the index. We remember to subtract one, since list indices start at zero.

```
def first(l):
    return l[0]

def last(l):
    return l[len(l) - 1]
```

In fact, we can also write `l[-1]` to retrieve the last element of a list in Python. What happens in each case if the list is empty?

2

We first create a fresh, empty list. Then we can iterate over the input list in order, inserting each element at index 0 in the new list. The effect is to produce a reversed list, which is then returned.

```
def reverse(l):
    l2 = []
    for x in l:
        l2.insert(0, x)
    return l2
```

Alternatively, we can use a `range` with a negative step value, in conjunction with the `append` method. Again, we begin with a fresh, empty list.

```
def reverse(l):
    l2 = []
    for x in range(len(l) - 1, -1, -1):
        l2.append(l[x])
    return l2
```

3

We will use a **for** loop to look at each element, updating two variables to keep track of the smallest and largest numbers seen so far. The important thing to to properly initialise the `minimum` and `maximum` variables. We do so by setting them to be equal to the first element of the list. Can you see why?

```
def minmax(l):
    minimum = l[0]
    maximum = l[0]
    for x in l:
        if x < minimum: minimum = x
        if x > maximum: maximum = x
    print('Minimum is ' + str(minimum))
    print('Maximum is ' + str(maximum))
```

This function has a minor inefficiency; it looks at the first element of the list twice. Can you fix that?

4

We need a step value of two, and start and stop values encompassing the whole list:

```
def evens(l):
    return l[0:len(l) + 1:2]
```

Of course, such start and stop values are the default, so we may also write:

```
def evens(l):
    return l[::2]
```

5

We follow the pattern of our second, shorter `evens` function above, and write simply:

```
def reverse(l):
    return l[::-1]
```

6

We begin by making a fresh, empty list. Then, for each element in the original list, we add it to the new list, unless it is already there:

```
def setify(l):
    l2 = []
    for x in l:
        if x not in l2: l2.append(x)
    return l2
```

7

We use our `setify` function to make a new list of the unique items in the original list. Then we iterate over this new list, finding the number of each of its elements in the original list using the `count` method, and print it.

```
def histogram(l):
    unique = setify(l)
    for x in unique:
        print(str(x) + ' appears ' + str(l.count(x)) + ' times.')
```

8

This is a simple exercise in the use of **`in`** and the boolean operator **`and`**:

```
def contains_all(s, a, b, c):
    return a in s and b in s and c in s
```

9

We create a fresh, empty list and append the elements of the original list one by one:

```
def copy_list(l):
    l2 = []
    for x in l: l2.append(x)
    return l2
```

Alternatively, we can use the slice operator with empty start and stop values:

```
def copy_list(l):
    return l[:]
```

10

We make a fresh list with our new `copy_list` function, remove the value using `remove` and then return the new list:

```
def remove_copy(l, x):
    l2 = copy_list(l)
    l2.remove(x)
    return l2
```

11

We set up our alphabet and use list slicing to write a function `rotate` which can rotate it by any number of places from 1 to 25, returning a new string:

```
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVQXYZ'

def rotate(n, a):
    return a[n:] + a[:n]
```

Now our encoding and decoding functions are simple, taking the text to encode or decode, and the rotated cipher. They are, as you would expect, somewhat symmetrical:

```
def encode(text, cipher):
    out = ''
    for x in text:
        if x == ' ': out = out + ' '
        else: out = out + cipher[alphabet.index(x)]
    return out

def decode(text, cipher):
    out = ''
    for x in text:
        if x == ' ': out = out + ' '
        else: out = out + alphabet[cipher.index(x)]
    return out
```

We remember to treat spaces specially.

12

Now that we know about Python's lists, we can dispense with the huge **`if`** construct of our previous Morse code solution, and work from two lists: the letters and their codes:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
           'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
           'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
           'Y', 'Z', '0', '1', '2', '3', '4', '5',
           '6', '7', '8', '9', '0']

codes = ['. -', '- . . .', '- . - .', '- . .',
         '.', '. . - .', '- - .', '. . . .',
         '. .', '. - - -', '- . -', '. - . .',
         '- -', '- .', '- - -', '. - - .',
         '- - . -', '. - .', '. . .', '-',
         '. . -', '. . . -', '. - -', '- . . -',
         '- . - -', '- - . .', '. - - - -', '. . - - -',
         '. . . - -', '. . . . -', '. . . . .', '- . . . .',
         '- - . . .', '- - - . .', '- - - - .', '- - - - -']
```

Now it is simple to modify our previous solution to look up codes in the list using the `index` method on lists:

```
def print_morse_letter(l):
    if l not in letters: print('bad letter')
    else: print(codes[letters.index(l)], end='   ')

def print_morse(s):
    for l in s:
        if l == ' ': print('    ', end='')
        else: print_morse_letter(l)
    print('')
```

13

First, we shall write a function which looks at a single guess calculates how many are a) the correct number in the correct place and b) the correct number in the incorrect place. This is surprisingly delicate, since we must make sure not to double count anything – for example, if the code is `1441` and the guess is `4444` we should identify two `4`s as being correct numbers in the correct places, but the other two `4`s are not identified as being correct numbers in the wrong place, because we have already used up all the `4`s in our code.

Our function will take two list of four numbers each, the code and the guess. We copy them, since we will be changing values in the lists to mark them as used. Then we set up two counters, to keep track of how many numbers are correct and in the right place, or correct and in the wrong place.

```
import random

def check_code_guess(code, guess):
    code = code.copy()
    guess = guess.copy()
    correct = 0
    correct_place = 0
    for x in range(0, 4):              first pass, correct number in correct place
        if guess[x] == code[x]:
            correct = correct + 1
            code[x] = -1                                        mark code as used
            guess[x] = -1                                       mark guess as used
    for x in range(0, 4):
        if guess[x] > -1:
            if guess[x] in code:                               if in remaining code
                correct_place = correct_place + 1
                code[code.index(guess[x])] = -1                      mark as used
    print('Correct number in correct place: ' + str(correct))
    print('Correct number in incorrect place: ' + str(correct_place))
    return code == guess
```

The function returns `True` if the code is completely correct, and `False` otherwise. Now we can write the main function which asks the user for a guess repeatedly:

```
def code_guesser():
    a = random.randint(1, 9)                          collect four random digits
    b = random.randint(1, 9)
    c = random.randint(1, 9)
    d = random.randint(1, 9)
    code = [a, b, c, d]
    tries = 1
    i = input()
    guess = [int(i[0]), int(i[1]), int(i[2]), int(i[3])]   extract integers
    while guess != code:
        if check_code_guess(code, guess):
            pass
        else:
            tries = tries + 1
            i = input()
            guess = [int(i[0]), int(i[1]), int(i[2]), int(i[3])]
    print('Correct. You took ' + str(tries) + ' guesses.')
```

There is currently no handling of errors here - what happens if you type in too few or too many numbers, for instance? Can you fix the program to handle this?

Chapter 5 (More with Lists and Strings)

1

We `split` to make a list, then the `sort` method to sort it in place.

```
def sorted_words(s):
    l = s.split()
    l.sort()
    return l
```

2

Using the `sorted` function removes the need to introduce the intermediate name `l` as in the previous question:

```
def sorted_words(s):
    return sorted(s.split())
```

This makes our function a little easier to read.

3

Here is the original from chapter 4:

```
def setify(l):
    l2 = []
    for x in l:
        if x not in l2: l2.append(x)
    return l2

def histogram(l):
    unique = setify(l)
    for x in unique:
        print(str(x) + ' appears ' + str(l.count(x)) + ' times.')
```

The modification is very simple: we use the `sorted` function when creating our list of unique values:

```
def histogram(l):
    unique = sorted(setify(l))
    for x in unique:
        print(str(x) + ' appears ' + str(l.count(x)) + ' times.')
```

4

If we write a function to remove any spaces at the front of a list of strings, we can then use it multiple times to deal with spaces at the beginning and end. Here is such a function:

```
def strip_leading_spaces(l):
    while len(l) > 0 and l[0] == ' ':
        del l[0]
```

For example, `strip_leading_spaces([' ', ' ', 'y', 'e', 's', ' '])` will return `['y', 'e', 's', ' ']`. Notice that the **and** operator does not try its right hand side if its left hand side is false. And so, if `len(l) > 0` is `True`, the first element of `l` will not be tested for equality, and the function succeeds even when the list is empty (or consists only of spaces).

Now we can write the main function, which uses our stripper twice, to remove the spaces at the beginning and end of the list made from the original string. One final reversal brings it back to the correct order, and we `join` it back into a string.

```
def remove_spaces(s):
    l = list(s)
    strip_leading_spaces(l)
    l.reverse()
    strip_leading_spaces(l)
    l.reverse()
    return ''.join(l)
```

5

We can (ab)use `split` and `join` to make a much simpler definition:

```
def remove_spaces(s):
    return ' '.join(s.split())
```

This will, however, also remove any excess multiple spaces in between words. Python provides a built-in method `strip` to remove just the parts at either end, leaving the rest untouched.

6

First, our `clip` function:

```
def clip(x):
    if x > 10:
        return 10
    elif x < 1:
        return 1
    else:
        return x
```

Now, we use `map` with our `clip` function, not forgetting to use `list` to get back an ordinary list before returning.

```
def clip_list(l):
    return list(map(clip, l))
```

7

We have already seen how a slice with a step of `-1` may be used to reverse a list. A palindrome is something which equals its own reverse, so it is easy to write out the definition:

```
def is_palindromic(s):
    return s == s[::-1]
```

We can use this `is_palindromic` function as a filter to return only such strings in a list as are palindromic:

```
def palindromes(l):
    return list(filter(is_palindromic, l))
```

Now, we can build only those numbers in a range whose strings are palindromic.

```
def palindromic_numbers_in(x, y):
    ps = palindromes(list(map(str, list(range(x, y)))))
    return list(map(int, ps))
```

We must remember to convert them back to integers before returning. We can achieve this with `map`, of course. Now we can find all the palindromic numbers up to 500:

```
>>> palindromic_numbers_in(1, 500)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55, 66, 77, 88, 99, 101, 111, 121,
131,141, 151, 161, 171, 181, 191, 202, 212, 222, 232, 242, 252, 262, 272, 282,
292,303, 313, 323, 333, 343, 353, 363, 373, 383, 393, 404, 414, 424, 434, 444,
454, 464, 474, 484, 494]
```

We can remove the instances of `list` since `range`, `filter`, and `map` are happy to accept iterators:

```
def is_palindromic(s):
    return s == s[::-1]

def palindromes(l):
    return filter(is_palindromic, l)

def palindromic_numbers_in(x, y):
    ps = palindromes(map(str, range(x, y)))
    return list(map(int, ps))
```

We have to chosen to return an actual list, not a generator, from the final `palindromic_numbers_in` function, however.

8

This is a simple list comprehension, just using `clip` on each item in the list:

```
def clip_list(l):
    return [clip(x) for x in l]
```

9

In this example, we use both **`for`** and **`if`** in a list comprehension to process the list of strings.

```
def palindromic_numbers_in(x, y):
    strings = map(str, range(x, y))
    return [int(x) for x in strings if is_palindromic(x)]
```

It is just about readable to put the whole thing in one expression:

```
def palindromic_numbers_in(x, y):
    return [int(x) for x in map(str, range(x, y)) if is_palindromic(x)]
```

Chapter 6 (Prettier Printing)

1

We must keep a counter to prevent the printing of an extra comma and space:

```
def print_list(l):
    length = len(l)
    print('[', end='')
    for x in l:
        print(x, end='')
        length -= 1
        if length > 0: print(', ', end='')
    print(']')
```

This is a lot more complicated than having `print` do the work for us, but it does allow us some customisation: for example if we wished to print without commas, or without spaces.

2

In this instance, format strings do not really help use remove any complexity. In fact, this solution is slightly longer than the one without format strings.

```
def print_list(l):
    length = len(l)
    print('[', end='')
    for x in l:
        length = length - 1
        if length > 0:
            print(f'{x}, ', end='')
        else:
            print(f'{x}', end='')
    print(']')
```

3

Without format strings, we must use `str` explicitly on each integer, as a prelude to calling the `rjust` method. We can use the `print` function with multiple arguments to print a whole line, though:

```
def print_powers(n):
    for x in range(1, n):
        x1 = str(x).rjust(5)
        x2 = str(x ** 2).rjust(5)
        x3 = str(x ** 3).rjust(5)
        x4 = str(x ** 4).rjust(5)
        x5 = str(x ** 5).rjust(5)
        print(x1, x2, x3, x4, x5)
```

4

This is a simple substitution of `zfill` for `rjust`:

```
def print_powers(n):
    for x in range(1, n):
        x1 = str(x).zfill(5)
        x2 = str(x ** 2).zfill(5)
        x3 = str(x ** 3).zfill(5)
        x4 = str(x ** 4).zfill(5)
        x5 = str(x ** 5).zfill(5)
        print(x1, x2, x3, x4, x5)
```

In fact, we can use `rjust(5, '0')` to achieve the same effect.

5

We use the **with** … **as** construct to safely open and close the file. Then it is a matter of carefully constructing the **while** condition and the variable it sets to get the right result. We then rearrange the parts of the name the user types in, and print it to file:

```
def names_to_file(filename):
    with open(filename, 'w') as f:
        name = 'not empty'
        while name != '':
            name = input('Title, forename and surname, please: ')
            if name != '':
                words = name.split()
                print(words[2], words[1], words[0], sep=', ', file=f)
```

Can you see why we had to initialise the `name` variable to a non-empty string?

6

Using a format string allows us to remove the `sep=', '` argument, but not a lot else:

```
def names_to_file(filename):
    with open(filename, 'w') as f:
        name = 'not empty'
        while name != '':
            name = input('Title, forename and surname, please: ')
            if name != '':
                words = name.split()
                print(f'{words[2]}, {words[1]}, {words[0]}', file=f)
```

7

For each sentence in the list, we find the position (if any) of the word. Remembering that failure to find the word results in a position of -1, we decide what to print to the screen:

```
def number_found(sentences, word):
    n = 0
    for s in sentences:
        n += 1
        p = s.find(word)
        if p == -1:
            print(f'{word} not found in sentence {n}')
        else:
            print(f'{word} found at position {p} in sentence {n}')
```

8

This is a simple alteration to the previous answer:

```
def number_found(sentences, word, filename):
    n = 0
    with open(filename, 'w') as f:
        for s in sentences:
            n += 1
            p = s.find(word)
            if p == -1:
                print(f'{word} not found in sentence {n}', file=f)
            else:
                print(f'{word} found at position {p} in sentence {n}', file=f)
```

Chapter 7 (Arranging Things)

1

This can be achieved by tuple unpacking:

```
Python
>>> a = 1
>>> b = 2
>>> a, b = (b, a)
>>> a
2
>>> b
1
```

Note we don't need parentheses on the tuple when doing multiple assignment of values to names:

```
Python
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

However, we cannot write this:

```
def swap(a, b): a, b = b, a
```

Why not?

2

We can use the `items` method on the dictionary, which allows iterating with a **`for`** loop using two variables, one for the key and one for the value. We return the result as a tuple.

```
def unzip(d):
    ks = []
    vs = []
    for k, v in d.items():
        ks.append(k)
        vs.append(v)
    return (ks, vs)
```

For example:

```
Python
>>> unzip({1: 'one', 2: 'two'})
([1, 2], ['one', 'two'])
```

3

We initialise a fresh, empty dictionary. Then, looping over the index positions in the list of keys, we add each key and its value to the dictionary.

```
def dict_of_keys_and_values(ks, vs):
    d = {}
    for x in range(0, len(ks)):
        d[ks[x]] = vs[x]
    return d
```

What happens if the lists `ks` and `vs` are of differing lengths?

4

Beginning with an empty dictionary, we loop over the items in each existing dictionary, adding the key and its associated value to the union dictionary.

```
def union(a, b):
    u = {}
    for x in b: u[x] = b[x]
    for x in a: u[x] = a[x]
    return u
```

The preference for values from dictionary `a` is achieved by processing it second. Duplicate entries from dictionary `b` are thus overwritten.

5

The list is being modified by deletion during the **`for`** loop, and so the indices change. Here is a possible working version, which repeatedly uses the `remove` method which, we remember, removes the first instance of a given element in a list:

```
def remove_zeroes(l):
    while 0 in l:
        l.remove(0)
```

6

We need to assign values to two names here, so we use the `items` method. The rest is then simple:

```
def reverse_dict(d):
    return {v:k for k, v in d.items()}
```

The output is not always the same length as the input, because a value may appear multiple times in the input, and so be used multiple times as a key in the output:

```
Python
>>> reverse_dict({1: 2, 2: 1, 3: 1})
{2: 1, 1: 3}
```

7

We remember that an empty set is created by `set()`. We loop over the input words, using `set` again to build a set of all the letters in each word, and the `|` operator to add them to our master set, which we then return:

```
def letter_set(l):
    letters = set()
    for x in l:
        letters = letters | set(x)
    return letters
```

For example:

```
Python
>>> letter_set(['one', 'two', 'three'])
{'w', 'n', 't', 'h', 'o', 'r', 'e'}
```

To do the inverse, we shall need a set of all the letters. Then we can use the set difference operator.

```
alphabet = set('qwertyuiopasdfghjklzxcvbnm')

def letters_not_used(l):
    return alphabet - letter_set(l)
```

For example:

```
Python
>>> letters_not_used(['one', 'two', 'three'])
{'m', 'd', 'f', 'q', 'l', 'y', 's', 'k', 'g', 'c', 'v', 'j', 'p', 'a', 'u', 'z',
'x', 'b', 'i'}
```

8

We can represent sets using dictionaries with the values ignored, for example all set to zero. Here is a function to build such a 'set' from a list:

```
def dset_of_list(l):
    set = {}
    for x in l:
        set[x] = 0
    return set
```

Now we can implement the operations. First, for the 'or' operation, we add entries to the new dictionary from both input lists:

```
def dset_or(a, b):
    result = {}
    for x in a: result[x] = 0
    for x in b: result[x] = 0
    return result
```

For 'and', we must check that the item is in both sets:

```
def dset_and(a, b):
    result = {}
    for x in a:
        if x in b:
            result[x] = 0
    return result
```

Set difference is very similar:

```
def dset_minus(a, b):
    result = {}
    for x in a:
        if x not in b:
            result[x] = 0
    return result
```

Finally, exclusive or can be achieved by using our existing functions:

```
def dset_exclusive_or(a, b):
    return dset_or(dset_minus(a, b), dset_minus(b, a))
```

9

We use two **for** portions to iterate over both input sets. Only when `x == y` do we have a match.

```
def comp_and(a, b):
    return {x for x in a for y in b if x == y}
```

This code checks every possible combination of elements of `a` and `b` and so is not very efficient.

10

If the type of the input value `t` is an integer, we return it. Otherwise, we loop over all the items in `t`, adding up their sums by recursive application of the `sum_all` function itself.

```
def sum_all(t):
    if type(t) == int:
        return t
    else:
        total = 0
        for x in t:
            total += sum_all(x)
        return total
```

The result works on any tuple containing only number and on numbers themselves:

```
Python
>>> sum_all((1, 2, 3))
6
>>> sum_all((1, (1, 2), 3))
7
>>> sum_all(10)
10
```

Chapter 8 (When Things Go Wrong)

1

We handle the `ValueError` resulting from `int` being used on a string which cannot reasonably be converted to an integer, and ignore the error by using **`pass`**:

```
def list_sum(l):
    total = 0
    for x in l:
        try:
            total += int(x)
        except ValueError:
            pass
    return total
```

Since the exception is raised by `int`, the `total` variable will not be updated in the case of a bad string. So we need not worry about the `+=` operation receiving a bad input.

2

We write two little functions. First, `safe_int`, which handles the `ValueError` exception raised by `int` and returns `None` instead. Second, the function `not_none` which returns `True` if a value is anything other than `None`. Then we can apply `map` and `filter` to build a list of results from `safe_int` and filter out the `None` values.

```
def safe_int(s):
    try:
        return int(s)
    except ValueError:
        return None

def not_none(x):
    return x != None

def list_sum(l):
    return sum(filter(not_none, map(safe_int, l)))
```

3

We handle the `ZeroDivisionError` exception, returning 0.

```
def safe_division(x, y):
    try:
        return x / y
    except ZeroDivisionError:
        return 0
```

4

We begin with a fresh dictionary. Then, we iterate over the keys and values of dictionary `a`. We try to insert the corresponding value from `b` into the new dictionary. If it fails, we handle `KeyError` and simply skip that key.

```
def dict_take(a, b):
    c = {}
    for k, v in a.items():
        try:
            c[k] = b[k]
        except KeyError:
            pass
    return c
```

5

It is easy to add all the items from our first dictionary to the new one – the keys are already unique. When we add items from the second, we check to see if the key exists already. If it does, we raise `KeyError`.

```
def safe_union(a, b):
    c = {}
    for k, v in a.items():
        c[k] = v
    for k, v in b.items():
        if k in c:
            raise KeyError
        else:
            c[k] = v
    return c
```

6

We check to see if the item is already in the set. If it is, we raise `KeyError`. If not, we add it as usual.

```
def add_exception(s, k):
    if k in s:
        raise KeyError
    else:
       s.add(k)
```

Chapter 9 (More with Files)

1

The **with** ... **as** construct allows us to combine the two statements in the original into a single block:

```
with open('gregor.txt') as f:
    for line in f:
        print(line, end='')
```

2

We use **with** ... **as** again, using the optional `file` argument of the `print` function to write each key and value:

```
def dict_to_file(d, filename):
    with open(filename, 'w') as f:
        for k, v in d.items():
            print(k, file=f)
            print(v, file=f)
```

3

This is a good example of the complications of reading from a file, expecting entries in a certain format, and finding data not fitting such a format. We begin with an empty dictionary, and then enter a **while** `True` loop. We then try to read keys and values, returning if we have reached the end of the file (or if the line is empty).

```
def dict_from_file(filename):
    d = {}
    with open(filename) as f:
        while True:
            try:
                k = f.readline()
                v = f.readline().strip()              to remove newline
                if k !='' and v !='':
                    d[int(k)] = v        int will remove the newline itself
                else:
                    return d
            except ValueError:
                print(f'{k} is not an integer')
```

The `ValueError` exception which may be raised is caught and a message is printed.

4

We open the two input files, and the output file in 'append' mode. Then it is as simple as copying the lines across, being sure not to introduce extra newlines.

```
def append_files(a, b, c):
    with open(a) as f_a, open(b) as f_b, open(c, 'a') as f_c:
        print(f_a.read(), file=f_c, end='')
        print(f_b.read(), file=f_c, end='')
```

5

We use `read` to get the whole contents of the file at once, split it into 'words', then convert them to integers with `map` and sum them:

```
def sum_file(filename):
    with open(filename) as f:
        return sum(map(int, f.read().split()))
```

6

This is similar to our `append_files` function from question 4:

```
def copy_file(a, b):
    with open(a) as f_in, open(b, 'w') as f_out:
        print(f_in.read(), file=f_out, end='')
```

7

We introduce a dictionary to store the character histogram, then create or increment an entry in the dictionary for each character encountered. See page 184 for the program.

8

For the word histogram, we introduce a function `clean_split` which splits a line into words, then processes each word to remove punctuation, and convert to lowercase. See page 185 for the program.

9

We can reuse `clean_split` here to get the words in each line. Then, we can use `enumerate` to iterate over the indices and lists of words for each line. We check for the presence of the search term, and print the line and its number if required. See page 186 for the program.

```
def is_full_stop(s):
    return s == '.'

def stats_from_file(f):
    lines = 0
    characters = 0
    words = 0
    sentences = 0
    histogram = {}
    for line in f:
        lines += 1
        characters += len(line)
        words += len(line.split())
        sentences += len(filter(is_full_stop, line))
        for x in line:
            current = 0
            if x in histogram:            get the current count, if it exists
                current = histogram[x]
            histogram[x] = current + 1
    return (lines, characters, words, sentences, histogram)

def stats_from_filename(filename):
    with open(filename) as f:
        return stats_from_file(f)
```

```
import string

def clean_split(line):
    return
        [s.strip(string.punctuation).lower() for s in line.split()]

def is_full_stop(s):
    return s == '.'

def stats_from_file(f):
    lines = 0
    characters = 0
    words = 0
    sentences = 0
    histogram = {}
    word_histogram = {}
    for line in f:
        lines += 1
        characters += len(line)
        words += len(line.split())
        sentences += len(filter(is_full_stop, line))
        for x in line:
            current = 0
            if x in histogram:
                current = histogram[x]
            histogram[x] = current + 1
        for x in clean_split(line):
            current = 0
            if x in word_histogram:
                current = word_histogram[x]
            word_histogram[x] = current + 1
    return
        (lines, characters, words, sentences,
         histogram, word_histogram)

def stats_from_filename(filename):
    with open(filename) as f:
        return stats_from_file(f)
```

```
import string

def clean_split(line):
    return
       [s.strip(string.punctuation).lower() for s in line.split()]

def search_word(filename, word):
    lines = []
    with open(filename) as f:
        lines = f.readlines()
    words = map(clean_split, lines)
    for n, ws in enumerate(words):
        if word in ws:
            print(f'{n}: ', end='')
            print(lines[n], end='')
```

10

We read the lines all at once with `readlines`. Then, by careful use of slices, we print five at a time, waiting for the user to press Enter.

```
def top(filename):
    lines = []
    with open(filename) as f:
        lines = f.readlines()
    while len(lines) > 0:
        for l in lines[:5]: print(l, end='')
        lines = lines[5:]
        enter = input()
```

How might this be rewritten to work well on huge files? In that case, reading all the lines at once would be inefficient.

Chapter 10 (The Other Numbers)

1

We calculate the ceiling and floor, and return the closer one, being careful to make sure that a point equally far from the ceiling and floor is rounded up.

```
import math

def round(x):
    c = math.ceil(x)
    f = math.floor(x)
    if c - x <= x - f:
        return c
    else:
        return f
```

2

The function returns another point, and is simple arithmetic.

```
def between(a, b):
    x0, y0 = a
    x1, y1 = b
    return ((x0 + x1) / 2, (y0 + y1) / 2)
```

3

The whole part is calculated using the `floor` function. We return a tuple, the first number being the whole part, the second being the original number minus the whole part. In the case of a negative number, we must be careful – `floor` always rounds downward, not toward zero!

```
import math

def parts(x):
    if x < 0:
        a, b = parts(-x)
        return (-a, b)
    else:
        return (math.floor(x), x - math.floor(x))
```

Notice that we are using the unary operator `-` to make the number positive.

4

We need to determine at which column the asterisk will be printed. It is important to make sure that the range $0 \ldots 1$ is split into fifty equal sized parts, which requires some careful thought. Then, we just print enough spaces to pad the line, add the asterisk.

```
import math

def star(x):
    i = math.floor(x * 50)
    if i == 50: i = 49
    print(' ' * (i - 1) + '*')
```

5

Our function takes another function as one of its argument. We use a variable to hold the current value, starting at the beginning of the range, and then loop until we are outside the range.

```
def plot(f, a, b, dy):
  pos = a
  while pos <= b:
      star(f(pos))
      pos += dy
```

No allowance has been made here for bad arguments (for example, `b` smaller than `a`). Can you extend our program to move the zero-point to the middle of the screen, so that the sine function can be graphed even when its result is less than zero?

Chapter 11 (The Standard Library)

1

Here is the documentation for the `factorial` function from the `math` module.

> `math.`**`factorial`**(*x*)
>
> Return *x* factorial as an integer. Raises `ValueError` if *x* is not integral or is negative.

We can try it out:

```
Python
>>> import math
>>> math.factorial(5)
120
>>> math.factorial(5.0)
120
>>> math.factorial(-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: factorial() not defined for negative values
```

How does our function differ? Here we have picked the improved factorial function from the questions to chapter 2:

```
Python
>>> def factorial(x):
...     if x < 0:
...         return 0
...     elif x == 0:
...         return 1
...     else:
...         return x * factorial(x - 1)
>>> factorial(5)
120
>>> factorial(5.0)
120.0
>>> factorial(-4)
0
```

We return a floating-point number for a floating-point input, unlike `math.factorial`, and we return zero for a negative input, where `math.factorial` raises a `ValueError` exception.

2

We assume the string does represent an integer, then check each potential digit. If it is not in the string `string.digits`, we unset the `is_integer` variable. We then return the variable as the result of the function.

```
import string

def string_is_integer(s):
    is_integer = True
    for x in s:
        if x not in string.digits: is_integer = False
    return is_integer
```

There is one small problem: `string_is_integer('')` will return `True`. Can you fix this?

3

This is a simple modification: we replace the use of `random.randint` with one of `getpass.getpass`, passing the prompt as an argument.

```
import getpass

def guessing_game():
    target = int(getpass.getpass('What is the target number?'))
    guess = int(input('Guess a number between 1 and 100\n'))
    tries = 1
    while guess != target:
        tries += 1
        if guess < target:
            guess = int(input('Higher!\n'))
        elif guess > target:
            guess = int(input('Lower!\n'))
    print(f'Correct! You took {tries} guesses.')
```

4

This is simple. We return them as a tuple.

```
import statistics

def stats(l):
    return
        (statistics.median(l), statistics.mode(l), statistics.mean(l))
```

What happens when there is no modal value in a list?

5

We use two functions: `time.sleep`, which does nothing for a given number of seconds, allowing us to give the user a count-down; and `time.time` which returns a floating-point value representing the number of seconds since an arbitrary point in the past. By measuring the time twice, and subtracting, we get the elapsed time.

```
import time

def reaction_test():
    print('Ready? Press Enter when you see ENTER')
    print('3')
    time.sleep(1)
    print('2')
    time.sleep(1)
    print('1')
    time.sleep(1)
    print('ENTER')
    t = time.time()
    enter = input()
    t2 = time.time()
    print(f'Your reaction time was {t2 - t} seconds')
```

What happens if the user presses Enter too soon? Can you fix this?

Chapter 12 (Building Bigger Programs)

1

The `guessing_game` function is unaltered. We need simply to check that there are enough arguments in `sys.argv`. If there are, we pass the string representing the maximum number to `guessing_game`:

```
import random
import getpass
import sys

def guessing_game(maxnum):
    target = int(getpass.getpass('What is the target number?'))
    guess = int(input(f'Guess a number between 1 and {maxnum}\n'))
    tries = 1
    while guess != target:
        tries += 1
        if guess < target:
            guess = int(input('Higher!\n'))
        elif guess > target:
            guess = int(input('Lower!\n'))
    print('Correct! You took {tries} guesses.')

if len(sys.argv) > 1:
    guessing_game(sys.argv[1])
else:
    guessing_game('100')
```

If not, we use the default value of `'100'`. We could, in fact, pass the number `100` instead of the string `'100'`, since the `int` function does not care if it is passed something which is already an integer. This, however, would make the program as a whole more difficult to read. Better to keep our types consistent.

2

We first write the file `draw.py` with our existing plotter:

```
import math

def star(x):
    i = math.floor(x * 50)
    if i == 50: i = 49
    print(' ' * (i - 1) + '*')

def plot(f, a, b, dy):
  pos = a
  while pos <= b:
      star(f(pos))
      pos += dy
```

Now the main `plot.py` program can use **`import`** to access the `plot` function from the `draw` module, passing the fabricated function `f` built from the command line argument (one function can sit inside another):

```
import sys
import draw

if len(sys.argv) > 4:
    def f(x): return eval(sys.argv[1])
    draw.plot(f,
              float(sys.argv[2]),
              float(sys.argv[3]),
              float(sys.argv[4]))
else:
    print('Bad arguments')
```

And so we may write:

```
$ python plot.py 'x * x' 0 1 0.1
*
*
 *
   *
       *
           *
                *
                     *
                          *
                               *
                                    *
```

What errors might occur? The wrong number of arguments is handled in our program, but what if `float` fails?

3

We write three little functions to list, add, and remove notes:

```
import sys

#List notes from a filename, numbered 1..
def list_notes(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
        for n, l in enumerate(lines):
            print(f'{n + 1}: {l}', end='')

#Append note to the given filename
def add_note(filename, text):
    with open(filename, 'a') as f:
        print(text, file=f)

#Remove note from a filename, given its number.
def remove_note(filename, n):
    with open(filename, 'r') as f_in:
        lines = f_in.readlines()
    with open(filename, 'w') as f_out:
        del lines[n - 1]
        for line in lines:
            f_out.write(line)
```

The main part of the program, then, must decode the command line to decide which operation to do, and what parameters it needs. If the command line too short or malformed, we print a message.

```
#Main
if len(sys.argv) > 1:
    if sys.argv[1] == 'list':
        list_notes(sys.argv[2] + '.txt')
    elif sys.argv[1] == 'remove':
        remove_note(sys.argv[2] + '.txt', int(sys.argv[3]))
    elif sys.argv[1] == 'add':
        add_note(sys.argv[2] + '.txt', sys.argv[3])
    else:
        print('Unrecognized command')
```

Project 1: Pretty Pictures

1

We remember `square` takes the length of the side of the square as an argument:

```
def square(x):
    for _ in range(4):
        t.fd(x)
        t.rt(90)
```

Now we can write `many_squares` which takes how many square to use for the star, and the length of the sides, and calls `square` repeatedly.

```
def many_squares(n, l):
    for _ in range(n):
        square(l)
        t.rt(360.0 / n)
```

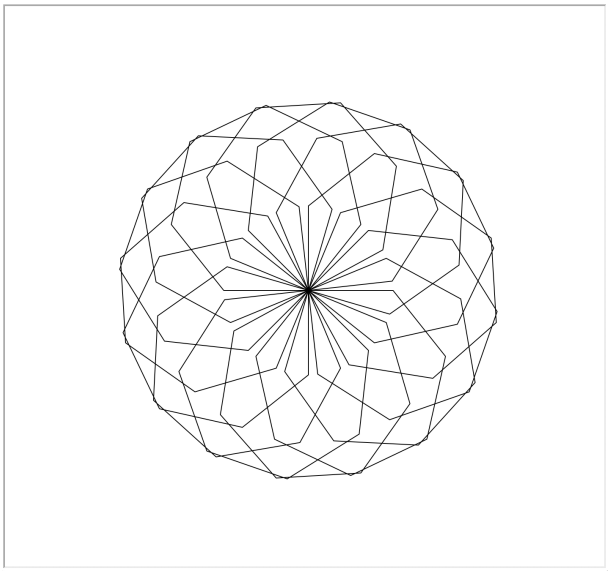
2

The `poly` function is unaltered. We define a new function `many_poly` which takes the number of sides, the number of polygons to draw, and the length of the side of each polygon, and calls `poly` repeatedly, turning between each one.

```
def poly(n, l):
    for _ in range(n):
        t.fd(l)
        t.rt(360.0 / n)

def many_poly(sides, number, side_length):
    for _ in range(number):
        poly(sides, side_length)
        t.rt(360.0 / number)
```

Here is the result of `many_poly(7, 16, 100)`:



3

The number of segments used to approximate a circle ought to be related to its circumference not its radius.

```
import math

def circle(r):
    circumference = 2.0 * math.pi * r
    poly(int(circumference), 1.0)
```

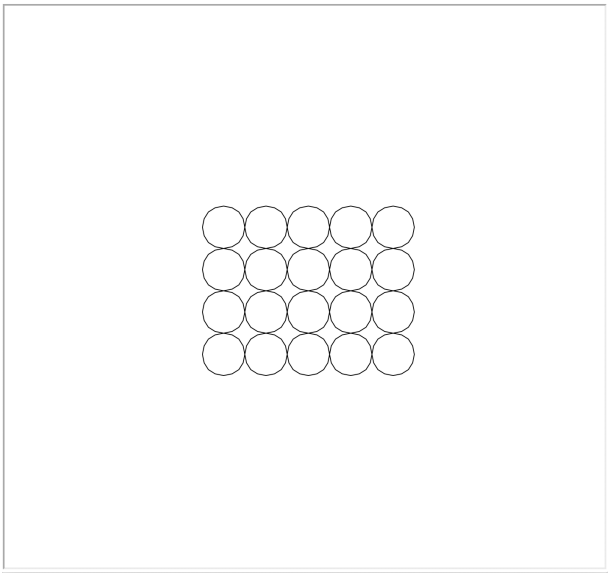
The smoothness may be fine-tuned by writing `int(circumference) * 1.5`, `int(circumference * 0.5)` etc.

4

We write a function `grid` which takes four arguments: the first two to represent the starting point (sx, sy) and the latter two to give the number of circles in the x and y directions:

```
def grid(sx, sy, nx, ny):
    for x in range(nx):
        for y in range(ny):
            t.penup()
            t.goto(sx + x * 50, sy + y * 50)
            t.pendown()
            t.circle(25)
```

Here is the result of `grid(-100, -100, 5, 4)`:



5

There are three dimensions to this data: red, green, and blue. And so we cannot display it directly on a 2D screen – we must flatten it in some way. We have chosen to slice the cube of data into slices based on the red value, and display the slices side by side.

First, we will need a function to draw a filled square of a given size at a given position.

```
def filled_square(x, y, l):
    t.penup()
    t.goto(x, y)
    t.begin_fill()
    t.setheading(90)
    for _ in range(4):
        t.fd(l)
        t.rt(90)
    t.end_fill()
```

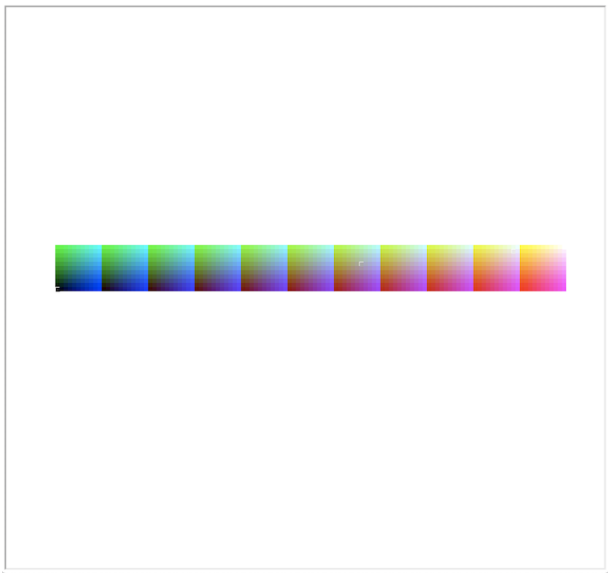
Now we write a function `red_gamut` to draw a slice of the cube at a given position for a given red value:

```
def red_gamut(x, y, r):
    for b in range(11):
        for g in range(11):
            t.color(r, g * 0.1, b * 0.1)
            filled_square(x + b * 5, y + g * 5, 5)
```

Now we can show them side by side.

```
def whole_gamut():
    for r in range(11):
        red_gamut(-300 + 55 * r, 0, r * 0.1)
```

Here is the result:



6

Like any other polygon, we simply use `begin_fill` and `end_fill`:

```
import math

def filled_circle(r):
    circumference = 2.0 * math.pi * r
    t.penup()
    t.begin_fill()
    poly(int(circumference), 1.0)
    t.end_fill()
```

PROJECT 1A

Here is one solution – there are, of course, many others. It has two interesting features. The first is a function which returns a function which evaluates the given formula using `eval`. This function may be passed to the graph plotter, and evaluated many times for increasing values of `x`:

```
def farg(arg):
    def f(x): return eval(arg)
    return f
```

The second is simply a method for cycling through a fixed number of colours in the event that the user wants to plot more graphs than we anticipate. Suppose we have a list of colours of length four:

```
colors = ["black", "red", "green", "blue"]
```

Then we can cycle through them with the `%` operator:

```
t.pencolor(colors[n % 4])
```

Here is the full program:

```
import sys
import turtle
import math

t = turtle.Turtle()

if len(sys.argv) < 2:
    print('No formula supplied')
    sys.exit(0)

def plot(f):
    t.penup()
    t.goto(-300, f(-300))
    t.pendown()
    for x in range(-300, 300, 1):
        t.goto(x, f(x))
```

```
def farg(arg):
    def f(x): return eval(arg)
    return f

def key(n, formula_text, color):
    t.color(color)
    t.penup()
    t.goto(-300, -200 - 20 * n)
    t.pendown()
    t.write(formula_text, font = ("Arial", 16, "normal"))

def line(x0, y0, x1, y1):
    t.penup()
    t.goto(x0, y0)
    t.pendown()
    t.goto(x1, y1)

def axes():
    t.color("black")
    line(-300, 0, 300, 0)
    line(0, -300, 0, 300)
    for x in range(-300, 301, 50):
        if x != 0:
            t.penup()
            t.goto(x, -20)
            t.pendown()
            t.write(str(x), font = ("Arial", 12, "normal"))
            line(x, -5, x, 5)
    for y in range(-300, 301, 50):
        if y != 0:
            t.penup()
            t.goto(-20, y)
            t.pendown()
            t.write(str(y), font = ("Arial", 12, "normal"))
            line(-5, y, 5, y)

colors = ["black", "red", "green", "blue"]

t.speed(0)

axes()

for n, arg in enumerate(sys.argv[1:]):
    t.pencolor(colors[n % 4])
    plot(farg(arg))
    key(n, arg, colors[n % 4])

turtle.mainloop()
```

PROJECT 1B

A clock face is a good example of a structure which is easier to produce with turtle-like commands than by calculating coordinates and using `goto`.

The main loop checks the time, clears the screen, and then draws the clock face. Then we use `turtle.Screen().update` to update the screen, and sleep for one second:

```
while True:
    tm = time.localtime()
    t.home()
    t.clear()
    clockface(tm.tm_hour, tm.tm_min, tm.tm_sec)
    turtle.Screen().update()
    time.sleep(1)
```

Here is the full program:

```
import turtle
import math
import time

def hand(length, thickness, angle):
    t.penup()
    t.home()
    t.setheading(90)
    t.pensize(thickness)
    t.pendown()
    t.rt(angle)
    t.fd(length)

def tickmarks():
    t.pensize(1)
    for a in range (0, 60):
        t.penup()
        t.home()
        t.setheading(90)
        t.rt(360 / 60 * a)
        t.fd(295)
        t.pendown()
        t.fd(5)

def clockface(h, m, s):
    t.penup()
    t.goto(0, -300)
    t.pensize(1)
    t.pendown()
    t.circle(300)
    tickmarks()
```

```
    hand(200, 3, 360 / 12 * (h % 12))
    hand(280, 3, 360 / 60 * m)
    hand(295, 1, 360 / 60 * s)

t = turtle.Turtle()
t.hideturtle()
turtle.Screen().tracer(0, 0)

while True:
    tm = time.localtime()
    t.home()
    t.clear()
    clockface(tm.tm_hour, tm.tm_min, tm.tm_sec)
    turtle.Screen().update()
    time.sleep(1)
```

Project 2: Counting Calories

1

We use the function `os.path.join` to combine the person's name and the name of the file where we expect to find the weights listed, and load the table with `table_of_file`. We can then iterate over the resultant table with the `items` method:

```
def list_weights(name):
    for k, vs in table_of_file(os.path.join(name, 'weight.txt')).items():
        print(f'{k} {vs[0]}')
```

Printing the dates and weights to the screen is then simple.

2

We use the suggested `os.listdir` function to get the list of filenames. We are not told anything about the order, so we sort it with `sorted` – owing to the format for dates which we have chosen, they sort correctly.

```
def list_dates(name):
    for filename in sorted(os.listdir(name)):
        if filename != 'weight.txt': print(filename[:-4])
```

We must exclude the `weight.txt` file, of course.

3

We must take account of the possibility that no weight was recorded for a given date. In this case, the table lookup will yield `None`.

```
def lookup_weight(name, date):
    table = table_of_file(os.path.join(name, 'weight.txt'))
    vs = table[date]
    if vs == None:
        print(f'No weight found for {date}')
    elif len(vs) > 0:
        print(f'Weight at {date} was {vs[0]}')
```

4

We make the new directory, then open a file in it. The file is created, even though we do not write anything to it.

```
def new_user(name):
    os.mkdir(name)
    with open(os.path.join(name, 'weight.txt'), 'w'):
        pass
```

6

Here is the full `csvcals.py` program:

```
import sys
import os
import datetime
import csv

def table_of_file(filename):
    with open(filename) as c:
        r = csv.reader(c)
        next(r)
        table = {}
        for row in r:
            table[row[0]] = row[1:]
        return table

def list_eaten(name, date):
    for k, vs in table_of_file(os.path.join(name, date) + '.csv').items():
        print(f'{k} {vs[0]}')

def list_weights(name):
```

```
    for k, vs in table_of_file(os.path.join(name, 'weight.csv')).items():
        print(f'{k} {vs[0]}')

def list_dates(name):
    for filename in sorted(os.listdir(name)):
        if filename != 'weight.csv': print(filename[:-4])

def list_foods():
    for k, vs in table_of_file('calories.csv').items():
        print(k, end=' ')
        for v in vs: print(v, end=' ')
        print('')

def lookup_calories(food):
    table = table_of_file('calories.csv')
    vs = table[food]
    if vs == None:
        print(f'Food {food} not found')
    else:
        if len(vs) > 1:
            weight = vs[0]
            calories = vs[1]
            print(f'There are {calories} calories in {weight}g of {food}')
        else:
            print(f'Malformed calorie entry for {food} in calories file')

def lookup_weight(name, date):
    table = table_of_file(os.path.join(name, 'weight.csv'))
    vs = table[date]
    if vs == None:
        print(f'No weight found for {date}')
    elif len(vs) > 0:
        print(f'Weight at {date} was {vs[0]}')

def total_date(name, date):
    calories = table_of_file('calories.csv')
    table = table_of_file(os.path.join(name, date) + '.csv')
    total = 0
    for k, vs in table.items():
        weight_and_calories = calories[k]
        reference_weight = int(weight_and_calories[0])
        reference_calories = int(weight_and_calories[1])
        calories_per_gram = reference_calories / reference_weight
        total += int(vs[0]) * calories_per_gram
    print(f'Total calories for {date}: {int(total)}')

def new_user(name):
    os.mkdir(name)
```

```
    with open(os.path.join(name, 'weight.csv'), 'w') as f:
        print('Date,Weight', file=f)

def date_today():
    d = datetime.datetime.now()
    return (f'{d.day:02}-{d.month:02}-{d.year}')

def eaten(name, food, grams):
    filename = os.path.join(name, date_today()) + '.csv'
    is_new = not os.path.exists(filename)
    with open(filename, 'a') as f:
        if is_new: print('Food,Weight', file=f)
        print(f'"{food}",{grams}', file=f)

def weighed(name, weight):
    filename = os.path.join(name, 'weight.csv')
    is_new = not os.path.exists(filename)
    with open(filename, 'a') as f:
        if is_new: print('Date,Weight', file=f)
        print(f'{date_today()},{weight}', file=f)

arg = sys.argv

if len(arg) > 1:
    cmd = arg[1]
    if cmd == 'list':
        if len(arg) > 3 and arg[2] == 'eaten':
            list_eaten(arg[3], arg[4])
        else:
            if arg[2] == 'weights' and len(arg) > 3:
                list_weights(arg[3])
            elif arg[2] == 'dates' and len(arg) > 3:
                list_dates(arg[3])
            elif arg[2] == 'foods':
                list_foods()
    elif cmd == 'lookup':
        if len(arg) > 2:
            if arg[2] == 'calories':
                lookup_calories(arg[3])
            elif arg[2] == 'weight' and len(arg) > 3:
                lookup_weight(arg[3], arg[4])
    elif cmd == 'total':
        if len(arg) > 3:
            total_date(arg[2], arg[3])
    elif cmd == 'newuser':
        if len(arg) > 2:
            new_user(arg[2])
    elif cmd == 'eaten':
```

```
        if len(arg) > 4:
            eaten(arg[2], arg[3], arg[4])
    elif cmd == 'weighed':
        if len(arg) > 3:
            weighed(arg[2], arg[3])
    else:
        print('Command not understood')
```

7

There are only two functions to change: those that write non-blank CSV files. We use `csv.writer` to create a CSV writer from the file, then the `writerow` method to write both column headers and data.

```
def eaten(name, food, grams):
    filename = os.path.join(name, date_today()) + '.csv'
    is_new = not os.path.exists(filename)
    with open(filename, 'a') as f:
        w = csv.writer(f)
        if is_new: w.writerow(['Food', 'Weight'])
        w.writerow([food, grams])

def weighed(name, weight):
    filename = os.path.join(name, 'weight.csv')
    is_new = not os.path.exists(filename)
    with open(filename, 'a') as f:
        w = csv.writer(f)
        if is_new: w.writerow(['Date', 'Weight'])
        w.writerow([date_today(), weight])
```

Project 3: Noughts and Crosses

1

A 1x1 game is always won by the first player to play on their first turn. A 2x2 game is always won by the first player to play on their second turn. In some sense, of course, 3x3 is not interesting either because, as every child finds out soon enough, a draw can always be forced. What happens with a 4x4 game?

2

The most important rules are winning when one can, and blocking the other player if they are about to win. When not in either of those situations, you might think about which spaces are better to hold, for example the centre square.

3

To make a random play, we can choose a number between zero and eight. If the space is blank, we play there. If not, we cycle around the positions until we find a blank one.

```
def random_play(pl, b):
    p = random.randint(0, 8)
    while b[p] != '_':
        p = (p + 1) % 9
    b[p] = pl
```

The function assumes that there is always at least one blank space to find. Now the `random_game` function is straightforward:

```
def random_game():
    b = emptyboard.copy()
    pl = 'O'
    while not (full(b) or wins('X', b) or wins('O', b)):
        print_board(b)
        print('')
        random_play(pl, b)
        if pl == 'O': pl = 'X'
        else: pl = 'O'
    print_board(b)
    print('Game over. Result:')
    if wins('O', b):
        print('O wins!')
    elif wins('X', b):
        print('X wins!')
    else:
        print('Draw!')
```

4

We ask for input from the user, in the form of a string. First, we check that it represents a digit, to avoid an error when using `int`. Then we check it is in range. Finally, we check the space is really blank. Only then can we make the move.

```
def human_move(board):
    n_input = input('Position 0..8? ')
    if n_input.isdigit():
        n = int(n_input)
        if n < 0 or n > 8:
            print('Board position must be from 0..8')
            human_move(board)
        else:
            if board[n] != '_':
                print('Position already taken')
                human_move(board)
            else:
                board[n] = 'O'
    else:
        print('Not a valid board position')
```

This function can also be written without recursion, with the use of a **while** loop.

5

Extending the `human_move` function is simple: we add an extra argument.

```
def human_move(pl, board):
    n_input = input('Position 0..8? ')
    if n_input.isdigit():
        n = int(n_input)
        if n < 0 or n > 8:
            print('Board position must be from 0..8')
            human_move(pl, board)
        else:
            if board[n] != '_':
                print('Position already taken')
                human_move(pl, board)
            else:
                board[n] = pl
    else:
        print('Not a valid board position')
        human_move(pl, board)
```

There are many ways we might choose to write the main `play` function. Here is one:

```
def play():
    pl = 'X'
    print('Board is numbered\n012\n345\n678\n')
    board = emptyboard.copy()
    while not (full(board) or wins('X', board) or wins('O', board)):
        print(f'Player {pl} to play...')
        human_move(pl, board)
        if pl == 'X':
            pl = 'O'
        else:
            pl = 'X'
        print_board(board)
    print('Game over. Result:')
    if wins('O', board):
        print('You win!')
    elif wins('X', board):
        print('Computer wins!')
    else:
        print('Draw!')
```

6

The empty corner and empty side tactics are simple (the order we search for a blank space does not matter):

```
def tactic_empty_corner(b):
    return try_to_take(b, [0, 2, 6, 8])

def tactic_empty_side(b):
    return try_to_take(b, [1, 3, 5, 7])
```

This tactic requires us to check that the opposite corner has been taken by the opposing side, so a single call to `try_to_take` cannot suffice.

```
def tactic_play_opposite_corner(b):
    if b[0] == 'X':
        if try_to_take(b, [8]): return True
    elif b[2] == 'X':
        if try_to_take(b, [6]): return True
    elif b[6] == 'X':
        if try_to_take(b, [2]): return True
    elif b[8] == 'X':
        return try_to_take(b, 0)
```

We can update the `computer_move` function, adding these three new tactics and removing our earlier `tactic_first_blank`, since the combination of the centre, empty corner and empty side tactics render it unused.

```
def computer_move(b):
    print('Computer has played:')
    if tactic_win(b):
        print('Used tactic_win')
        return
    if tactic_block(b):
        print('Used tactic_block')
        return
    if tactic_play_centre(b):
        print('Used tactic_centre')
        return
    if tactic_play_opposite_corner(b):
        print('Used tactic_play_opposite_corner')
        return
    if tactic_empty_corner(b):
        print('Used tactic_empty_corner')
        return
    if tactic_empty_side(b):
        print('Used tactic_empty_side')
        return
    print('No tactic applied: error in tactic implementations')
```

7

One solution is overleaf. The boolean `human_goes_first` is true if the human player moves first.

8

The fork tactic requires us to look at each pair of intersecting lines, trying to find two such lines each of which have one of our pieces and two blank spaces. If we find such a pair, and if the intersecting space is blank, we play it. Otherwise, the tactic fails. So we shall need a list of the pairs of intersecting lines in a board, together with the space at which they intersect:

```
intersecting_lines = [(h1, v1, 0), (h1, v2, 1), (h1, v3, 2),
                      (h2, v1, 3), (h2, v2, 4), (h2, v3, 5),
                      (h3, v1, 6), (h3, v2, 7), (h3, v3, 8),
                      (d1, h1, 0), (d1, h2, 4), (d1, h3, 8),
                      (d1, v1, 0), (d1, v2, 4), (d1, v3, 8),
                      (d2, h1, 2), (d2, h2, 4), (d2, h3, 6),
                      (d2, v1, 2), (d2, v2, 4), (d2, v3, 6),
                      (d1, d2, 4)]
```

Now for the fork tactic itself, we go through the pairs of intersecting lines. For each one, we look up those positions on the board. Then we can check the count of pieces in each, and check that the intersecting space is blank. If so, we make the play. If not, we return `False`.

```
def play(human_goes_first):
    print('Board is numbered\n012\n345\n678\n')
    board = emptyboard.copy()
    if human_goes_first:
        print('You go first...')
        print_board(board)
    else:
        print('Computer goes first...')
    while not (full(board) or wins('X', board) or wins('O', board)):
        if human_goes_first:
            human_move(board)
        else:
            computer_move(board)
        human_goes_first = not human_goes_first
        print_board(board)
    print('Game over. Result:')
    if wins('O', board):
        print('You win!')
    elif wins('X', board):
        print('Computer wins!')
    else:
        print('Draw!')
```

```
def tactic_fork(b):
    for (l, l2, i) in intersecting_lines:
        bl = [b[x] for x in l]
        bl2 = [b[x] for x in l2]
        l_fits = bl.count('_') == 2 and bl.count('X') == 1
        l2_fits = bl2.count('_') == 2 and bl2.count('X') == 1
        if l_fits and l2_fits and b[i] == '_':
            b[i] = 'X'
            return True
    return False
```

The block fork tactic is somewhat more complicated. Part of the condition (two intersecting lines with one opponent's piece and two blanks) is similar to the fork tactic, but we do not necessarily move to the intersection space, even if it is blank. First, we check to find a place to move which makes two of our pieces in a row. If so, we take it instead, forcing our opponent to block instead of fork.

The function `find_two_in_a_row`, given a board and a position, checks to see if the position, if taken, would make us two in a row. If so, we take it.

```
def find_two_in_row(b, p):
    if b[p] == '_':
      for l in lines:
          if p in l:
              bl = [b[x] for x in l]
              if bl.count('X') == 1 and bl.count('O') == 0:
                  b[p] = 'X'
                  return True
    else:
        return False
```

Now the main function checks the initial conditions, calls `two_in_a_row` as required and, should it fail each time, deals with the case of the intersecting space:

```
def tactic_block_fork(b):
    for (l, l2, i) in intersecting_lines:
        bl = [b[x] for x in l]
        bl2 = [b[x] for x in l2]
        l_fits = bl.count('_') == 2 and bl.count('O') == 1
        l2_fits = bl.count('_') == 2 and bl.count('O') == 1
        if l_fits and l2_fits and b[i] == '_':
            if find_two_in_row(b, l[0]): return True
            elif find_two_in_row(b, l[1]): return True
            elif find_two_in_row(b, l[2]): return True
            elif find_two_in_row(b, l2[0]): return True
            elif find_two_in_row(b, l2[1]): return True
            elif find_two_in_row(b, l2[2]): return True
            else:
                if b[i] == '_':
                    b[i] = 'X'
                    return True
    return False
```

Here is the complete `computer_move` function for all our tactics, including printing each one out if it is applied, for debugging purposes:

```
def computer_move(b):
    print('Computer has played:')
    if tactic_win(b):
        print('Used tactic_win')
        return
    if tactic_block(b):
        print('Used tactic_block')
        return
    if tactic_fork(b):
        print('Used tactic_fork')
        return
    if tactic_block_fork(b):
        print('Used tactic_block_fork')
        return
    if tactic_play_centre(b):
        print('Used tactic_centre')
        return
    if tactic_play_opposite_corner(b):
        print('Used tactic_play_opposite_corner')
        return
    if tactic_empty_corner(b):
        print('Used tactic_empty_corner')
        return
    if tactic_empty_side(b):
        print('Used tactic_empty_side')
        return
    print('No tactic applied: error in tactic implementations')
```

9

O wins 77904 times, calculated by a similar function to the one we used to find how many times X wins:

```
def sum_o_wins(t):
    b, bs = t
    ns = wins('O', b)
    for board in bs:
        ns += sum_o_wins(board)
    return ns

o_wins = sum_o_wins(x_game_tree)
```

The number of drawn games is 46080 can be calculated similarly, counting one for each board in the tree which is full but not won by any player:

```
def drawn_games(t):
    b, bs = t
    ns = wins('X', b) and not wins('O', b) and full(b)
    for board in bs:
        ns += drawn_games(board)
    return ns

drawn = drawn_games(x_game_tree)
```

To calculate the total number of games, we can look for all boards which are full or won. This comes to 255168.

```
def num_games(t):
    b, bs = t
    ns = wins('O', b) or wins('X', b) or full(b)
    for board in bs:
        ns += num_games(board)
    return ns

games = num_games(x_game_tree)
```

Another way to find all boards which are full or won is to look for boards with no sub-trees.

Of course, we need only find two of the three outcomes of a game – we can deduce the third by subtraction from the total number of games.

10

We write a function traverses the tree, counting one for each time the function passed to it returns `True`.

```
def sum_game_tree(f, t):
    b, bs = t
    ns = f(b)
    for sb in bs:
        ns += sum_game_tree(f, sb)
    return ns
```

Now, we can write simple little functions to pass to `sum_game_tree`:

```
x_game_tree = game_tree('X')

def f(b): return wins('X', b)
x_wins = sum_game_tree(f, x_game_tree)

def f(b): return wins('O', b)
o_wins = sum_game_tree(f, x_game_tree)

def f(b): return not wins('X', b) and not wins('O', b) and full(b)
draw = sum_game_tree(f, x_game_tree)

def f(b): return wins('X', b) or wins('O', b) or full(b)
games = sum_game_tree(f, x_game_tree)
```

11

We can write the tree out using nested tuples, each consisting of three elements: the current node, the left branch and the right branch. We use `'?'` for nodes which do not correspond to a valid letter or number:

```
tree = ('?',
          ('E',
              ('I',
                  ('S',
                      ('H', '5', '4'),
                      ('V', '?', '3')),
              ('U',
                  'F',
                      ('?', '?', '2'))),
               ('A',
                   ('R', 'L', '?'),
                   ('W', 'P',
                       ('J', '?', '1')))),
           ('T',
               ('N',
                   ('D',
                       ('B', '6', '?'), 'X'),
                   ('K', 'C', 'Y')),
               ('M',
                   ('G',
                       ('Z', '7', '?'), 'Q'),
                   ('O',
                       ('?', '8', '?'), ('?', '9', '0'))))))
```

Now we need a function to look through a given code, and traverse the tree, going left for each dot and right for each dash. When we have finished, we check to see if we have a string or a tuple and extract the string if we need to.

```
def decode_morse(code):
    t = tree
    for c in code:
        if c == ' ':
            pass
        elif c == '.':
            n, l, r = t
            t = l
        else:
            n, l, r = t
            t = r
    if type(t) == tuple:
        n, l, r = t
        return n
    else:
        return t
```

Now we must write a function to split a string into individual codes, recognising seven spaces as one space in the output:

```
def split_string(string):
    codes = []                                          output codes
    spaces = 0                                  spaces in current run of spaces
    code = ''                                           current code
    for c in string:
        if c == ' ':
            if code != '' and spaces > 0:
                codes.append(code)                      completed code
                code = ''
            spaces = spaces + 1
        else:
            if spaces == 7: codes.append(' ')           word space found
            spaces = 0
            code = code + c
    if code != '': codes.append(code)
    return codes
```

Now the main function is simple: we use `split_string` to get a list of codes, including the spaces we have found, and decode each non-Space code and print it:

```
def decode_morse_string(string):
  for code in split_string(string):
      if code == ' ': print(' ', end='')
      else: print(decode_morse(code), end='')
  print('')
```

Project 4: Photo Finish

1

We will design appropriate functions for brightness and contrast, and then test them on our image.

For some combinations of brightness and contrast factors and pixel values, these functions will return values less than 0 or more than 255. We begin, then, with a function `clamp` to make sure that does not happen, and the resulting pixel value is in range:

```
def clamp(x):
    if x < 0: return 0
    elif x > 255: return 255
    else: return x
```

There is no right or wrong formula for brightness or contrast. Here, we have chosen to take brightness values expected to be generally from -2 to 2, -2 meaning very dim, 2 meaning very bright:

```
def brightness(p, x):
    r, g, b = p
    r_out = clamp(int(r + x * 128))
    g_out = clamp(int(g + x * 128))
    b_out = clamp(int(b + x * 128))
    return (r_out, g_out, b_out)
```

For contrast, we use simple multiplication. This assumes inputs of 0 upwards.

```
def contrast(p, x):
    r, g, b = p
    r_out = clamp(int(r * x))
    g_out = clamp(int(g * x))
    b_out = clamp(int(b * x))
    return (r_out, g_out, b_out)
```

Now we can define functions to perform our operation on a whole image, using the `process_pixels` function we wrote earlier:

```
def brightness_image(i, x):
    def b(p): return brightness(p, x)
    return process_pixels(b, i)

def contrast_image(i, x):
    def c(p): return contrast(p, x)
    return process_pixels(c, i)
```

Now we can use these functions with some test values for brightness and contrast, and save them.

```
bright = brightness_image(i, 0.5)
dim = brightness_image(i, -0.5)
low_contrast = contrast_image(i, 0.25)
high_contrast = contrast_image(i, 1.5)

bright.save('bright.png')
dim.save('dim.png')
low_contrast.save('low_contrast.png')
high_contrast.save('high_contrast.png')
```

Here are the results. They are, from right to left: `bright.png`, `dim.png`, `low_contrast.png`, and `high_contrast.png`:

2

To flip horizontally, we range over the left hand half of the image, swapping pixels with the equivalents on the right hand side.

```
def hflip(i):
    p = i.load()
    sx, sy = i.size
    for x in range(sx // 2):
        for y in range(sy):
           r = p[x, y]
           p[x, y] = p[sx - x - 1, y]
           p[sx - x - 1, y] = r
```

If the image has an odd width, the middle column is not touched. This is a consequence of the rounding-down behaviour of the `//` operator. A similar function can be written for the vertical flip:

```
def vflip(i):
    p = i.load()
    sx, sy = i.size
    for y in range(sy // 2):
        for x in range(sx):
           r = p[x, y]
           p[x, y] = p[x, sy - y - 1]
           p[x, sy - y - 1] = r
```

Rotation by 180 degrees is a simple combination of the two (try it with a piece of paper you have marked the corners of):

```
def rotate180(i):
    hflip(i)
    vflip(i)
```

3

The changes are simple:

```
def blur_in_place(i):
    p = i.load()
    sx, sy = i.size
    for x in range(3, sx - 3):
        for y in range(3, sy - 3):
            sumr, sumg, sumb = 0, 0, 0
            for dx in range(-1, 2):
                for dy in range(-1, 2):
                    sourcer, sourceg, sourceb = p[x + dx, y + dy]
                    sumr = sumr + sourcer
                    sumg = sumg + sourceg
                    sumb = sumb + sourceb
            p[x, y] = (int(sumr / 9), int(sumg / 9), int(sumb / 9))
```

We can test by blurring three times, just like we did with our original `blur` function. The new (right) and old (left) results are similar but not the same – the in-place blur is blurrier.

4

We need a border of width 1 for each blur operation, to avoid losing content over the edge:

```
def blur_auto(i, n):
    i = border(i, n, (255, 255, 255))
    for x in range(n):
        i = blur(i)
    return i
```

5

This is simple enough – and you could extend it to add a border too.

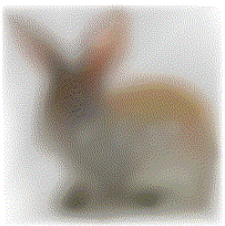
```
i = Image.open('rabbit.png')

images = [i]

for x in range(99):
    i = blur(i)
    images.append(i)

images[0].save('blur.gif', save_all=True, append_images=images[1:],
               duration=100, loop=0)
```

Here is frame 100:



Hints for Questions

Chapter 1
Starting Off

1

Try to work these out on paper, and then check by typing them in. Can you show possible steps of evaluation for each expression?

2

Type it in. What does Python print? Consider the precedence of + and $*$.

Chapter 2
Names and Functions

2

What does the function take as arguments? You can use the `!=` operator and the **`and`** keyword here.

3

The function will have three arguments i.e. **`def`** `volume(w, h, d):` ...

4

Can you define this in terms of the `is_vowel` function we have already written?

5

When does it not terminate? Can you add a check to see when it might happen, and return 0 instead? What is the factorial of 0 anyway?

6

What is the sum of all the integers from $1 \ldots 1$? Perhaps this is a good start.

7

What happens when you raise a number to the power 0? What about the power 1? What about a higher power?

8

You can use an additional argument to keep track of which number the function is going to try to divide by.

Chapter 3
Again and Again

1

Make sure to consider how the start and stop arguments are defined at the beginning of this chapter.

3

You will need a local variable to store the count.

6

The `input` function with an argument, and using the `\n` newline sequence might look like this:

```
entered =
    input('Please enter the password\n')
```

To remove the `entered` variable, we can do the `input` inside the **`while`** loop's test itself.

7

You might need three variables: the chosen secret number, the current guess, and the number of tries so far. Remember the `int` function can convert a string to an integer.

8

This is just a big **`if`** construct. Make sure not to output both a letter space and a word space at the end of a word – just a word space.

Chapter 4
Making Lists

2

Try making a fresh, empty list, and then putting items from the original list into it one by one. Maybe you could use the `insert` method to put them in a particular place.

3

What initial value can we use for keeping track of both the maximum number seen and the minimum number seen?

6

Start from a fresh, empty list. Then, looking at each element of the original list, decide whether it should go into the new list or not.

7

You can use the `setify` function you have already written to find the unique items, and then the `count` method to find out how many of each appear in the input list.

11

The rotation may be achieved by slicing.

13

The problem may be split into two. First identify "correct numbers in the correct place" then "correct numbers in incorrect place". In the latter stage, be careful not to use any position in the code identified in the first stage, nor to use a position twice.

Chapter 5
More with Lists and Strings

1

We already know how to make the list of words with `split`.

4

Just a function to remove spaces from the beginning of a list is required; the rest can be done with list reversal.

9

Remember that a list comprehension can have an **`if`** part as well as a **`for`** part.

Chapter 6
Prettier Printing

1

Remember not to add a comma or space after the final item. We did this once before, in chapter 3 question 4.

3

Recall that `print` can take multiple values.

5

How will the user signal that they have no more names to type in?

Chapter 7
Arranging Things

1

Remember that we can assign to a tuple using tuple unpacking: `a, b = ...`

2

The `items` method, described in the chapter, can be used to iterate on two variables at once with **`for`** `k, v = ...`

5

Consider the indices when deletion happens.

6

The `items` method, described in the chapter can be used to iterate on two variables at once with **`for`** `k, v = ...`

7

Remember that `set` can build a set of the letters in a string.

Chapter 8
When Things Go Wrong

2

The technique is to use map to build a list (possibly containing `None` values), then filter it to remove them.

Chapter 9
More with Files

3

What happens if `int` cannot proceed because the file is malformed? How will you know when all the entries have been read?

5

The `split` method works, of course, equally well on numbers as on words.

7

A dictionary is suitable for storing a histogram. When do we need to add a new entry? When do we need to increment an existing entry?

Chapter 10
The Other Numbers

1

Consider the two functions `math.ceil` and `math.floor`.

3

Consider the function `math.floor`. What should happen in the case of a negative number?

4

Calculate the column number for the asterisk carefully. How can it be printed in the correct column?

5

You will need to call the `star` function with an appropriate argument at points between the beginning and end of the range, as determined by the step.

Chapter 11
The Standard Library

2

The string `string.digits` is `'0123456789'`.

5

The use of `time.sleep` is to provide a count-down for the user.

Chapter 12
Building Bigger Programs

1

The two cases (a maximum is provided, and is not provided) may be distinguished by testing the length of the `sys.argv` list.

2

The `plot` function requires a function to be passed to it. We can build such a function from the argument provided on the command line, using the `eval` function.

Project 1: Pretty Pictures

3

The number of segments used to approximate a circle ought to be related to its circumference not its radius.

5

There are three dimensions to the gamut: red, green and blue. Since we cannot display these on a 2D screen, you must find a way to 'flatten' the space out.

PROJECT 1A

We need to build a function which can be called repeatedly to evaluate an expression from the command line, in terms of x. Can you write a function to return such a function?

PROJECT 1B

Recall that `turtle.Screen().tracer(0, 0)` turns off animation, and that you will need `turtle.Screen().update()` to show the clock face when you have finished drawing it, and that `time.sleep` may be used to wait for the next time we need to draw a clock.

Project 2: Counting Calories

1

Remember that the `items` method may be used to iterate over the keys and values of the table resulting from `table_of_file`.

2

The dates will sort as if they were in alphabetical order – the digits too are ordered.

3

If a table lookup fails, `None` is returned.

Project 3: Noughts and Crosses

3

The `random_move` function will need a mechanism to pick a random blank space. You might repeatedly pick spaces until one is found to be blank, though this may be inefficient when few spaces remain.

Another way would be to pick a random place to start, and then cycle through the positions in turn until a blank one is found.

6

Our `try_to_take` function is useful here.

8

Draw out diagrams of the situations described by the rules. You will need a list of intersecting lines and the points at which they intersect.

9

To calculate the total number of boards, remember only to count ones which are full or won.

10

The function must traverse the whole tree, counting once for each board for which the function passed to it returns `True`.

11

The tree may be represented by nested tuples of three items each, representing the data, the left branch and the right branch.

Project 4: Photo Finish

1

There is no right or wrong answer for the formulae for brightness and contrast. Design them to produce a sensible result for a sensible input.

2

Rotation may be achieved by a combination of flips.

4

Consider how the blurring operation spreads the colour of a pixel around: how do we make sure we lose none?

Index