Chapter 9

Strings

You are probably accustomed to referring to a collection of readable characters as "text." By "readable characters," we mean the symbols that one typically finds on a keyboard such as letters of the alphabet, digits zero through nine, and punctuation marks (including a blank space). As we will soon see, computers use a mapping between characters and numbers: each character has a corresponding unique numeric value. With this mapping in place, we can interpret (and display) the ones and zeros stored in the computer as a character or we can interpret (and display) them as a number. In either case, we are talking about the same combination of ones and zeros. The only thing that differs is whether we decide to interpret the combination as a number or, using the established mapping, as a character.

The first standardized mapping created for numbers and characters was the American Standard Code for Information Interchange (ASCII) which is still in use today. ASCII specifies the numeric values corresponding to 128 characters and we will focus our attention on these 128 characters.

If text is stored in a computer using only ASCII characters, we refer to this as "plain text." So, for example, a Word document is *not* a plain text file. A Word document contains information that goes far beyond the text you ultimately read in the rendered document. For example, a Word document contains information pertaining to the page layout, the fonts to be used, and the history of previous edits. On the other hand, a plain-text file specifies none of this. The Python files you create with IDLE (i.e., the `.py` files) are plain-text files. They contain the ASCII characters corresponding to the code and nothing more. Essentially every byte of information in the file is "visible" when you open the file in IDLE (this "visibility" may be in the form of space or a new line). The file contains no information about fonts or page size or anything other than the code itself.

Many cutting-edge problems involve the manipulation of plain text. For example, Web pages typically consist of plain text (i.e., hyper-text markup language [HTML]) commands. The headers of email messages are in plain text. Data files often are written in the form of plain text. XML (extensible markup language) files are written in plain text. Thus, the ability to manipulate plain text is a surprisingly important skill when solving problems in numerous disciplines.

As you have seen, in Python we store a collection of characters in a data type known as a string, i.e., a `str`. When we say "text" we will typically be using it in the usual sense of the word to refer to readable content. On the other hand, when we say "string" we will be referring to the

From the file: `strings.tex`

Python `str` data type which consists of a collection of methods and attributes (and the attributes include readable content, i.e., text). In this chapter we will review some of the points we have already learned about strings and introduce several new features of strings and tools for working with them.

9.1 String Basics

The following summarizes several string-related features or tools that we discussed previously:

- A string literal is enclosed in either single quotes, double quotes, or either quotation mark repeated three times. When the quotation mark is repeated three times, the string may span multiple lines.
- Strings are immutable.
- Strings can be concatenated using the plus operator.
- A string can be repeated by multiplying it by an integer.
- The `str()` function converts its argument to a string.
- A string is a sequence. Thus:
  - A string can be used as the iterable in the header of a `for`-loop (in which case the loop variable takes on the values of the individual characters of the string).
  - The `len()` function returns the length of a string, i.e., the number of characters.
  - An individual character in a string can be accessed by specifying the character's index within brackets.
  - Slicing can be used to obtain a portion (or all) of a string.

Listing 9.1 demonstrates several basic aspects of strings and string literals. Note that in line 5 triple quotes are used for a string literal that spans two lines. In this case the newline character becomes part of the string. In contrast to this, in line 13 a string is enclosed in single quotes and the string itself is written across two lines. To do this, the newline character is escaped (i.e., the backslash is entered immediately before typing return). In this case `s3` is a single-line string—the newline character is not embedded in the string. In line 21 a string is created that includes the hash symbol (`#`). When the hash symbol appears in a string, it becomes part of the string and does not specify the existence of a comment.

Listing 9.1 Demonstration of several basic string operations including the different ways in which a literal can be quoted.

```
1 >>> s1 = 'This is a string.'
2 >>> print(s1)
3 This is a string.
4 >>> # Concatenation of string literals quoted in different ways.
```

```
>>> s2 = "This" + 'is' + """also""" + '''a
... string.'''
>>> s2          # Check contents of s2.
'Thisisalsoa\nstring.'
>>> print(s2) # Printed version of s2.
Thisisalsoa
string.
>>> # Single-line string written across multiple lines.
>>> s3 = "this is a \
... single line"
>>> print(s3)
this is a single line
>>> s4 = "Why? " * 3    # String repetition.
>>> print(s4)
Why? Why? Why?
>>> # Within a string the hash symbol does not indicate a comment.
>>> s5 = "this is # not a comment"
>>> print(s5)
this is # not a comment
```

Listing 9.2 demonstrates the conversion of an integer, a `float`, and a `list` to strings using the `str()` function. In Python every object has a `__str__()` method that is used, for instance, by `print()` to determine how to display that object. When an object is passed to the `str()` function, this simply serves to call the `__str__()` method for that object. So, when we write `str(<object>)` we are essentially asking for that object's string representation (as would be used in a `print()` statement). In line 5 in Listing 9.2 each of the objects is converted to a string and concatenated with the other strings to make a rather messy string. Note that in line 5 the string `s0` is passed as the argument to the `str()` function. Because `s0` is already a string, this is unnecessary, but it is done here to illustrate it is not an error to pass a string to `str()`.

Listing 9.2 Demonstration of the use of the `str()` function to convert objects to strings.

```
>>> x = 10
>>> y = 23.4
>>> zlist = [1, 'a', [2, 3]]
>>> s0 = "Hi!"
>>> all = str(x) + str(y) + str(zlist) + str(s0)
>>> print(all)
1023.4[1, 'a', [2, 3]]Hi!
```

Listing 9.3 provides code that demonstrates that strings are immutable sequences. A string is created in line 1 that has embedded quotes. If the embedded quotes are distinct from the surrounding quotes, this does not pose a difficulty.[1] In line 10 an attempt is made to change the second

[1] To embed a quotation mark of the same type as the surrounding quotation marks requires the embedded quotation mark to be escaped as discussed in Sec. 9.3.

character of the string from an uppercase `U` to a lowercase `u`. This fails because of the immutability of strings. The remaining code illustrates two different constructs for looping over the characters of a string. Because the `print()` statement in line 16 has `end` set to a hyphen, the subsequent interactive prompt appears on the same line as the output (as shown in line 18). In this particular session the user typed return to obtain another prompt on a new line (line 19), but a command could have been entered on line 18. The `for`-loop in lines 19 and 20 has the loop variable vary between zero and one less than the length of the string. The `print()` statement in line 20 uses the negative of this variable as the index for a character of the string. Since zero is neither positive nor negative, `-0`, `+0`, and `0` are all equivalent. Thus, the first character produced by the loop (line 22) is the first character of the string. The subsequent output (lines 23 through 27) progresses from the back of the string toward the front. Although not shown here, strings can be used with the `enumerate()` function so that characters and their corresponding indices can be obtained simultaneously in the header of a `for`-loop.

Listing 9.3 Demonstration that strings are immutable sequences.

```
1  >>> s1 = '"Unbroken" by Laura Hillenbrand'
2  >>> print(s1)
3  "Unbroken" by Laura Hillenbrand
4  >>> len(s1)       # Number of characters.
5  32
6  >>> s1[16 : 20]    # Slice of s1.
7  'aura'
8  >>> s1[1]         # Second character of s1.
9  'U'
10 >>> s1[1] = 'u'  # Attempt to change s1.
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: 'str' object does not support item assignment
14 >>> s2 = "Loopy!"
15 >>> for ch in s2:
16 ...     print(ch, end="-")
17 ...
18 L-o-o-p-y-!->>>
19 >>> for i in range(len(s2)):
20 ...     print(-i, s2[-i])
21 ...
22 0 L
23 -1 !
24 -2 y
25 -3 p
26 -4 o
27 -5 o
```

9.2 The ASCII Characters

As discussed in Chap. 1 and in the introduction to this chapter, all data are stored as collections of ones and zeros. This is true of the characters of a string as well. In the early 1960's a standard was published, known as the American Standard Code for Information Interchange (ASCII, pronounced "ask-ee"), which specifies a mapping of binary values to characters. The word "code" is not meant to imply anything having to do with encryption. Rather, this is a mapping of numeric values to 128 characters. These characters, together with their ASCII values, are shown in Listing 9.4. This character set consists of both non-printable characters and graphic (or printable) characters. The non-printable characters are indicated by a two- or three-letter abbreviation and these are often identified as "control characters." Some of the non-printable characters are discussed following the listing and the remainder are listed in an appendix.

Listing 9.4 The 128 ASCII characters together with their corresponding ASCII value. Non-printable characters are listed by their two- or three-letter abbreviations. The space character (32) is indicated by a blank space.

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL
8 BS	9 HT	10 LF	11 VT	12 FF	13 CR	14 SO	15 SI
16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB
24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
32	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 DEL

The ASCII character set is obviously limited in that it was not designed to handle characters from other languages. Newer codes exist, such as Unicode, that attempt to accommodate all languages. The Unicode standard currently defines more than 110,000 characters! Although Python provides support for Unicode, we will use ASCII almost exclusively. One thing to be aware of is that the larger codes invariably include ASCII as a subset. Thus, everything we will consider here remains true in the larger character sets.

ASCII is also somewhat dated in that it was designed to accommodate hardware that existed in the early 1960's and yet it did not precisely specify the behavior associated with some of the non-printable characters. This led to some confusion and incompatibilities. For example, some

manufacturers interpreted Line Feed (10) to indicate that output should continue at the beginning of the next line. However, other manufacturers required a Carriage Return (13) preceding a Line Feed to indicate that output should continue at the beginning of the next line. In Python when we say a "newline character" or simply "newline," we mean the ASCII Line Feed character (10) and take this, by itself, to mean output should continue on the beginning of the next line.

The distinction between graphic (printable) and non-printable characters is somewhat fuzzy. For example, Horizontal Tab (9) is considered a non-printable character while the space character (32) is classified as a "non-printing graphic character." However the Horizontal Tab typically produces between one and eight spaces. So, why consider Horizontal Tab a non-printable character and the space character a graphic character? Also, although Line Feed doesn't produce ink on the page, so to speak, it does affect the appearance of the output. So, why not also consider it a non-printing graphic character? But these types of question do not truly concern us so we will simply accept that there are 33 non-printable characters, 95 graphic characters, and 128 total characters in the ASCII character set.

Because there are 128 total ASCII characters, all the characters can be represented using seven bits, i.e., $2^7 = 128$. However, characters are almost invariably stored using eight bits, otherwise known as one byte.[2]

9.3 Escape Sequences

Escape sequences were introduced in Sec. 2.5 where we saw we could write `\n` to include a newline character within a string. The backslash is used to indicate that the character following it does not have the usual meaning we associate with it within a string context (sometimes the backslash escapes multiple characters as will be shown). Given the description above of ASCII characters, we know the newline character (LF with a value of 10) is really no more than a binary value corresponding to the decimal number 10. To represent a newline character we write `\n`. Put another way, `\n` serves to place the ASCII Line Feed (LF) character in a string (i.e., the numeric equivalent of 10 is placed in the string). This escape sequence is used by Python and most other major computer languages.

As listed in Listing 9.5, Python provides escape sequences for eight of the ASCII characters. It is unlikely you will ever need Null (`\0`), Backspace (`\b`), Vertical Tab (`\v`), Form Feed (`\f`), or Carriage Return (`\r`). On the other hand, Bell (`\a`) can be useful. It typically causes the computer to "beep" or sound some tone.[3] Horizontal Tab (`\t`) and Line Feed (`\n`) are used frequently and we often refer to them simply as tab and newline, respectively.

Listing 9.5 ASCII characters for which Python provides a two-character escape sequence.

---

[2] In the early days of computers the eighth bit was sometimes used as a *parity bit*. A parity bit can be used to check whether an error has occurred in the storage or transmission of data. For example, the value of the bit can ensure the total number of bits in the byte that are set to one sums to an even number. To do so, the parity bit is zero if the other bits sum to an even number and is one if the other bits sum to an odd number. A parity bit cannot be used to directly correct an error, but it can signal the occurrence of an error and then appropriate steps can be taken.

[3] However, if you try to print `\a` from within IDLE, generally this will *not* succeed in sounding the system "bell."

ASCII Value	Abbrv.	Escape Sequence	Description
0	NUL	`\0`	Null character
7	BEL	`\a`	Bell (alert)
8	BS	`\b`	Backspace
9	HT	`\t`	Horizontal Tab (tab)
10	LF	`\n`	Line Feed (newline)
11	VT	`\v`	Vertical Tab
12	FF	`\f`	Form Feed
13	CR	`\r`	Carriage Return

You will often hear the term "whitespace" mentioned in connection with strings. Collectively, whitespace is any character that introduces space into a document (without printing anything). Thus, in addition to the space character, Horizontal Tab (tab), Line Feed (newline), Vertical Tab, Form Feed, and Carriage Return are all considered whitespace (ASCII values 9 through 13).

There are other escape sequences in Python that aren't directly associated with ASCII characters. These are listed in Listing 9.6. Several of these are not of direct concern but are given for the sake of completeness and with the understanding that you may encounter them even if you don't use them yourself (we will see an example of this in a moment).

Listing 9.6 Escape sequences that aren't explicitly related to ASCII characters. In the first escape sequence, `\<newline>` means a backslash followed by the newline character that has been entered by hitting the enter (or return) key. For the other escape sequences, the terms between angle brackets are place-holders for an appropriate value. For example, `\N{registered sign}` produces ® while `\x41` produces the character `A` since the decimal number 65 is written 41 in hexadecimal.

Escape Sequence	Description
`\<newline>`	Ignore the following newline character
`\\`	Literal backslash (`\`)
`\'`	Literal single quote (`'`)
`\"`	Literal double quote (`"`)
`\N{<name>}`	Unicode character *<name>*
`\<ooo>`	Character with octal value *<ooo>*
`\u<hhhh>`	Character with 16-bit hexadecimal value *<hhhh>*
`\U<hhhhhhhh>`	Character with 32-bit hexadecimal value *<hhhhhhhh>*
`\x<hh>`	Character with 8-bit hexadecimal value *<hh>*

Listing 9.7 provides several examples of escape sequences. The string created in line 1 uses two backslashes to specify that there is a single backslash in the string. This string also includes the escape sequence for a newline character. The code following this demonstrates various ways of handling a string with embedded quotation marks. One way to handle an embedded quote is by enclosing the string in triple quotes. This is demonstrated in line 16 where the string is surrounded by a single quotation mark repeated three times. Python is not confused by the embedded single quote. However, we see on line 22 that this approach falls apart if we try to repeat the double

quotation mark three times. The problem is that the trailing edge of the string is a double quote. Python attempts to end the string with the first three double quotes it encounters on the right side of the string. But, this leaves the last double quote unaccounted for.

Listing 9.7 Demonstration of escape sequences for backslash and embedded quotes.

```
>>> s0 = "\\ and\nburn"
>>> print(s0)
\ and
burn
>>> s1 = 'I didn\'t know he said, "Know!"'
>>> print(s)
I didn't know he said, "Know!"
>>> s1
'I didn\'t know he said, "Know!"'
>>> s2 = "I didn't know he said, \"Know!\""
>>> print(s2)
I didn't know he said, "Know!"
>>> s2
'I didn\'t know he said, "Know!"'
>>> # Triple quotes can do away with need to escape embedded quotations.
>>> s3 = '''Now, I didn't know he said, "Know!"'''
>>> print(s3)
Now, I didn't know he said, "Know!"
>>> # Tripling the double quotation mark does not work for this string
>>> # because the desired string itself is terminated by a double
>>> # quotation mark.
>>> s4 = """I didn't know he said, "Know!""""
  File "<stdin>", line 1
    s4 = """I didn't know he said, "Know!""""
                                           ^
SyntaxError: EOL while scanning string literal
```

Listing 9.8 demonstrates the embedding of the Null and Bell characters in a string (the embedding of Nulls is not something that is typically done, but the embedding of Bell characters is useful). In line 2 the variable `koan` is assigned to a string that contains four Null characters.[4] When this is printed, the Null characters do not appear (it is as if they are not in the string). However, the length of the string, as shown in lines 5 and 6, does reflect the fact that the string contains Null characters (the Null characters each occupy one byte of computer memory). Note that each Null character counts as a single character (and is stored as a single byte) despite the fact that we represent it in a string as the two-character escape sequence `\0`. The expression in line 9 serves to echo the contents of `koan`. The output in line 10 displays the Null characters as the escape sequence `\x00`. This is a two-digit hexadecimal (base 16) representation of the value. Thus, even though we wrote `\0`, Python displays this as `\x00`. A string with three Bell characters is created

[4] In some computer languages, such as C, a Null is used to indicate the termination of a string. This is not the case in Python.

in line 11. When this is printed, these characters do not appear visually, but they should appear aurally as a sounding of the computer's "bell." When we display the contents of this string, as done in line 14, Python again displays the characters in their hexadecimal form despite the fact that we entered the characters as `\a`.

Listing 9.8 Embedding Null and Bell in a string and demonstration that Python displays these as two-digit hexadecimal values.

```
1  >>> # Create a string with four embedded Null characters.
2  >>> koan = "I contain\0\0\0\0 nothing."
3  >>> print(koan)  # Null characters do not appear in output.
4  I contain nothing.
5  >>> len(koan)    # Each Null counts as one character.
6  22
7  >>> # Python represents the Null character by its two-digit
8  >>> # hexadecimal value.
9  >>> koan
10 'I contain\x00\x00\x00\x00 nothing.'
11 >>> alert = "Wake Up!\a\a\a"  # String with Bell characters.
12 >>> print(alert)              # Will sound computer's "bell."
13 Wake Up!
14 >>> alert                     # Internal representation of string.
15 'Wake Up!\x07\x07\x07'
16 >>>
```

9.4 chr() and ord()

The built-in function `chr()` takes an integer argument and produces the corresponding character. The built-in function `ord()` is effectively the inverse of `chr()`. `ord()` takes a string of length one, i.e., a single character, and returns the corresponding ASCII value. The code in Listing 9.9 illustrates the behavior of these functions.

Listing 9.9 Demonstration of the use of `ord()` and `chr()`.

```
1  >>> ord('A'), ord('B'), ord('!'), ord(' ')
2  (65, 66, 33, 32)
3  >>> chr(65), chr(66), chr(33), chr(32)
4  ('A', 'B', '!', ' ')
5  >>> ord('Z'), chr(90) # 90 and Z are ASCII pairs.
6  (90, 'Z')
7  >>> ord(chr(90))      # ord(chr()) returns original argument.
8  90
9  >>> chr(ord('Z'))     # chr(ord()) returns original argument.
10 'Z'
```

```
11 >>> for i in range(65, 75):
12 ...     print(chr(i), end="")
13 ...
14 ABCDEFGHIJ>>>
15 >>> for ch in "ASCII = numbers":
16 ...     print(ord(ch), end=" ")
17 ...
18 65 83 67 73 73 32 61 32 110 117 109 98 101 114 115 >>>
```

In line 1 `ord()` is used to obtain the numeric values for four characters. Comparing the result in line 2 to the values given in Listing 9.4, we see these are indeed the appropriate ASCII values. In line 3 the numeric values produced by line 1 (i.e., the values on line 2) are used as the arguments of the `chr()` function. The output in line 4 corresponds to the characters used in line 1. The statements in lines 7 and 9 also confirm that `ord()` and `chr()` are inverses (i.e., one does the opposite of the other); each result is identical to the value provided as the argument of the inner function.

The `for`-loop in lines 11 and 12 sets the loop variable `i` to values between 65 and 74 (inclusive). Each value is used as the argument of `chr()` to produce the characters `ABCDEFGHIJ`, i.e., the first ten characters of the alphabet as shown in line 8 (the interactive prompt also appears on this line since the `end` parameter of the `print()` statement is set to the empty string). The `for`-loop in lines 15 and 16 sets the loop variable `ch` to the characters of the string that appears in the header. This variable is used as the argument of `ord()` to display the corresponding ASCII value for each character. The output is shown in line 18.

We have seen that we cannot add integers to strings. But, of course, we can add integers to integers and thus we can add integers to ASCII values. Note there is an offset of 32 between the ASCII values for uppercase and lowercase letters. Thus, if we have a string consisting of all uppercase letters we can easily convert it to lowercase by adding 32 to each letter's ASCII value and then converting that back to a character. This is demonstrated in Listing 9.10 where, in line 1, we start with an uppercase string. Line 2 is used to display the ASCII value of the first character. Lines 4 and 5 show the integer value obtained by adding 32 to the first character's ASCII value. Then, in lines 6 and 7, we see that by converting this offset value back to a character, we obtain the lowercase equivalent of the first character. Line 8 initializes the variable `soft` to the empty string. This variable will serve as a string accumulator with which we build the lowercase version of the uppercase string. For each iteration of the `for`-loop shown in lines 9 through 11 we concatenate an additional character to this accumulator. The `print()` statement in line 11 shows the accumulator for each iteration of the loop (and after the concatenation). The output in lines 13 through 18 show that we do indeed obtain the lowercase equivalent of the original string.

Listing 9.10 Demonstration of the use of an integer offset to convert one character to another.

```
1 >>> loud = "SCREAM"          # All uppercase.
2 >>> ord(loud[0])             # ord() of first character.
3 83
4 >>> ord(loud[0]) + 32        # ord() of first character plus 32.
5 115
```

```
>>> chr(ord(loud[0]) + 32) # Lowercase version of first character.
's'
>>> soft = ""               # Empty string accumulator
>>> for ch in loud:
...     soft = soft + chr(ord(ch) + 32) # Concatenate to accumulator.
...     print(soft)         # Show value of accumulator.
...
s
sc
scr
scre
screa
scream
```

More interesting, perhaps, is when we add some other offset to the ASCII values of a string. If we make the offset somewhat arbitrary, the original string may start as perfectly readable text, but the resulting string will probably end up looking like nonsense. However, if we can later remove the offset that turned things into nonsense, we can get back the original readable text. This type of modification and reconstruction of strings is, in fact, the basis for encryption and decryption!

As an example of simple encryption, consider the code shown in Listing 9.11. We start, in line 1, with the unencrypted text `CLEARLY`, which we identify as the "clear text" and store as the variable `clear_text`. Line two contains a list of randomly chosen offsets that we store in the list `keys`. The number of offsets in this list corresponds to the number of characters in the clear text, but this is not strictly necessary—there should be at least as many offsets as characters in the string, but there can be more offsets (they simply won't be used). In line 3 we initialize the accumulator `cipher_text` to the empty string. The `for`-loop in lines 4 through 6 "zips" together the offsets and the character of clear text, i.e., for each iteration of the loop the loop variables `offset` and `ch` correspond to an offset from `keys` and a character from `clear_text`. Line 5, the first line in the body of the `for`-loop, adds the offset to the ASCII value of the character, concatenates this to the accumulator, and stores the result back in the accumulator. The `print()` statement in line 6 shows the progression of the calculation. The final result, shown in line 16, is a string that has no resemblance to the original clear text.

Listing 9.11 Demonstration of encryption by adding arbitrary offsets to the ASCII values of the characters of clear text.

```
>>> clear_text = "CLEARLY"             # Initial clear text.
>>> keys = [5, 8, 27, 45, 17, 31, 5]   # Offsets to be added.
>>> cipher_text = ""
>>> for offset, ch in zip(keys, clear_text):
...     cipher_text = cipher_text + chr(ord(ch) + offset)
...     print(ch, offset, cipher_text)
...
C 5 H
L 8 HT
E 27 HT`
```

```
A 45 HT`n
R 17 HT`nc
L 31 HT`nck
Y 5 HT`nck^
>>> print(cipher_text)                    # Display final cipher text.
HT`nck^
```

Now, let's go the other way: let's start with the encrypted text, also known as the cipher text, and try to reconstruct the clear text. We must have the same keys in order to subtract the offsets. The code in Listing 9.12 illustrates how this can be done. This code is remarkably similar to the code in Listing 9.11. In fact, other than the change of variable names and the different starting string, the only difference is in the body of the `for`-loop where we subtract the offset rather than add it. Note that in line 1 we start with cipher text. Given the keys/offsets used to construct this cipher text, we are able to recreate the clear text as shown in line 16.

Listing 9.12 Code to decrypt the string that was encrypted in Listing 9.11.

```
>>> cipher_text = "HT`nck^"               # Initial cipher text.
>>> keys = [5, 8, 27, 45, 17, 31, 5]      # Offsets to be subtracted.
>>> clear_text = ""
>>> for offset, ch in zip(keys, cipher_text):
...       clear_text = clear_text + chr(ord(ch) - offset)
...       print(ch, offset, clear_text)
...
H 5 C
T 8 CL
` 27 CLE
n 45 CLEA
c 17 CLEAR
k 31 CLEARL
^ 5 CLEARLY
>>> print(clear_text)                     # Display final clear text.
CLEARLY
```

Let's continue to explore encryption a bit further. Obviously there is a shortcoming to this exchange of information in that both the sender and the receiver have to have the same keys. How do they exchange the keys? If a third party (i.e., an eavesdropper) were able to gain access to these keys, then they could easily decipher the cipher text, too. So exchanging (and protecting) keys can certainly be a problem. Suppose the parties exchanging information work out a system in which they generate keys "on the fly" and "in the open." Say, for instance, they agree to generate keys based on the first line of the third story published on National Public Radio's Morning Edition Web site each day. This information is there for all to see, but who would think to use this as the basis for generating keys? But keep in mind that the characters in the story, once converted to ASCII, are just collections of integers. One can use as many characters as needed from the story to encrypt a string (if the string doesn't have more characters than the story).

We'll demonstrate this in a moment, but let's first consider a couple of building blocks. We have previously used the `zip()` function which pairs the elements of two sequences. This function can be called with sequences of different lengths. When the sequences are of different lengths, the pairing is only as long as the shorter sequence. Let's assume the clear text consists only of spaces and uppercase letters (we can easily remove this restriction later, but it does make the implementation simpler). When we add the offset, we want to ensure we still have a valid ASCII value. Given that the largest ASCII value for the clear text can be 90 (corresponding to `Z`) and the largest possible ASCII value for a printable character is 126, we cannot use an offset larger than 36. If we take an integer modulo 37, we obtain a result between 0 and 36. Given this, consider the code in Listing 9.13 which illustrates the behavior of `zip()` and shows how to obtain an integer offset between 0 and 36 from a line of text.[5] The code is discussed below the listing.

Listing 9.13 Demonstration of `zip()` with sequences of different lengths. The `for`-loop in lines 5 and 6 converts a string to a collection of integer values. These values can be used as the "offsets" in an encryption scheme.

```
1  >>> keys = "The 19th century psychologist William James once said"
2  >>> s = "Short"
3  >>> list(zip(keys, s))
4  [('T', 'S'), ('h', 'h'), ('e', 'o'), (' ', 'r'), ('1', 't')]
5  >>> for ch in keys:
6  ...     print(ord(ch) % 37, end=" ")
7  ...
8  10 30 27 32 12 20 5 30 32 25 27 36 5 6 3 10 32 1 4 10 25 30 0 34 0 29
9  31 4 5 32 13 31 34 34 31 23 35 32 0 23 35 27 4 32 0 36 25 27 32 4
10 23 31 26
```

In line 1 the variable `keys` is assigned a "long" string. In line 2 the variable `s` is assigned a short string. In line 3 we zip these two strings together. The resulting list, shown in line 4, has only as many pairings as the shorter string, i.e., 5 elements. The `for`-loop in lines 5 and 6 loops over all the characters in `keys`. The ASCII value for each character is obtained using `ord()`. This value is taken modulo 37 to obtain a value between 0 and 36. These values are perfectly suited to be used as offsets for clear text that consists solely of uppercase letters (as well as whitespace).

Now let's use this approach to convert clear text to cipher text. Listing 9.14 starts with the clear text in line 1. We need to ensure that the string we use to generate the keys is at least this long. Line 2 shows the "key-generating string" that has been agreed upon by the sender and receiver. The `print()` statement in the body of the `for`-loop is not necessary but is used to show how each individual character of clear text is converted to a character of cipher text. Line 23 shows the resulting cipher text. Obviously this appears to be indecipherable nonsense to the casual observer. However, to the person with the key-generating string, it's meaningful!

Listing 9.14 Conversion of clear text to cipher text using offsets that come from a separate string (that the sender and receiver agree upon).

[5] This line came from **www.npr.org/templates/transcript/transcript.php?storyId=147296743**

```
>>> clear_text = "ONE IF BY LAND"
>>> keys = "The 19th century psychologist William James once said"
>>> cipher_text = ""
>>> for kch, ch in zip(keys, clear_text):
...     cipher_text = cipher_text + chr(ord(ch) + ord(kch) % 37)
...     print(kch, ch, cipher_text)
...
T O Y
h N Yl
e E Yl`
    Yl`@
1 I Yl`@U
9 F Yl`@UZ
t   Yl`@UZ%
h B Yl`@UZ%`
  Y Yl`@UZ%`y
c   Yl`@UZ%`y9
e L Yl`@UZ%`y9g
n A Yl`@UZ%`y9ge
t N Yl`@UZ%`y9geS
u D Yl`@UZ%`y9geSJ
>>> print(cipher_text)
Yl`@UZ%`y9geSJ
```

Listing 9.15 demonstrates how we can start from the cipher text and reconstruct the clear text. Again, this code parallels the code used to create the cipher text. The only difference is that we start with a different string (i.e., we're starting with the cipher text) and, instead of adding offsets, we subtract offsets. For the sake of brevity, the `print()` statement has been removed from the body of the `for`-loop. Nevertheless, lines 7 and 8 confirm that the clear text has been successfully reconstructed.

Listing 9.15 Reconstruction of the clear text from Listing 9.14 from the cipher text.

```
>>> cipher_text = "Yl`@UZ%`y9geSJ"
>>> keys = "The 19th century psychologist William James once said"
>>> clear_text = ""
>>> for kch, ch in zip(keys, cipher_text):
...     clear_text = clear_text + chr(ord(ch) - ord(kch) % 37)
...
>>> print(clear_text)
ONE IF BY LAND
```

9.5 Assorted String Methods

String objects provide several methods. These can be listed using the `dir()` function with an argument of a string literal, a string variable, or even the `str()` function (with or without the parentheses). So, for example, `dir("")` and `dir(str)` both list all the methods available for strings. This listing is shown in Listing 5.5 and not repeated here. To obtain help on a particular method, don't forget that `help()` is available. For example, to obtain help on the `count()` method, one could type `help("".count)`. In this section we will consider a few of the simpler string methods.

Something to keep in mind with all string methods is that they never change the value of a string. Indeed, they *cannot* change it because strings are immutable. If we want to modify the string associated with a particular identifier, we have to create a new string (perhaps using one or more string methods) and then assign this new string back to the original identifier. This is demonstrated in the examples below.

9.5.1 `lower()`, `upper()`, `capitalize()`, `title()`, and `swapcase()`

Methods dealing with case return a new string with the case of the original string appropriately modified. It is unlikely that you will have much need for the `swapcase()` method. `title()` and `capitalize()` can be useful at times, but the most useful case-related methods are `lower()` and `upper()`. These last two methods are used frequently to implement code that yields the same result independent of the case of the input, i.e., these methods are used to make the code "insensitive" to case. For example, if input may be in either uppercase or lowercase, or even a mix of cases, and yet we want to code to do the same thing regardless of case, we can use `lower()` to ensure the resulting input string is in lowercase and then process the string accordingly. (Or, similarly, we could use `upper()` to ensure the resulting input string is in uppercase.) These five methods are demonstrated in Listing 9.16. The `print()` statement in line 12 and the subsequent output in line 13 show that the value of the string `s` has been unchanged by the calls to the various methods. To change the string associated with `s` we have to assign a new value to it as is done in line 14 where `s` is reset to the lowercase version of the original string. The `print()` statement in line 15 confirms that `s` has changed.

Listing 9.16 Demonstration of the methods used to establish the case of a string.

```
1  >>> s = "Is this IT!?"
2  >>> s.capitalize() # Capitalize only first letter of entire string.
3  'Is this it!?'
4  >>> s.title()      # Capitalize first letter of each word.
5  'Is This It!?'
6  >>> s.swapcase()   # Swap uppercase and lowercase.
7  'iS THIS it!?'
8  >>> s.upper()      # All uppercase.
9  'IS THIS IT!?'
10 >>> s.lower()      # All lowercase.
11 'is this it!?'
```

```
>>> print(s)       # Show original string s unchanged.
Is this IT!?
>>> s = s.lower() # Assign new value to s.
>>> print(s)       # Show that s now all lowercase.
is this it!?
```

9.5.2 count()

The count() method returns the number of "non-overlapping" occurrences of a substring within a given string. By non-overlapping we mean that a character cannot be counted twice. So, for example, the number of times the (sub)string zz occurs in zzz is once. (If overlapping were allowed, the middle z could serve as the end of one zz and the start of a second zz and thus the count would be two. But this is *not* what is done.) The matching is case sensitive. Listing 9.17 demonstrates the use of count(). Please see the comments within the listing.

Listing 9.17 Demonstration of the count() method.

```
>>> s = "I think, therefore I am."
>>> s.count("I")    # Check number of uppercase I's in s.
2
>>> s.count("i")    # Check number of lowercase i's in s.
1
>>> s.count("re")   # Check number of re's in s.
2
>>> s.count("you") # Unmatched substrings result in 0.
0
```

9.5.3 strip(), lstrip(), and rstrip()

strip(), lstrip(), and rstrip() remove leading and/or trailing whitespace from a string. lstrip() removes leading whitespace (i.e., removes it from the left of the string), rstrip() removes trailing whitespace (i.e., removes it from the right of the string), and strip() removes both leading and trailing whitespace. As we will see in Sec. 10.1.1, when reading lines of text from a file, the newline character terminating the line is considered part of the text. Such terminating whitespace is often unwanted and can be conveniently removed with strip() or rstrip(). Listing 9.18 demonstrates the behavior of these methods. Please read the comments within the listing.

Listing 9.18 Demonstration of the strip(), lstrip(), and rstrip() methods.

```
>>> # Create a string with leading, trailing, and embedded whitespace.
>>> s = "   Indent a line\n   and another\nbut not this.  \n"
>>> print(s)          # Print original string.
```

```
4     Indent a line
5     and another
6  but not this.
7
8  >>> print(s.lstrip()) # Remove leading whitespace.
9  Indent a line
10    and another
11 but not this.
12
13 >>> print(s.rstrip()) # Remove trailing whitespace.
14    Indent a line
15    and another
16 but not this.
17 >>> print(s.strip())  # Remove leading and trailing whitespace.
18 Indent a line
19    and another
20 but not this.
21 >>> # Create a new string with leading and trailing whitespace removed.
22 >>> s_new = s.strip()
23 >>> # Use echoed value in interactive environment to see all leading and
24 >>> # trailing whitespace removed.
25 >>> s_new
26 'Indent a line\n   and another\nbut not this.'
```

9.5.4 `__repr__()`

All objects in Python contain the method `__repr__()`. This method provides the "official" string representation of a given object, i.e., you can think of `__repr__()` as standing for "representation." As with most methods that begin with underscores, this is not a method you typically use in day-to-day programming. However, it can be useful when debugging. Consider the string `s_new` created in line 22 of Listing 9.18. In line 25 this string is entered at the interactive prompt. The interactive environment echoes the string representation of this object. *But*, this is not the same as printing the object. Were we to print `s_new`, we would see the same output shown in lines 18 through 20. Note that in this output we cannot truly tell if the spaces were removed from the end of the line (we can tell the newline character was removed, but not the spaces that came before the newline character). However, from the output in line 26, we can see that these trailing spaces were removed.

Now assume you are writing (and running) a program but not working directly in the interactive environment. You want to quickly see if the internal representation of a string (such as `s_new` in the example above) is correct. What should you do? If you `print()` the string itself, it might mask the detail you seek. Instead, you can use the `__repr__()` method with `print()` to see the internal representation of the object. This is illustrated in Listing 9.19.

Listing 9.19 Demonstration that the `__repr__()` method gives the internal representation of a string.

```
>>> s = "   Indent a line\n   and another\nbut not this.  \n"
>>> s_new = s.strip()       # Create stripped string.
>>> # Print stripped string.  Cannot tell if all trailing space removed.
>>> print(s_new)
Indent a line
   and another
but not this.
>>> # Print __repr__() of stripped string.  Shows trailing space removed.
>>> print(s_new.__repr__())
'Indent a line\n   and another\nbut not this.'
>>> # Print __repr__() of original string.
>>> print(s.__repr__())
'   Indent a line\n   and another\nbut not this.  \n'
```

9.5.5 `find()` and `index()`

The `find()` and `index()` methods search for one string within another. The search is case sensitive. Both return the starting index where a substring can be found within a string. They only differ in that `find()` returns `-1` if the substring is not found while `index()` generates a `ValueError` (i.e., an exception) if the substring is not found. You may wonder why there is a need for these two different functions. Other than for convenience, there isn't a true need for two different functions. However, note that, because of negative indexing, `-1` is a valid index: it corresponds to the last element of the string. So, `find()` always returns a valid index and it is up to your code to recognize that `-1` really means "not found." In some situations it may be preferable to produce an error when the substring is not found. But, if you do not want this error to terminate your program, you must provide additional code to handle the exception.

The code in Listing 9.20 demonstrates basic operation of these two methods. In line 1 a string is created that has the character `I` in the first and 20th positions, i.e., in locations corresponding to indices of `0` and `19`. The `find()` method is used in line 2 to search for an occurrence of `I` and the result, shown in line 3, indicates `I` is the first character of the string. You may wonder if it is possible to find all occurrences of a substring, rather than just the first. The answer is yes and we'll return to this issue following the listing. Lines 4 and 5 show that the search is case sensitive, i.e., `i` and `I` do not match. Line 6 shows that one can search for a multi-character substring within a string. Here `index()` is used to search for `ere` in the string `s`. The resulting value of `11` gives the index where the substring starts within the string. In line 8 `find()` is used to search for `You` within the string. Since this does not occur in `s`, the result is `-1` as shown in line 9. Finally, in line 10, `index()` is used to search for `You`. Because this string is not found, `index()` produces a `ValueError`.

Listing 9.20 Demonstration of the use of the `find()` and `index()` methods.

```
>>> s = "I think, therefore I am."
>>> s.find("I")     # Find first occurrence of I.
0
>>> s.find("i")     # Search is case sensitive.
4
>>> s.index("ere") # Find first occurrence of ere.
11
>>> s.find("You")  # Search for You.  Not found.  Result of -1.
-1
>>> s.index("You") # Search for You.  Not Found.  Result is Error.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Let's return to the question of finding more than one occurrence of a substring. Both `find()` and `index()` have two additional (optional) arguments that can be used to specify the range of indices over which to perform a search. By providing both the optional `start` and `stop` arguments, the search will start from `start` and go up to, but not include, the `stop` value. If a `stop` value is not given, it defaults to the end of the string.

To demonstrate the use of these optional arguments, consider the code shown in Listing 9.21. In line 1 the same string is defined as in line 1 of Listing 9.20. This has `I`'s in the first and 20th positions. As in Listing 9.20, the `find()` method is used in line 2 without an optional argument to search for an occurrence of `I`. The result indicates `I` is the first character of the string. In line 4 `find()` is again asked to search for an `I` but the optional start value of `1` tells `find()` to start its search offset `1` from the beginning of the string, i.e., the search starts from the second character which is one beyond where the first occurrence is found. In this case `find()` reports that `I` is also the 20th character of the string. The statement in line 6 checks whether there is another occurrence by continuing the search from index 20. In this case `find()` reports, by returning `-1`, that the search failed. The discussion continues following the listing.

Listing 9.21 Demonstration of the use of the optional start argument for `find()`. `index()` behaves the same way except when the substring is not found (in which case an error is produced).

```
>>> s = "I think, therefore I am."
>>> s.find("I")        # Search for first occurrence.
0
>>> s.find("I", 1)     # Search for second occurrence.
19
>>> s.find("I", 20)    # Search for third occurrence.
-1
>>> s[1 : ].find("I") # Search on slice differs from full string.
18
```

As Listing 9.21 demonstrates, we can find multiple occurrences by continuing the search from the index one greater than that given for a previous successful search. Note that this continued search is related to, but slightly different from, searching a slice of the original string that starts just beyond the index of the previous successful search. This is illustrated in lines 7 and 8. In line 7 the slice `s[1 : ]` excludes the leading `I` from `s`. Thus, `find()` finds the second `I`, but it reports this as the 19th character (since it is indeed the 19th character of this slice but it is not the 19th character of the original string). If we exchange `index()` for `find()` in Listing 9.21, the results would be the same except when the substring is not found (in which case an exception is thrown instead of the method returning `-1`).

Let's consider a more complicated example in which we use a `for`-loop to search for all occurrences of a substring within a given string. Each occurrence is displayed with a portion of the "trailing context" in which the substring occurred. Code to accomplish this is shown in Listing 9.22. The code is discussed following the listing.

Listing 9.22 Code to search for multiple occurrences of a substring and to display the context in which the substring is found.

```
1  >>> s = "I think, therefore I am."
2  >>> look_for = "I" # Substring to search for.
3  >>> # Initialize variable that specifies where search starts.
4  >>> start_index = -1
5  >>> for i in range(s.count(look_for)):
6  ...     start_index = s.index(look_for, start_index + 1)
7  ...     print(s[start_index : start_index + len(look_for) + 5])
8  ...
9  I thin
10 I am.
11 >>> look_for = "th"
12 >>> start_index = -1
13 >>> for i in range(s.count(look_for)):
14 ...     start_index = s.index(look_for, start_index + 1)
15 ...     print(s[start_index : start_index + len(look_for) + 5])
16 ...
17 think,
18 therefo
```

In line 1 we again create a string with two `I`'s. In line 2 the variable `look_for`, which specifies the substring of interest, is set to `I`. In line 4 `start_index` is set to the integer value `-1`. This variable is used in the loop both to indicate where the substring was found and where a search should start. The starting point for a search is actually given by `start_index + 1`, hence `start_index` is initialized to `-1` to ensure that the first search starts from the character with an index of `0`. The header of the `for`-loop in line 5 yields a simple counted loop (i.e., we never use the loop variable). The `count()` method is used in line 5 to ensure the number of times the loop is executed is the number of times `look_for` occurs in `s`.

The first statement in the body of the `for`-loop, line 6, uses the `index()` method with a starting index of `start_index + 1` to determine where `look_for` is found. (Since the body of

the loop only executes as many times as the substring exists in s, we will not encounter a situation where the substring is not found. Thus, it doesn't matter if we use find() or index().) In line 6 the variable start_index is reset to the index where the substring was found. The print() statement in line 7 prints a slice of s that starts where the substring was found. The output is the substring itself with up to an additional five characters. Recall it is not an error to specify a slice that has a start or stop value that is outside the range of valid indices. For the second I in the string s there are, in fact, only four characters past the location of this substring. This is indicated in the output that appears in lines 9 and 10.

In line 11 the variable look_for is set to th (i.e., the substring for which the search will be performed is now two characters long) and start_index is reset to -1. Then, the same for-loop is executed as before. The output in lines 17 and 18 shows the two occurrences of the substring th within the string s together with the next five characters beyond the substring (although in line 17 the final character is the space character so it looks like only four additional characters are shown).

Something to note is that it is possible to store the *entire* contents of a file as a single string. Assume we want to search for occurrences of a particular (sub)string within this file. We can easily do so and, for each occurrence, display the substring with both a bit of leading and trailing context using a construct such as shown in Listing 9.22.

9.5.6 replace()

The replace() method is used to replace one substring by another. By default, all occurrences of the string targeted for replacement are replaced. An optional third argument can be given that specifies the maximum number of replacements. The use of the replace() method is demonstrated in Listing 9.23.

Listing 9.23 Demonstration of the replace() method. The optional third argument specifies the maximum number of replacements. It is not an error if the substring targeted for replacement does not occur.

```
1 >>> s = "I think, therefore I am."
2 >>> s.replace("I", "You")        # Replace I with You.
3 'You think, therefore You am.'
4 >>> s.replace("I", "You", 1)     # Replace only once.
5 'You think, therefore I am.'
6 >>> s.replace("I", "You", 5)     # Replace up to 5 times.
7 'You think, therefore You am.'
8 >>> s.replace("He", "She", 5)    # Target substring not in string.
9 'I think, therefore I am.'
```

9.6 split() and join()

split() breaks apart the original string and places each of the pieces in a list. If no argument is given, the splitting is done at every whitespace occurrence; consecutive whitespace characters

are treated the same way as a single whitespace character. If an argument is given, it is the substring at which the split should occur. We will call this substring the separator. Consecutive occurrences of the separator cause repeated splits and, as we shall see, produce an empty string in the resulting `list`.

Listing 9.24 demonstrates the behavior of `split()`. In line 1 a string is created containing multiple whitespace characters (spaces, newline, and tabs). The `print()` statement in line 2 shows the formatted appearance of the string. In line 5 the `split()` method is used to produce a `list` of the words in the string and we see the result in line 6. The `for`-loop in lines 7 and 8 uses `s.split()` as the iterable to cycle over each word in the string.

Listing 9.24 Demonstration of the `split()` method using the default argument, i.e., the splitting occurs on whitespace.

```
1  >>> s = "This  is \n only\t\ta test."
2  >>> print(s)
3  This  is
4   only          a test.
5  >>> s.split()
6  ['This', 'is', 'only', 'a', 'test.']
7  >>> for word in s.split():
8  ...     print(word)
9  ...
10 This
11 is
12 only
13 a
14 test.
```

Now consider the code in Listing 9.25 where `split()` is again called, but now an explicit argument is given. In line 1 a string is created with multiple repeated characters. In line 2 the splitting is done at the separator `ss`. Because there are two occurrences of `ss` in the original string, the resulting `list` has three elements. In line 4 the separator is `i`. Because there are four `i`'s in the string, the resulting `list` has five elements. However, since the final `i` is at the end of the string, the final element of the `list` is an empty string. When the separator is `s`, the resulting `list` contains two empty strings since there are two instances of repeated `s`'s in the string. The call to `split()` in line 8, with a separator of `iss`, results in a single empty string in the `list` because there are two consecutive occurrences of the separator `iss`.

Listing 9.25 Demonstration of the `split()` method when the separator is explicitly given.

```
1  >>> s = "Mississippi"
2  >>> s.split("ss")
3  ['Mi', 'i', 'ippi']
4  >>> s.split("i")
5  ['M', 'ss', 'ss', 'pp', '']
```

```
>>> s.split("s")
['Mi', '', 'i', '', 'ippi']
>>> s.split("iss")
['M', '', 'ippi']
```

Now let's consider an example that is perhaps a bit more practical. Assume there is `list` of strings corresponding to names. The names are written as a last name, then a comma and a space, and then the first name. The goal is to write these names as the first name followed by the last name (with no comma). The code in Listing 9.26 shows one way to accomplish this. The `list` of names is created in line 1. The `for`-loop cycles over all the names, setting the loop variable `name` to each of the individual names. The first line in the body of the loop uses the `split()` method with a separator consisting of both a comma and a space. This produces a `list` in which the last name is the first element and the first name is the second element. Simultaneous assignment is used to assign these values to appropriately named variables. The next line in the body of the `for`-loop simply prints these in the desired order. The output in lines 6 through 8 shows the code is successful.

Listing 9.26 Rearrangement of a collection of names that are given as last name followed by first name into output where the first name is followed by the last name.

```
>>> names = ["Obama, Barack", "Bush, George", "Jefferson, Thomas"]
>>> for name in names:
...     last, first = name.split(", ")
...     print(first, last)
...
Barack Obama
George Bush
Thomas Jefferson
```

The `join()` method is, in many ways, the converse of the `split()` method. Since `join()` is a string method, it must be called with (or on) a particular string object. Let's also call this string object the separator (for reasons that will be apparent in a moment). The `join()` method takes as its argument a `list` of strings.[6] The strings from the `list` are joined together to form a single new string but the separator is placed between these strings. The separator may be the empty string. Listing 9.27 demonstrates the behavior of the `join` method.

Listing 9.27 Demonstration of the `join()` method.

```
>>> # Separate elements from list with a comma and space.
>>> ", ".join(["Obama", "Barack"])
'Obama, Barack'
>>> "-".join(["one", "by", "one"])   # Hyphen separator.
```

[6] In fact the argument can be any iterable that produces strings, but for now we will simply say the argument is a `list` of strings.

```
'one-by-one'
>>> "".join(["smashed", "together"]) # No separator.
'smashedtogether'
```

Let's construct something a little more complicated. Assume we want a function to count the number of printable characters (excluding space) in a string. We want to count only the characters that produce something visible on the screen or page (and we will assume the string doesn't contain anything out of the ordinary such as Null or Backspace characters). This can be accomplished by first splitting a string on whitespace and then joining the list back together with an empty-space separator. The resulting string will have all the whitespace discarded and we can simply find the length of this string. Listing 9.28 demonstrates how to do this and provides a function that yields this count.

Listing 9.28 Technique to count the visible characters in a string.

```
>>> text = "A B C D B's?\nM N R no B's.  S A R!"
>>> print(text)
A B C D B's?
M N R no B's.  S A R!
>>> tlist = text.split()     # List of strings with no whitespace.
>>> clean = "".join(tlist)   # Single string with no whitespace.
>>> clean
"ABCDB's?MNRnoB's.SAR!"
>>> len(clean)               # Length of string.
21
>>> def ink_counter(s):
...     slist = s.split()
...     return len("".join(slist))
...
>>> ink_counter(text)
21
>>> ink_counter("Hi    Ho")
4
```

9.7 Format Strings and the `format()` Method

We have used the `print()` function to generate output and we have used the interactive interpreter to display values (by entering an expression at the interactive prompt and then hitting return). In doing so, we have delegated the task of formatting the output to the the `print()` function or to the interactive interpreter.[7] Often, however, we need to exercise finer control over the appearance of our output. We can do this by creating strings that have precisely the appearance we desire. These strings are created using *format strings* in combination with the `format()` method.

[7] Actually, the formatting is largely dictated by the objects themselves since it is the `__str__()` method of an object that specifies how an object should appear.

Format strings consist of literal characters that we want to appear "as is" as well as *replacement fields* that are enclosed in braces. A replacement field serves a dual purpose. First, it serves as a placeholder, showing the relative placement where additional text should appear within a string. Second, a replacement field *may* specify various aspects concerning the formatting of the text, such as the number of spaces within the string to dedicate to displaying an object, the alignment of text within these spaces, the character to use to fill any space that would otherwise be unfilled, the digits of precision to use for `floats`, etc. We will consider several of these formatting options, but not all.[8] However, as will be shown, a replacement field need not specify formatting information. When formatting information is omitted, Python uses default formats (as is done when one simply supplies an object as an argument to the `print()` function).

To use a format string, you specify the format string *and* invoke the `format()` method on this string. The arguments of the `format()` method supply the objects that are used in the replacement fields. A replacement field is identified via a pair of (curly) braces, i.e., `{` and `}`.

Without the `format()` method, braces have no special meaning. To emphasize this, consider the code in Listing 9.29. The strings given as arguments to the `print()` functions in lines 1 and 3 and the string in line 5 all contain braces. If you compare these strings to the subsequent output in lines 2, 4, and 6, you see that the output mirrors the string literals—nothing has changed. However, as we will see, this is *not* the case when the `format()` method is invoked, i.e., `format()` imparts special meaning to braces and their contents.

Listing 9.29 Demonstration that braces within a string have no special significance without the `format()` method.

```
1 >>> print("These {}'s are simply braces.  These are too: {}")
2 These {}'s are simply braces.  These are too: {}
3 >>> print("Strange combination of characters: {:<20}")
4 Strange combination of characters: {:<20}
5 >>> "More strange characters: {:=^20.5f}"
6 'More strange characters: {:=^20.5f}'
```

9.7.1 Replacement Fields as Placeholders

For examples of the use of format strings, consider the code in Listing 9.30. In these examples the replacement fields, which are the braces, are simply placeholders—they mark the location (or locations) where the argument(s) of the `format()` method should be placed. There is a one-to-one correspondence between the (placeholder) replacement fields and the arguments of `format()`. This code is discussed further following the listing.

Listing 9.30 Demonstration of format strings where the replacement fields provide only placement information (not formatting information).

```
1 >>> "Hello {}!".format("Jack")
2 'Hello Jack!'
```

[8] Complete details can be found at **docs.python.org/py3k/library/string.html#formatstrings**.

```
3 >>> "Hello {} and {}!".format("Jack", "Jill")
4 'Hello Jack and Jill!'
5 >>> "Float: {}, List: {}".format(1 / 7, [1, 2, 3])
6 'Float: 0.14285714285714285, List: [1, 2, 3]'
```

The string in line 1 contains braces and the `format()` method is invoked on this string. Thus, by definition, this is a *format string*. Within this string there is single replacement field between `Hello` and the exclamation point. The one argument of the `format()` method, i.e., the string `Jack`, is substituted into this placeholder. The resulting string is shown in line 2. In line 3, there are two replacement fields: one after `Hello` and one immediately before the exclamation mark. Correspondingly, `format()` has two arguments. The first argument is substituted into the first replacement field and the second argument is substituted into the second replacement field. The resulting string is shown in line 4.

Line 5 illustrates that replacement fields can serve as generic placeholders for any object, i.e., any object we provide as an argument to the `print()` function can be "mapped" to a replacement field. In line 5 `format()`'s first argument is an expression that evaluates to a `float` and the second is a three-element `list`. In the resulting string, shown in line 6, the braces are replaced by the `float` and `list` values.

Replacement fields can be much more flexible, powerful, and, well, complicated than these simple placeholders. Rather than using merely empty braces, information can be provided between the braces. We will consider two pieces of information: the field "name" and the format specifier.

A replacement field can contain a *field name*. Although the field name can be constructed in various ways, we will consider only a numeric "name." If an integer value is given as the field name, it specifies the argument of `format()` to which the replacement field corresponds. You can think of the "name" as an index and the arguments of `format()` as having indices starting from zero. Thus, a field name of `{0}` corresponds to the first argument of the `format()` method, a field name of `{1}` corresponds to the second argument, and so on. In this way `format()` doesn't have to have a separate argument for each replacement field. Listing 9.31 demonstrates this:

Listing 9.31 Demonstration of the use of field names in the replacement fields of a format string. Only numeric "names" are considered. The field name specifies the argument of the `format()` method to which the replacement field corresponds.

```
1 >>> "Hello {0} and {1}.  Or, is it {1} and {0}?".format("Jack", "Jill")
2 'Hello Jack and Jill.  Or, is it Jill and Jack?'
3 >>> "Hello {0} {0} {0} {0}.".format("Major")
4 'Hello Major Major Major Major.'
5 >>> "Hello {} {} {} {}.".format("Major")
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 IndexError: tuple index out of range
```

In line 1 there are four replacement fields in the format string, but the `format()` method has only two arguments. Note that these replacement fields are still simply placeholders—they do

not provide any formatting information beyond location. Since these replacement fields now have field names, they specify the arguments of the `format()` method for which they are placeholders. Again, a replacement field of `{0}` specifies that the first argument of `format()` should appear in this location, while `{1}` specifies the second argument should replace the replacement field. Note the resulting string in line 2 and how `Jack` and `Jill` appear twice (and in different order). In the format string in line 3 there are four replacement fields but there is only one argument in the `format()` method. Because each of the replacement fields is `{0}`, the first (and only) argument of the `format()` method appears four times in the resulting string in line 4.[9]

The expression in line 5 also has a format string with four replacement fields and the `format()` method again has a single argument. Here, however, the replacement fields do not provide field names and, as indicated by the resulting exception shown in lines 6 through 8, Python cannot determine how to match the replacement fields to `format()`'s argument. (In this particular case the error message itself is not especially useful in describing the root of the problem.)

Listing 9.32 gives another example in which the format string simply provides placeholder information (and the replacement fields are unnamed). This code further demonstrates that objects with different types can be used with format strings. Additionally, this code illustrates that the `format()` method returns a string (and it does not differ from any other string—it can be used in any way one would normally use a string).

Listing 9.32 The arguments of the `format()` method can be variables, literals, or other expressions. This code also demonstrates that the return value of the `format()` method is simply a string.

```
1  >>> opponent = "Eagles"
2  >>> score_us = 17
3  >>> score_them = 22
4  >>> "Seahawks {}, {} {}".format(score_us, opponent, score_them)
5  'Seahawks 17, Eagles 22'
6  >>> final = "Seahawks {}, {} {}".format(score_us + 7, opponent,
7  ... score_them)
8  >>> type(final)
9  <class 'str'>
10 >>> print(final)
11 Seahawks 24, Eagles 22
```

In line 4 `format()`'s arguments are all variables—the first is an integer, the second is a string, and the third is also an integer. As you would expect, these arguments must all be separated by commas. When it comes to replacement fields within the format string, there is no restriction placed on what comes between the fields. In line 4 there is a comma between the first and second replacement field, but this is a literal comma—this indicates a comma should appear in the resulting string at this location. Line 5 shows the resulting string. In line 6, `format()`'s first argument is an integer expression and the other two arguments remain unchanged. The resulting string is assigned to the variable `final`. After confirming `final`'s type is `str` (lines 8 and 9), the `print()` statement in line 10 displays the `final` string.

[9] If you haven't read Joseph Heller's *Catch-22*, you should.

9.7.2 Format Specifier: Width

The contents of a replacement field can also provide instructions about the formatting of output. We will consider several of the myriad ways in which the output can be formatted. Section 9.7.1 explains that replacement fields can be named. To provide formatting information, we must provide a *format specifier* within the replacement field. The format specifier must be preceded by a colon. The field name, if present, appears to the left of this colon. Thus, we can think of a general replacement field as consisting of

```
<replacement_field> = {<field_name>:<format_specifier>}
```

As both the field name and format specifier are optional, alternate forms of replacement fields are

```
<replacement_field> = {}
```

and

```
<replacement_field> = {<field_name>}
```

and

```
<replacement_field> = {:<format_specifier>}
```

Perhaps the simplest formatting information is the (minimum) width of the resulting field. Thus, for a particular replacement field, this is the number of spaces provided to hold the string representation of the corresponding object given as an argument to `format()`. If the number of spaces is insufficient to display the given object, Python will use as many additional spaces as necessary. If the object requires fewer characters than the given width, the "unused" spaces are, by default, filled with blank spaces. The code in Listing 9.33 demonstrates various aspects of the width specifier.

Listing 9.33 Demonstration of the width specifier. If the string representation of the corresponding object is longer than the width, the output will exceed the width as necessary. If the object is shorter than the width, blank spaces are used to fill the field.

```
>>> "{0:5}, {1:5}, and {2:5}!".format("Red", "White", "Periwinkle")
'Red  , White, and Periwinkle!'
>>> "{0:5}, {1:5}, and {2:5}!".format(5, 10, 15)
'    5,    10, and    15!'
>>> # "Name" portion of replacement field is unnecessary when there is
>>> # a one-to-one and in-order correspondence between the fields and
>>> # arguments of format().
>>> "{:5}, {:5}, and {:5}!".format(5, 10, 15)
'    5,    10, and    15!'
>>> # Colon in replacement field is not an operator.  Cannot have
>>> # spaces between the field name and colon.
>>> "{0 : 5}, {1 : 5}, and {2 : 7}!".format(5, 10, 15)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '0 '
```

In line 1 there are three replacement fields, each of which has a width of 5. The argument corresponding to the first field (Red) has a length of 3, i.e., two characters less than the width of the field. Thus, in line 2, there are two spaces between Red and the comma that follows the field. Note that Red has been left-justified in the field. The argument corresponding to the second replacement field (White) has a length of 5 and hence in line 2 there is no space between White and the comma that follows the field. The argument corresponding to the third replacement field has a length of 9, i.e., four characters longer than the width of the field. In cases such as this the field simply grows to whatever length is necessary to accommodate the string. (When no width is given, the field is always as wide as the string representation of the object.)

In line 3 of Listing 9.33 the same format string is used as in line 1, but now the arguments of the format() method are integers. The output in line 4 shows that numbers are right justified within a field. Line 8 is similar to line 3 except the field names have been dropped (i.e., the portion of the replacement field to the left of the colon). A field name is unnecessary when there is a one-to-one and in-order correspondence between the replacement fields and the arguments of the format() method.

To enhance readability in this textbook, we typically surround an operator by spaces. However, the colon in a replacement field is not an operator and, as of Python 3.2.2, there cannot be space between the field name and the colon.[10] This is illustrated in lines 12 through 15 where a space preceding the colon results in an error. (Again, the error message is not especially useful in pointing out the cause of the error.)

9.7.3 Format Specifier: Alignment

Listing 9.33 shows that, by default, strings are left-justified while numbers are right-justified within their fields, but we can override this default behavior. We can explicitly control the alignment by providing one of four characters prior to the width: < for left justification, ^ for centering, > for right justification, and = to left-justify the sign of a number while right-justifying its magnitude. This is demonstrated in Listing 9.34. The same format string is used in both lines 1 and 3. This string has three replacement fields. The first specifies left justification, the second specifies centering, and the third specifies right justification. The output in lines 2 and 4 demonstrates the desired behavior. The expressions in lines 5 and 7 use the = alignment character. By default the plus sign is not displayed for positive numbers.

Listing 9.34 Alignment of output within a field can be controlled with the aligned characters < (left), ^ (centering), and > (right).

```
1 >>> print("|{:<7}||{:^7}||{:>7}|".format("bat", "cat", "dat"))
2 |bat    ||  cat  ||    dat|
3 >>> print("|{:<7}||{:^7}||{:>7}|".format(123, 234, 345))
4 |123    ||  234  ||    345|
5 >>> "{:=7}".format(-123)
6 '-   123'
7 >>> "{:=7}".format(123)
```

[10] A space is permitted following the colon but, as described in 9.7.4, is interpreted as the "fill character" and can affect formatting of the output.

```
'    123'
```

9.7.4 Format Specifier: Fill and Zero Padding

One can also specify a "fill character." If an object does not fill the allocated space of the replacement field, the default "fill character" is a blank space. However, this can be overridden by providing an additional character between the colon and the alignment character, i.e., at the start of the format specifier.[11] This is demonstrated in Listing 9.35. In line 1 the fill characters for the three fields are <, *, and =. In line 3 the fill characters are >, ^, and 0. The last fill character is of particular interest for reasons that will become clear shortly. The discussion of this code continues following the listing.

Listing 9.35 A "fill character" can be specified between the colon and alignment character. This character will be repeated as necessary to fill any spaces that would otherwise be blank.

```
1  >>> print("|{:<<7}||{:*^7}||{:=>7}|".format("bat", "cat", "dat"))
2  |bat<<<<||**cat**||====dat|
3  >>> print("|{:><7}||{:^^7}||{:0>7}|".format(123, 234, 345))
4  |123>>>>||^^234^^||0000345|
5  >>> "{:07} {:07} {:07}".format(345, 2.5, -123)
6  '0000345 00002.5 -000123'
7  >>> "{:0=7} {:0=7} {:0>7}".format(2.5, -123, -123)
8  '00002.5 -000123 000-123'
9  >>> "{:07}".format("mat")
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 ValueError: '=' alignment not allowed in string format specifier
13 >>> "{:0>7}".format("mat")
14 '0000mat'
```

When the argument of the `format()` method is a numeric value, a zero (`0`) can precede the width specifier. This indicates the field should be "zero padded," i.e., any unused spaces should be filled with zeroes, but the padding should be done between the sign of the number and its magnitude. In fact, a zero before the width specifier is translated to a fill and alignment specifier of `0=`. This is demonstrated in lines 5 through 8 of Listing 9.35. In line 5 the arguments of `format()` are an integer, a `float`, and a negative integer. For each of these the format specifier is simply `07` and the output on line 6 shows the resulting zero-padding. In line 7 the arguments of `format()` are a `float` and two negative integers. The first two replacement fields use `0=7` as the format specifier. The resulting output on line 8 indicates the equivalence of `07` and `0=7`. However, the third field in line 7 uses `0>7` as the format specifier. This is *not* equivalent to `07` (or `0=7`) in that the sign of the number is now adjacent to the magnitude rather than on the left side of the field.

[11] One restriction is that the fill character cannot be a closing brace (}).

Lines 9 through 12 of Listing 9.35 indicate that zero-padding cannot be used with a string argument. The error message may appear slightly cryptic until one realizes that zero-padding is translated into alignment with the = alignment character. Note that, as shown in lines 13 and 14, a format specifier of `:0>7` does not produce an error with a string argument and does provide a type of zero-padding. But, really this is just right justification of the string with a fill character of `0`.

9.7.5 Format Specifier: Precision (Maximum Width)

Section 9.7.2 describes how an integer is used to specify the (minimum) width of a field. Somewhat related to this is the *precision* specifier. This is also an integer, but unlike the width specifier, the precision specifier is preceded by a dot. Precision has different meanings depending on the type of the object being formatted. If the object is a `float`, the precision dictates the number of digits to display. If the object is a string, the precision indicates the maximum number of characters allowed to represent the string. (One cannot specify a precision for an integer argument.) As will be shown in a moment, when a width is specified, it can be less than, equal to, or greater than the precision. Keep in mind that the width specifies the size of the field while the precision more closely governs the characters used to format a given object.

Listing 9.36 demonstrate how the precision specifier affects the output. Line 1 defines a `float` with several non-zero digits while line 2 defines a `float` with only three non-zero digits. Line 3 defines a string with seven characters. Lines 5 and 6 show the default formatting of these variables. The output in line 6 corresponds to the output obtained when we write `print(x, y, s)` (apart from the leading and trailing quotation marks identifying this as a string). The discussion continues following the listing.

Listing 9.36 Use of the precision specifier to control the number of digits or characters displayed.

```
1  >>> x = 1 / 7       # float with lots of digits.
2  >>> y = 12.5        # float with only three digits.
3  >>> s = "stringy"   # String with seven characters.
4  >>> # Use default formatting.
5  >>> "{}, {}, {}".format(x, y, s)
6  '0.14285714285714285, 12.5, stringy'
7  >>> # Specify a precision of 5.
8  >>> "{:.5}, {:.5}, {:.5}".format(x, y, s)
9  '0.14286, 12.5, strin'
10 >>> # Specify a width of 10 and a precision of 5.
11 >>> "{:10.5}, {:10.5}, {:10.5}".format(x, y, s)
12 '   0.14286,       12.5, strin     '
13 >>> # Use alternate numeric formatting for second term.
14 >>> "{:10.5}, {:#10.5}, {:10.5}".format(x + 100, y, s)
15 '    100.14,     12.500, strin     '
```

In line 8 each of the format specifiers simply specifies a precision of `5`. Thus, in line 9, we see five digits of precision for the first `float` (the zero to the left of the decimal point is not considered a

significant digit). The second `float` is displayed with all its digits (i.e., all three of them). The string is now truncated to five characters. In line 11 both a width and a precision are given. The width is `10` while the precision is again `5`. In this case the "visible" output in line 12 is no different from that of line 9. However, these characters now appear in fields of width `10` (that are filled with spaces as appropriate). In line 14 the value of the first argument is increased by `100` while the format specifier of the second argument now has the hash symbol (`#`) preceding the width. In this context the hash symbol means to use an "alternate" form of numeric output. For a `float` this translates to showing any trailing zeros for the given precision. Notice that in line 15 the first term still has five digits of precision: three before the decimal point and two following it.

As we will see in the next section, the meaning of precision changes slightly for `floats` depending on the *type specifier*.

9.7.6 Format Specifier: Type

Finally, the format specifier may be terminated by a character that is the *type specifier*. The type specifier is primarily used to control the appearance of integers or `floats`. There are some type specifiers that can only be applied to integers. One of these is `b` which dictates using the binary representation of the number rather than the usual decimal representation. Two other type specifiers that pertain only to integers are `d` and `c`. A decimal representation is obtained with the `d` type specifier, but this is the default for integers and thus can be omitted if decimal output is desired. The type specifier `c` indicates that the value should be displayed as a character (as if using `chr()`).

The type specifiers that are nominally for `floats` also work for integer values. These type specifiers include `f` for fixed-point representation, `e` for exponential representation (with one digit to the left of the decimal point), and `g` for a "general format" that typically tries to format the number in the "nicest" way. If the value being formatted is a `float` and no type specifier is provided, then the output is similar to `g` but with at least one digit beyond the decimal point (by default `g` will not print the decimal point or a trailing digit if the value is a whole number).

The use of type specifiers is demonstrated in Listing 9.37. The code is discussed following the listing.

Listing 9.37 Use of various type specifiers applied to an integer and two `floats`.

```
1 >>> # b, c, d, f, e, and g type specifiers with an integer value.
2 >>> "{0:b}, {0:c}, {0:d}, {0:f}, {0:e}, {0:g}".format(65)
3 '1000001, A, 65, 65.000000, 6.500000e+01, 65'
4 >>> # f, e, and g with a float value with zero fractional part.
5 >>> "{0:f}, {0:e}, {0:g}".format(65.0)
6 '65.000000, 6.500000e+01, 65'
7 >>> # f, e, and g with a float value with non-zero fractional part.
8 >>> "{0:f}, {0:e}, {0:g}".format(65.12345)
9 '65.123450, 6.512345e+01, 65.1235'
```

In line 2 the argument of `format()` is the integer `65`. The subsequent output in line 3 starts with `1000001` which is the binary equivalent of `65`. We next see `A`, the ASCII character corresponding

to 65. The remainder of the line gives the number 65 formatted in accordance with the d, f, e, and g type specifiers.

In line 5 the f, e, and g specifiers are used with the float value 65.0, i.e., a float with a fractional part of zero. Note that the output produced with the g type specifier lacks the trailing decimal point and 0 that we typically expect to see for a whole-number float value. (It is an error to use any of the integer type specifiers with a float value, e.g., b, c, and d cannot be used with a float value.) Line 8 shows the result of using these type specifiers with a float that has a non-zero fractional part.

We can combine type specifiers with any of the previous format specifiers we have discussed. Listing 9.38 is an example where the alignment, width, and precision are provided together with a type specifier. In the first format specifier the alignment character indicates right alignment, but since this is the default for numeric values, this character can be omitted. This code demonstrates the changing interpretation of the precision caused by use of different type specifiers. The first two values in line 2 show that the precision specifies the number of digits to the right of the decimal sign for f and e specifiers, but the third values shows that it specifies the total number of digits for the g specifier.

Listing 9.38 The intepretation of precision depends on the type specifier. For e and f the precision is the number of digits to the right of the decimal sign. For g the precision corresponds to the total number of digits.

```
1 >>> "{0:>10.3f}, {0:<10.3e}, {0:^10.3g}".format(65.12345)
2 '    65.123, 6.512e+01 ,    65.1   '
```

9.7.7 Format Specifier: Summary

To summarize, a format specifier starts with a colon and then may contain any of the terms shown in brackets in the following (each of the terms is optional):

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

A brief description of the terms is provided below. We note that not all of these terms have been discussed in the preceding material. The interested reader is encouraged either to read the formatting information available online[12] or simply to enter these in a format string and observe the resulting output.

fill: Fill character (may be any character other than }). When given, the fill character must be followed by an alignment character.

align: < (left), ^ (center), > (right), or = (for numeric values left-justify sign and right-justify magnitude).

[12] **docs.python.org/py3k/library/string.html#formatstrings**.

sign: + (explicitly show positive and negative signs), − (show only negative signs; default), or <*space*> (leading space for positive numbers, negative sign for negative numbers).

#: Use alternate form of numeric output.

0: Use zero padding (equivalent to fill and alignment of `0=`).

width: Minimum width of the field (integer).

,: Show numeric values in groups of three, e.g., `1,000,000`.

.precision: Maximum number of characters for strings (integer); number of digits of precision for `floats`. For `f`, `F`, `e`, and `E` type specifiers this is the number of digits to the right of the decimal point. For `g`, `G`, and <*none*> type specifiers the precision is the total number of digits.

type: Integers: `b` (binary), `c` (convert to character; equivalent to using `chr()`), `d` (decimal; default), `o` (octal), `x` or `X` (hexadecimal with lowercase or uppercase letters), `n` (same as `d` but with "local" version of the `,` grouping symbol). `floats`: `e` or `E` (exponential notation), `f` or `F` (fixed point), `g` or `G` (general format), `n` (same as `g` but with "local" version of `,` grouping symbol), `%` (multiply value by 100 and display using `f`), <*none*> (similar to `g` but with at least one digit beyond the decimal point). Strings: `s` (default).

9.7.8 A Formatting Example

Assume we must display a time in terms of minutes and seconds down to a tenth of a second. (Here we are thinking of "time" as a duration, as in the time it took a person to complete a race. We are not thinking in terms of time of day.) In general, times are displayed with the minutes and seconds separated by a colon. For example, 19:34.7 would be 19 minutes and 34.7 seconds.

Assume we have a variable `mm` that represents the number of minutes and a variable `ss` that represents the number of seconds. The question now is: How can we put these together so that the result has the "standard" form of MM:SS.s? The code in Listing 9.39 shows an attempt in line 3 to construct suitable output for the given minutes and seconds, but this fails because of the spaces surrounding the colon. In line 5 a seemingly successful attempt is realized using a statement similar to the one in line 3 but setting `sep` to the empty string. Line 7 also appears to yield the desired output by converting the minutes and seconds to strings and then concatenating these together with a colon. But, are the statements in lines 5 and 7 suitable for the general case?

Listing 9.39 Attempt to construct a "standard" representation of a time to the nearest tenth of a second.

```
1 >>> mm = 19      # Minutes.
```

```
>>> ss = 34.7    # Seconds.
>>> print(mm, ":", ss)              # Fails because of spurious space.
19 : 34.7
>>> print(mm, ":", ss, sep="")      # Output appears as desired.
19:34.7
>>> print(str(mm) + ":" + str(ss)) # Output appears as desired.
19:34.7
```

Let's consider some other values for the seconds but continue to use the `print()` statement from line 7 of Listing 9.39. In line 2 of Listing 9.40 the seconds are set to an integer value. The subsequent time in line 4 is flawed in that it does not display the desired tenths of a second. In line 5 the seconds are set to a value that is less than ten. In this case the output, shown in line 7 is again flawed in that two digits should be used to display the seconds (i.e., the desired format is MM:SS.s and MM:S.s is not considered acceptable). Finally, in line 8 the seconds are set to a value with many digits of precision. The output in line 10 is once again not what is desired in that too many digits are displayed—we only want the seconds to the nearest tenth.

Listing 9.40 Demonstration that the desired format of MM:SS.s is not obtained for the given values of seconds when the `print()` statement from line 7 of Listing 9.39 is used.

```
>>> mm = 19
>>> ss = 14
>>> print(str(mm) + ":" + str(ss))  # Output lacks tenths of second.
19:14
>>> ss = 7.3
>>> print(str(mm) + ":" + str(ss))  # Output missing a digit.
19:7.3
>>> ss = 34.7654321
>>> print(str(mm) + ":" + str(ss))  # Too many digits in output.
19:34.7654321
```

At this point you probably realize that a general solution for these different values for seconds is achieved using a format string with suitable format specifiers. We assume that the minutes are given as an integer value and set the width for the minutes to `2`, but we know the output will increase appropriately if the number of minutes exceeds two digits. For the seconds, we want a fixed-point (`float`) representation with the width of the field equal to `4` and one digit of precision (one digit to the right of the decimal point), and zero-padding is used if the seconds value is less than ten. Listing 9.41 demonstrates that the desired output is obtained for all the values of seconds used in Listing 9.40.

Listing 9.41 Use of a format string to obtain the desired "standard" representation of time.

```
>>> mm = 19
>>> ss = 14
>>> "{0:2d}:{1:04.1f}".format(mm, ss)
```

```
'19:14.0'
>>> ss = 7.3
>>> "{0:2d}:{1:04.1f}".format(mm, ss)
'19:07.3'
>>> ss = 34.7654321
>>> "{0:2d}:{1:04.1f}".format(mm, ss)
'19:34.8'
```

9.8 Chapter Summary

Strings are *immutable*, i.e., they cannot be changed (although an identifier assigned to a string variable can be assigned to a new string).

Strings can be concatenated using the plus operator. With operator overloading, a string can be repeated by multiplying it by an integer.

Indexing and slicing of strings is the same as for `lists` and `tuples` (but working with characters rather than elements).

The **`len()`** function returns the number of characters in its string argument.

The **`str()`** function returns the string representation of its argument.

ASCII provides a mapping of characters to numeric values. There are a total of 128 ASCII characters, 95 of which are printable (or graphic) characters.

The **`ord()`** function returns the numeric value of its character argument. The **`chr()`** function returns the character for its numeric argument. `ord()` and `chr()` are inverses.

An *escape sequence* within a string begins with the backslash character which alters the usual meaning of the adjacent character or characters. For example, `'\n'` is the escape sequence for the newline character.

Strings have many methods including the following, where "a given string" means the string on which the method is invoked:

- **`split()`**: Returns a `list` of elements obtained by splitting the string apart at the specified substring argument. Default is to split on whitespace.
- **`join()`**: Concatenates the (string) elements of the `list` argument with a given string inserted between the elements. The insertion may be any string including an empty string.
- **`capitalize()`**, **`title()`**, **`lower()`**, **`upper()`**, **`swapcase()`**: Case methods that return a new string with the case set appropriately.
- **`count()`**: Returns the number of times the substring argument occurs in a given string.
- **`find()`** and **`index()`**: Return the index at which the substring argument occurs within a given string. Optional arguments can be used to specify the range of the search. `find()` returns `-1` if the substring is not found while `index()` raises an exception if the substring is not found.
- **`lstrip()`**, **`rstrip()`**, **`strip()`**: Strip whitespace from a given string (from the left, right, or from both ends, respectively).

- **`replace()`**: Replaces an "old" substring with a "new" substring in a given string.
- **`__repr__()`**: Returns a string that shows the "official" representation of a given string (useful for debugging purposes when not in an interactive environment).
- **`format()`**: Allows fine-grain control over the appearance of an object within a string. Used for formatting output.

9.9 Review Questions

1. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[1])
```

   (a) `J`
   (b) `e`
   (c) `Jane`
   (d) `a`

2. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[-1])
```

   (a) `J`
   (b) `e`
   (c) `Jane`
   (d) `a`

3. What is printed by the following Python fragment?

```
s = "Jane Doe"
print(s[1:3])
```

   (a) `Ja`
   (b) `Jan`
   (c) `an`
   (d) `ane`

4. What is the output from the following program, if the input is `Spam And Eggs`?

```
def main():
    msg = input("Enter a phrase: ")
    for w in msg.split():
        print(w[0], end="")

main()
```

(a) `SAE`

(b) `S A E`

(c) `S S S`

(d) `Spam And Eggs`

(e) None of the above.

5. What is the output of this program fragment?

```
for x in "Mississippi".split("i"):
    print(x, end="")
```

(a) `Mssspp`

(b) `M ssissippi`

(c) `Mi ssi ssi ppi`

(d) `M ss ss pp`

6. ASCII is

(a) a standardized encoding of written characters as numeric codes.

(b) an encryption system for keeping information private.

(c) a way of representing numbers using binary.

(d) computer language used in natural language processing.

7. What function can be used to get the ASCII value of a given character?

(a) `str()`

(b) `ord()`

(c) `chr()`

(d) `ascii()`

(e) None of the above.

8. What is output produced by the following?

```
s = "absense makes the brain shrink"
x = s.find("s")
y = s.find("s", x + 1)
print(s[x : y])
```

(a) `sens`

(b) `ens`

(c) `en`

(d) `sen`

9. One difference between strings and lists in Python is that

(a) strings are sequences, but lists aren't.

(b) lists can be indexed and sliced, but strings can't.

(c) lists are mutable (changeable), but strings immutable (unchangeable).

(d) strings can be concatenated, but lists can't.

10. What is an appropriate `for`-loop for writing the characters of the string `s`, one character per line?

(a)
```
for ch in s:
    print(ch)
```

(b)
```
for i in range(len(s)):
    print(s[i])
```

(c) Neither of the above.

(d) Both of the above.

11. The following program fragment is meant to be used to find the sum of the ASCII values for all the characters in a string that the user enters. What is the missing line in this code?

```
phrase = input("Enter a phrase: ")
ascii_sum = 0       # accumulator for the sum
for ch in phrase:
    ##### missing line here
print(ascii_sum)
```

(a) `ascii_sum = ascii_sum + ch`

(b) `ascii_sum = chr(ch)`

(c) `ascii_sum = ascii_sum + chr(ch)`

(d) `ascii_sum = ascii_sum + ord(ch)`

12. What is the result of evaluating the expression `chr(ord('A') + 2)`?

    (a) `'A2'`

    (b) `'C'`

    (c) `67`

    (d) An error.

    (e) None of the above.

13. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0:
    s1 = ch + s1

print(s1)
```

    (a) `A Toyota`

    (b) `atoyoT A`

    (c) None of the above.

14. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0[ : : -1]:
    s1 = ch + s1

print(s1)
```

    (a) `A Toyota`

    (b) `atoyoT A`

    (c) None of the above.

15. What is the output of the following code?

```
s0 = "A Toyota"
s1 = ""
for ch in s0[-1 : 0 : -1]:
    s1 = s1 + ch

print(s1)
```

(a) `A Toyota`

(b) `atoyoT A`

(c) None of the above.

16. What is the value of `z` after the following has been executed:

```
s = ''
for i in range(-1, 2):
   s = s + str(i)

z = int(s)
```

(a) `0`

(b) `2`

(c) `-1012`

(d) `-101`

(e) This code produces an error.

17. What is the value of `ch` after the following has been executed?

```
ch = 'A'
ch_ascii = ord(ch)
ch = chr(ch_ascii + 2)
```

(a) `'A'`

(b) `67`

(c) `'C'`

(d) This code produces an error.

18. What is the output produced by the `print()` statement in the following code?

```
s1 = "I'd rather a bottle in front of me than a frontal lobotomy."
s2 = s1.split()
print(s2[2])
```

(a) `'`

(b) `d`

(c) `rather`

(d) `a`

(e) `bottle`

19. What is the output produced by the `print()` statement in the following code?

```
s1 = "I'd\nrather a bottle in front of me than a frontal lobotomy."
s2 = s1.split()
print(s2[2])
```

(a) `'`

(b) `d`

(c) `rather`

(d) `a`

(e) `bottle`

(f) None of the above.

20. The variable `s` contains the string `'cougars'`. A programmer wants to change this variable so that it is assigned the string `'Cougars'`. Which of the following will accomplish this?

(a)
```
s.upper()
```

(b)
```
s[0] = 'C'
```

(c)
```
s = 'C' + s[1 : len(s)]
```

(d)
```
s.capitalize()
```

(e) All of the above.

(f) None of the above.

21. What output is produced by the following code?

```
1 s = "Jane Doe"
2 print(s[1 : 3: -1])
```

(a) `aJ`

(b) `naJ`

(c) `na`

(d) `en`

(e) None of the above.

22. After the following commands have been executed, what is the value of `x`?

```
s = "this is a test"
x = s.split()
```

23. After the following commands have been executed, what is the value of `y`?

```
s = "this is a test"
y = s.split("s")
```

24. Recall that the `str()` function returns the string equivalent of its argument. What is the output produced by the following:

```
a = 123456
s = str(a)
print(s[5] + s[4] + s[3] + s[2])
```

25. What is the value of `count` after the following code has been executed?

```
s = "He said he saw Henry."
count = s.count("he")
```

(a) `0`

(b) `1`

(c) `2`

(d) `3`

(e) None of the above.

26. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[ : -2] + "w."
```

(a) `Grow.`

(b) `kw.`

(c) `k!w`

(d) None of the above.

27. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[-2] + "w."
```

(a) `Grow.`

(b) `kw.`

(c) `k!w`

(d) None of the above.

28. What is the value of `s2` after the following has been executed?

```
s1 = "Grok!"
s2 = s1[-2 : ] + "w."
```

(a) `kw.`

(b) `Grow.`

(c) `k!w`

(d) None of the above.

ANSWERS: 1) d; 2) b; 3) c; 4) a; 5) a; 6) a; 7) b; 8) d; 9) c; 10) d; 11) d; 12) b; 13) b; 14) a; 15) c; 16) d; 17) c; 18) d; 19) d; 20) c; 21) e; 22) `['this', 'is', 'a', 'test']`; 23) `['thi', ' i', ' a te', 't']`; 24) `6543`; 25) b; 26) a; 27) b; 28) d.