`readlines` returns all of the remaining lines as a list of strings:

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence `012`.

At the end of the file, `readline` returns the empty string and `readlines` returns the empty list:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

The following is an example of a line-processing program. `filterFile` makes a copy of `oldFile`, omitting any lines that begin with `#`:

```
def filterFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.readline()
    if text == "":
      break
    if text[0] == '#':
      continue
    f2.write(text)
  f1.close()
  f2.close()
  return
```

The `continue` statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.

Thus, if `text` is the empty string, the loop exits. If the first character of `text` is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy `text` into the new file.

11.2 Writing variables

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings first. The easiest way to do that is with

the `str` function:

```
>>> x = 52
>>> f.write (str(x))
```

An alternative is to use the **format operator** `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string.

As a simple example, the **format sequence** `"%d"` means that the first expression in the tuple should be formatted as an integer. Here the letter *d* stands for "decimal":

```
>>> cars = 52
>>> "%d" % cars
'52'
```

The result is the string `'52'`, which is not to be confused with the integer value `52`.

A format sequence can appear anywhere in the format string, so we can embed a value in a sentence:

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
'In July we sold 52 cars.'
```

The format sequence `"%f"` formats the next item in the tuple as a floating-point number, and `"%s"` formats the next item as a string:

```
>>> "In %d days we made %f million %s." % (34,6.1,'dollars')
'In 34 days we made 6.100000 million dollars.'
```

By default, the floating-point format prints six decimal places.

The number of expressions in the tuple has to match the number of format sequences in the string. Also, the types of the expressions have to match the format sequences:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

In the first example, there aren't enough expressions; in the second, the expression is the wrong type.

For more control over the format of numbers, we can specify the number of digits as part of the format sequence:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

The number after the percent sign is the minimum number of spaces the number will take up. If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:

```
>>> "%-6d" % 62
'62    '
```

For floating-point numbers, we can also specify the number of digits after the decimal point:

```
>>> "%12.2f" % 6.1
'        6.10'
```

In this example, the result takes up twelve spaces and includes two digits after the decimal. This format is useful for printing dollar amounts with the decimal points aligned.

For example, imagine a dictionary that contains student names as keys and hourly wages as values. Here is a function that prints the contents of the dictionary as a formatted report:

```
def report (wages) :
  students = wages.keys()
  students.sort()
  for student in students :
    print "%-20s %12.2f" % (student, wages[student])
```

To test this function, we'll create a small dictionary and print the contents:

```
>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}
>>> report (wages)
joe                          5.45
joshua                       4.25
mary                         6.23
```

By controlling the width of each value, we guarantee that the columns will line up, as long as the names contain fewer than twenty-one characters and the wages are less than one billion dollars an hour.