Strings

This lab covers

- [Understanding strings as sequences of characters]
- [Using basic string operations]
- [Inserting special characters and escape sequences]
- [Converting from objects to strings]
- [Formatting strings]
- [Using the byte type]

Handling text---from user input to filenames to chunks of text to be processed---is a common chore in programming. Python comes with powerful tools to handle and format text. This lab discusses the standard string and string-related operations in Python.

Strings as sequences of characters

For the purposes of extracting characters and substrings, strings can be considered to be sequences of characters, which means that you can use index or slice notation:

```
>>> x = "Hello"
>>> x[0]
'H'
>>> x[-1]
'o'
>>> x[1:]
'ello'
```

One use for slice notation with strings is to chop the newline off the end of a string (usually, a line that's just been read from a file):

```
>>> x = "Goodbye\n"
>>> x = x[:-1]
>>> x
'Goodbye'
```

This code is just an example. You should know that Python strings have other, better methods to strip unwanted characters, but this example illustrates the usefulness of slicing.

You can also determine how many characters are in the string by using the [len] function, just like finding out the number of elements in a list:

```
>>> len("Goodbye")
7
```

But strings aren't lists of characters. The most noticeable difference between strings and lists is that unlike lists, *strings can't be modified*. Attempting to say something like [string.append('c')] or [string[0] = 'H'] results in an error. You'll notice in the previous example that I stripped off the newline from the string by creating a string that was a slice of the previous one, not by modifying the previous string directly. This is a basic Python restriction, imposed for efficiency reasons.

Basic string operations

The simplest (and probably most common) way to combine Python strings is to use the string concatenation operator [+]:

```
>>> x = "Hello " + "World"
>>> x
'Hello World'
```

Python also has an analogous string multiplication operator that I've found to be useful sometimes, but not often:

```
>>> 8 * "x"
'xxxxxxxx'
```

Special characters and escape sequences

You've already seen a few of the character sequences that Python regards as special when used within strings: [\n] represents the newline character, and [\t] represents the tab character. Sequences of characters that start with a backslash and that are used to represent other characters are called *escape sequences*. Escape sequences are generally used to represent *special characters*---that is, characters (such as tab and newline) that don't have a standard one-character printable representation. This section covers escape sequences, special characters, and related topics in more detail.

Numeric (octal and hexadecimal) and Unicode escape sequences

You can include any ASCII character in a string by using an octal (base 8) or hexadecimal (base 16) escape sequence corresponding to that character. An octal escape sequence is a backslash followed by three digits defining an octal number; the ASCII character corresponding to this octal number is substituted for the octal escape sequence. A hexadecimal escape sequence is with [\x] rather than just [\] and can consist of any number of hexadecimal digits. The escape sequence is terminated when a character is found that's not a hexadecimal digit. For example, in the ASCII character table, the character *m* happens to have decimal value 109. This value is octal value 155 and hexadecimal value 6D, so

```
>>> 'm'
'm'
>>> '\155'
'm'
>>> '\x6D'
'm'
```

All three expressions evaluate to a string containing the single character *m*. But these forms can also be used to represent characters that have no printable representation. The newline character [\n], for example, has octal value 012 and hexadecimal value 0A:

```
>>> '\n'
'\n'
>>> '\012'
'\n'
>>> '\x0A'
'\n'
```

Because all strings in Python 3 are Unicode strings, they can also contain almost every character from every language available. Although a discussion of the Unicode system is far beyond the scope of this course, the following examples illustrate that you can also escape any Unicode character, either by number (as shown earlier) or by Unicode name:

```
>>> unicode_a ='\N{LATIN SMALL LETTER A}'
>>> unicode_a
'a'
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> "\u00E1"
'á'
>>>
```

The Unicode character set includes the common ASCII characters ***2***.

Printing vs. evaluating strings with special characters

I talked earlier about the difference between evaluating a Python expression interactively and printing the result of the same expression by using the [print] function. Although the same string is involved, the two operations can produce screen outputs that look different. A string that's evaluated at the top level of an interactive Python session is shown with all of its special characters as octal escape sequences, which makes clear what's in the string. Meanwhile, the [print] function passes the string directly to the terminal program, which may interpret special characters in special ways. Here's what happens with a string consisting of an [a] followed by a newline, a tab, and a [b]:

```
>>> 'a\n\tb'
'a\n\tb'
>>> print('a\n\tb')
a
    b
```

In the first case, the newline and tab are shown explicitly in the string; in the second, they're used as newline and tab characters.

A normal [print] function also adds a newline to the end of the string. Sometimes (that is, when you have lines from files that already end with newlines), you may not want this behavior. Giving the [print] function an [end] parameter of [""] causes the [print] function to not append the newline:

```
>>> print("abc\n")
abc

>>> print("abc\n", end="")
abc
>>>
```

String methods

Most of the Python string methods are built into the standard Python string class, so all string objects have them automatically. The standard [string] module also contains some useful constants. Modules are discussed in detail in [chapter 10].

For the purposes of this section, you need only remember that most string methods are attached to the string object they operate on by a dot ([.]), as in [x.upper()]. That is, they're prepended with the string object followed by a dot. Because strings are immutable, the string methods are used only to obtain their return value and don't modify the string object they're attached to in any way.

I begin with those string operations that are the most useful and most commonly used; then I discuss some less commonly used but still useful operations. At the end of this section, I discuss a few miscellaneous points related to strings. Not all the string methods are documented here. See the documentation for a complete list of string methods.

The split and join string methods

Anyone who works with strings is almost certain to find the [split] and [join] methods invaluable. They're the inverse of one another: [split] returns a list of substrings in the string, and [join] takes a list of strings and puts them together to form a single string with the original string between each element. Typically, [split] uses whitespace as the delimiter to the strings it's splitting, but you can change that behavior via an optional argument.

String concatenation using [+] is useful but not efficient for joining large numbers of strings into a single string, because each time [+] is applied, a new string object is created. The previous "Hello World" example produces two string objects, one of which is immediately discarded. A better option is to use the [join] function:

```
>>> " ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
```

By changing the string used to [join], you can put anything you want between the joined strings:

```
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'
```

You can even use an empty string, [""], to join elements in a list:

```
>>> "".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

The most common use of [split] is probably as a simple parsing mechanism for string-delimited records stored in text files. By default, [split] splits on any whitespace, not just a single space character, but you can also tell it to split on a particular sequence by passing it an optional argument:

```
>>> x = "You\t\t can have tabs\t\n \t and newlines \n\n " \
               "mixed in"
>>> x.split()
['You', 'can', 'have', 'tabs', 'and', 'newlines', 'mixed', 'in']
>>> x = "Mississippi"
>>> x.split("ss")
['Mi', 'i', 'ippi']
```

Sometimes, it's useful to permit the last field in a joined string to contain arbitrary text, perhaps including substrings that may match what [split] splits on when reading in that data. You can do this by specifying how many splits [split] should perform when it's generating its result, via an optional second argument. If you specify *n* splits, [split] goes along the input string until it has performed *n* splits (generating a list with *n*+1 substrings as elements) or until it runs out of string. Here are some examples:

```
>>> x = 'a b c d'
>>> x.split(' ', 1)
['a', 'b c d']
>>> x.split(' ', 2)
['a', 'b', 'c d']
>>> x.split(' ', 9)
['a', 'b', 'c', 'd']
```

When using [split] with its optional second argument, you must supply a first argument. To get it to split on runs of whitespace while using the second argument, use [None] as the first argument.

I use [split] and [join] extensively, usually when working with text files generated by other programs. If you want to create more standard output files from your programs, good choices are the [csv] and [json] modules in the Python standard library.

Quick Check: split and join

How could you use [split] and [join] to change all the whitespace in string x to dashes, such as changing ["this is a test"] to ["this-is-a-test"]?

Converting strings to numbers

You can use the functions [int] and [float] to convert strings to integer or floating-point numbers, respectively. If they're passed a string that can't be interpreted as a number of the given type, these functions raise a [ValueError] exception. Exceptions are explained in [chapter 14].

In addition, you may pass [int] an optional second argument, specifying the numeric base to use when interpreting the input string:

```
>>> float('123.456')
123.456
>>> float('xxyy')
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: could not convert string to float: 'xxyy'
>>> int('3333')
3333
>>> int('123.456')
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 10: '123.456'
>>> int('10000', 8)
4096
>>> int('101', 2)
5
>>> int('ff', 16)
255
>>> int('123456', 6)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 6: '123456'
```

Did you catch the reason for that last error? I requested that the string be interpreted as a base 6 number, but the digit 6 can never appear in a base 6 number. Sneaky!

Quick Check: Strings to Numbers

Which of the following will not be converted to numbers, and why?

```
int('a1')
int('12G', 16)
float("12345678901234567890")
int("12*2")
```

Getting rid of extra whitespace

A trio of surprisingly useful simple methods are the [strip], [lstrip], and [rstrip] functions. [strip] returns a new string that's the same as the original string, except that any whitespace at the *beginning or end* of the string has been removed. [lstrip] and [rstrip] work similarly, except that they remove whitespace only at the left or right end of the original string, respectively:

```
>>> x = "  Hello,    World\t\t "
>>> x.strip()
'Hello,    World'
>>> x.lstrip()
'Hello,    World\t\t '
>>> x.rstrip()
'  Hello,    World'
```

In this example, tab characters are considered to be whitespace. The exact meaning may differ across operating systems, but you can always find out what Python considers to be whitespace by accessing the [string.whitespace] constant. On my Windows system, Python returns the following:

```
>>> import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> " \t\n\r\v\f"
' \t\n\r\x0b\x0c'
```

The characters given in backslashed hex ([\xnn]) format represent the vertical tab and formfeed characters. The space character is in there as itself. It may be tempting to change the value of this variable, to attempt to affect how [strip] and so forth work, but don't do it. Such an action isn't guaranteed to give you the results you're looking for.

But you can change which characters [strip], [rstrip], and [lstrip] remove by passing a string containing the characters to be removed as an extra parameter:

```
>>> x = "www.python.org"
>>> x.strip("w")
'.python.org'
>>> x.strip("gor")
'www.python.'
>>> x.strip(".gorw")
'python'
```

Note that [strip] removes any and all of the characters in the extra parameter string, no matter in which order they occur ***2***.

The most common use for these functions is as a quick way to clean up strings that have just been read in. This technique is particularly helpful when you're reading lines from files (discussed in [chapter 13]), because Python always reads in an entire line, including the trailing newline, if one exists. When you get around to processing the line read in, you typically don't want the trailing newline. [rstrip] is a convenient way to get rid of it.

Quick Check: strip

If the string [x] equals ["(name, date),\n"], which of the following would return a string containing ["name, date"]?

```
x.rstrip("),")
x.strip("),\n")
x.strip("\n)(,")
```

String searching

The string objects provide several methods to perform simple string searches. Before I describe them, though, I'll talk about another module in Python: [re]. (This module is discussed in-depth in [chapter 16]

Another method for searching strings: the re module

The [re] module also does string searching but in a far more flexible manner, using *regular expressions*. Rather than search for a single specified substring, a [re] search can look for a string pattern. You could look for substrings that consist entirely of digits, for example.

Why am I mentioning this when [re] is discussed fully later? In my experience, many uses of basic string searches are inappropriate. You'd benefit from a more powerful searching mechanism but aren't aware that one exists, so you don't even look for something better. Perhaps you have an urgent project involving strings and don't have time to read this entire book. If basic string searching does the job for you, that's great. But be aware that you have a more powerful alternative.

The four basic string-searching methods are similar: [find], [rfind], [index], and [rindex]. A related method, [count], counts how many times a substring can be found in another string. I describe [find] in detail and then examine how the other methods differ from it.

[find] takes one required argument: the substring being searched for. [find] returns the position of the first character of the first instance of [substring] in the [string] object, or [--1] if [substring] doesn't occur in the string:

```
>>> x = "Mississippi"
>>> x.find("ss")
2
>>> x.find("zz")
-1
```

[find] can also take one or two additional, optional arguments. The first of these arguments, if present, is an integer [start]; it causes [find] to ignore all characters before position [start] in [string] when searching for [substring]. The second optional argument, if present, is an integer [end]; it causes [find] to ignore characters at or after position [end] in [string]:

```
>>> x = "Mississippi"
>>> x.find("ss", 3)
5
>>> x.find("ss", 0, 3)
-1
```

[rfind] is almost the same as [find], except that it starts its search at the end of [string] and so returns the position of the first character of the last occurrence of [substring] in [string:]

```
>>> x = "Mississippi"
>>> x.rfind("ss")
5
```

[rfind] can also take one or two optional arguments, with the same meanings as those for [find].

[index] and [rindex] are identical to [find] and [rfind], respectively, except for one difference: If [index] or [rindex] fails to find an occurrence of [substring] in [string], it doesn't return [--1] but raises a [ValueError] exception. Exactly what this means will be clear after you read [chapter 14].

[count] is used identically to any of the previous four functions, but returns the number of non-overlapping times the given substring occurs in the given string:

```
>>> x = "Mississippi"
>>> x.count("ss")
2
```

You can use two other string methods to search strings: [startswith] and [endswith]. These methods return a [True] or [False] result, depending on whether the string they're used on starts or ends with one of the strings given as parameters:

```
>>> x = "Mississippi"
>>> x.startswith("Miss")
True
>>> x.startswith("Mist")
False
>>> x.endswith("pi")
True
>>> x.endswith("p")
False
```

Both [startswith] and [endswith] can look for more than one string at a time. If the parameter is a tuple of strings, both methods check for all the strings in the tuple and return [True] if any one of them is found:

```
>>> x.endswith(("i", "u"))
True
```

[startswith] and [endswith] are useful for simple searches where you're sure that what you're checking for is at the beginning or end of a line.

Quick Check: String searching

If you wanted to check whether a line ends with the string ["rejected"], what string method would you use? Would there be any other ways to get the same result?

Modifying strings

Strings are immutable, but string objects have several methods that can operate on that string and return a new string that's a modified version of the original string. This provides much the same effect as direct modification for most purposes. You can find a more complete description of these methods in the documentation.

You can use the [replace] method to replace occurrences of [substring] (its first argument) in the string with [newstring] (its second argument). This method also takes an optional third argument (see the documentation for details):

```
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

Like the string search functions, the [re] module is a much more powerful method of substring replacement.

The functions [string.maketrans] and [string.translate] may be used together to translate characters in strings into different characters. Although rarely used, these functions can simplify your life when they're needed.

Suppose that you're working on a program that translates string expressions from one computer language into another. The first language uses [~] to mean logical not, whereas the second language uses [!]; the first language uses [^] to mean logical and, the second language uses [&]; the first language uses [(] and [)], whereas the second language uses [[] and []]. In a given string expression, you need to change all instances of [~] to [!], all instances of

[^] to [&], all instances of [(] to [[], and all instances of [)] to []]. You could do this by using multiple invocations of [replace], but an easier and more efficient way is

```
>>> x = "~x ^ (y % z)"
>>> table = x.maketrans("~^()", "!&[]")
>>> x.translate(table)
'!x & [y % z]'
```

The second line uses [maketrans] to make up a translation table from its two string arguments. The two arguments must each contain the same number of characters, and a table is made such that looking up the *n*th character of the first argument in that table gives back the *n*th character of the second argument.

Next, the table produced by [maketrans] is passed to [translate]. Then [translate] goes over each of the characters in its [string] object and checks to see whether they can be found in the table given as the second argument. If a character can be found in the translation table, [translate] replaces that character with the corresponding character looked up in the table to produce the translated string.

You can give [translate] an optional argument to specify characters that should be removed from the string. See the documentation for details.

Other functions in the [string] module perform more specialized tasks. [string.lower] converts all alphabetic characters in a string to lowercase, and [upper] does the opposite. [capitalize] capitalizes the first character of a string, and [title] capitalizes all words in a string. [swapcase] converts lowercase characters to uppercase and uppercase to lowercase in the same string. [expandtabs] gets rid of tab characters in a string by replacing each tab with a specified number of spaces. [ljust], [rjust], and [center] pad a string with spaces to justify it in a certain field width. [zfill] left-pads a numeric string with zeros. Refer to the documentation for details on these methods.

Modifying strings with list manipulations

Because strings are immutable objects, you have no way to manipulate them directly in the same way that you can manipulate lists. Although the operations that produce new strings (leaving the original strings unchanged) are useful for many things, sometimes you want to be able to manipulate a string as though it were a list of characters. In that case, turn the string into a list of characters, do whatever you want, and then turn the resulting list back into a string:

```
>>> text = "Hello, World"
>>> wordList = list(text)
>>> wordList[6:] = []
>>> wordList.reverse()
>>> text = "".join(wordList)
>>> print(text)
,olleH
```

You can also turn a string into a tuple of characters by using the built-in [tuple] function. To turn the list back into a string, use [""].join()].

You shouldn't go overboard with this method because it causes the creation and destruction of new [string] objects, which is relatively expensive. Processing hundreds or thousands of strings in this manner probably won't have much of an impact on your program; processing millions of strings probably will.

Quick Check: Modifying strings

What would be a quick way to change all punctuation in a string to spaces?

Useful methods and constants

[string] objects also have several useful methods to report various characteristics of the string, such as whether it consists of digits or alphabetic characters, or is all uppercase or lowercase:

```
>>> x = "123"
>>> x.isdigit()
True
>>> x.isalpha()
False
>>> x = "M"
>>> x.islower()
False
>>> x.isupper()
True
```

For a list of all the possible string methods, refer to the string section of the official Python documentation.

Finally, the [string] module defines some useful constants. You've already seen [string.whitespace], which is a string made up of the characters Python thinks of as whitespace on your system. [string.digits] is the string ['0123456789']. [string.hexdigits] includes all the characters in [string.digits], as well as ['abcdefABCDEF'], the extra characters used in hexadecimal numbers. [string.octdigits] contains ['01234567']---only those digits used in octal numbers. [string.lowercase] contains all lowercase alphabetic characters; [string.uppercase] contains all uppercase alphabetic characters; [string.letters] contains all the characters in [string.lowercase] and [string.uppercase]. You might be tempted to try assigning to these constants to change the behavior of the language. Python would let you get away with this action, but it probably would be a bad idea.

Try this: String operations

Suppose that you have a list of strings in which some (but not necessarily all) of the strings begin and end with the double quote character:

```
x = ['"abc"', 'def', '"ghi"', '"klm"', 'nop']
```

What code would you use on each element to remove just the double quotes?

What code could you use to find the position of the last [p] in [Mississippi]? When you've found that position, what code would you use to remove just that letter?

Converting from objects to strings

In Python, almost anything can be converted to some sort of a string representation by using the built-in [repr] function. Lists are the only complex Python data types you're familiar with so far, so here, I turn some lists into their representations:

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
>>> x = [1]
>>> x.append(2)
>>> x.append([3, 4])
>>> 'the list x is ' + repr(x)
'the list x is [1, 2, [3, 4]]'
```

The example uses [repr] to convert the list [x] to a string representation, which is then concatenated with the other string to form the final string. Without the use of [repr], this code wouldn't work. In an expression like ["string" + [1, 2] + 3], are you trying to add strings, add lists, or just add numbers? Python doesn't know what you want in such a

circumstance, so it does the safe thing (raises an error) rather than make any assumptions. In the previous example, all the elements had to be converted to string representations before the string concatenation would work.

Lists are the only complex Python objects that I've described to this point, but [repr] can be used to obtain some sort of string representation for almost any Python object. To see this, try [repr] around a built-in complex object, which is an actual Python function:

```
>>> repr(len)
'<built-in function len>'
```

Python hasn't produced a string containing the code that implements the [len] function, but it has at least returned a string---[<built-in function len>]---that describes what that function is. If you keep the [repr] function in mind and try it on each Python data type (dictionaries, tuples, classes, and the like) in the book, you'll see that no matter what type of Python object you have, you can get a string that describes something about that object.

This is great for debugging programs. If you're in doubt about what's held in a variable at a certain point in your program, use [repr] and print out the contents of that variable.

I've covered how Python can convert any object to a string that describes that object. The truth is, Python can do this in either of two ways. The [repr] function always returns what might be loosely called the *formal string representation* of a Python object. More specifically, [repr] returns a string representation of a Python object from which the original object can be rebuilt. For large, complex objects, this may not be the sort of thing you want to see in debugging output or status reports.

Python also provides the built-in [str] function. In contrast to [repr], [str] is intended to produce *printable* string representations, and it can be applied to any Python object. [str] returns what might be called the *informal string representation* of the object. A string returned by [str] need not define an object fully and is intended to be read by humans, not by Python code.

You won't notice any difference between [repr] and [str] when you start using them, because until you begin using the object-oriented features of Python, there's no difference. [str] applied to any built-in Python object always calls [repr] to calculate its result. Only when you start defining your own classes does the difference between [str] and [repr] become important, as discussed in [chapter 15].

So why talk about this now? I want you to be aware that there's more going on behind the scenes with [repr] than just being able to easily write [print] functions for debugging. As a matter of good style, you may want to get into the habit of using [str] rather than [repr] when creating strings for displaying information.

Using the format method

You can format strings in Python 3 in two ways. The newer way is to use the string class's [format] method. The [format] method combines a format string containing replacement fields marked with [{ }] with replacement values taken from the parameters given to the [format] command. If you need to include a literal [{] or [}] in the string, you double it to [{{] or [}}]. The [format] command is a powerful string-formatting mini-language that offers almost endless possibilities for manipulating string formatting. Conversely, it's fairly simple to use for the most common use cases, so I look at a few basic patterns in this section. Then, if you need to use the more advanced options, you can refer to the string-formatting section of the standard library documentation.

The format method and positional parameters

A simple way to use the string [format] method is with numbered replacement fields that correspond to the parameters passed to the [format] function:

```
>>> "{0} is the {1} of {2}".format("Ambrosia", "food", "the gods")     1
'Ambrosia is the food of the gods'
```

```
>>> "{{Ambrosia}} is the {0} of {1}".format("food", "the gods")        2
'{Ambrosia} is the food of the gods'
```

Note that the [format] method is applied to the format string, which can also be a string variable ***1***. Doubling the [{ }] characters escapes them so that they don't mark a replacement field ***2***.

This example has three replacement fields, [{0}], [{1}], and [{2}], which are in turn filled by the first, second, and third parameters. No matter where in the format string you place [{0}], it's always be replaced by the first parameter, and so on.

You can also use the named parameters.

The format method and named parameters

The [format] method also recognizes named parameters and replacement fields:

```
>>> "{food} is the food of {user}".format(food="Ambrosia",
...       user="the gods")
'Ambrosia is the food of the gods'
```

In this case, the replacement parameter is chosen by matching the name of the replacement field with the name of the parameter given to the [format] command.

You can also use both positional and named parameters, and you can even access attributes and elements within those parameters:

```
>>> "{0} is the food of {user[1]}".format("Ambrosia",
...          user=["men", "the gods", "others"])
'Ambrosia is the food of the gods'
```

In this case, the first parameter is positional, and the second, [user[1]], refers to the second element of the named parameter [user].

Format specifiers

Format specifiers let you specify the result of the formatting with even more power and control than the formatting sequences of the older style of string formatting. The format specifier lets you control the fill character, alignment, sign, width, precision, and type of the data when it's substituted for the replacement field. As noted earlier, the syntax of format specifiers is a mini-language in its own right and too complex to cover completely here, but the following examples give you an idea of its usefulness:

```
>>> "{0:10} is the food of gods".format("Ambrosia")                      1
'Ambrosia   is the food of gods'
>>> "{0:{1}} is the food of gods".format("Ambrosia", 10)                 2
'Ambrosia   is the food of gods'
>>> "{food:{width}} is the food of gods".format(food="Ambrosia", width=10)
'Ambrosia   is the food of gods'
>>> "{0:>10} is the food of gods".format("Ambrosia")                     3
'  Ambrosia is the food of gods'
>>> "{0:&>10} is the food of gods".format("Ambrosia")                    4
'&&Ambrosia is the food of gods'
```

:[10] is a format specifier that makes the field 10 spaces wide and pads with spaces ***1***. :[{1}] takes the width from the second parameter ***2***. :[>10] forces right-justification of the field and pads with spaces ***3***. :[&>10] forces right-justification and pads with [&] instead of spaces ***4***.

Quick Check: the format() method

What will be in x when the following snippets of code are executed?:

```
x = "{1:{0}}".format(3, 4)
x = "{0:$>5}".format(3)
x = "{a:{b}}".format(a=1, b=5)
x = "{a:{b}}:{0:$>5}".format(3, 4, a=1, b=5, c=10)
```

Formatting strings with %

This section covers formatting strings with the *string modulus* ([%]) operator. This operator is used to combine Python values into formatted strings for printing or other use. C users will notice a strange similarity to the [printf] family of functions. The use of [%] for string formatting is the old style of string formatting, and I cover it here because it was the standard in earlier versions of Python, and you're likely to see it in code that's been ported from earlier versions of Python or was written by coders who are familiar with those versions. This style of formatting shouldn't be used in new code, however, because it's slated to be deprecated and then removed from the language in the future.

Here's an example:

```
>>> "%s is the %s of %s" % ("Ambrosia", "food", "the gods")
'Ambrosia is the food of the gods'
```

The string modulus operator (the bold [%] that occurs in the middle, not the three instances of [%s] that come before it in the example) takes two parts: the left side, which is a string, and the right side, which is a tuple. The string modulus operator scans the left string for special *formatting sequences* and produces a new string by substituting the values on the right side for those formatting sequences, in order. In this example, the only formatting sequences on the left side are the three instances of [%s], which stands for "Stick a string in here."

Passing in different values on the right side produces different strings:

```
>>> "%s is the %s of %s" % ("Nectar", "drink", "gods")
'Nectar is the drink of gods'
>>> "%s is the %s of the %s" % ("Brussels Sprouts", "food",
...        "foolish")
'Brussels Sprouts is the food of the foolish'
```

The members of the tuple on the right have [str] applied to them automatically by [%s], so they don't have to already be strings:

```
>>> x = [1, 2, "three"]
>>> "The %s contains: %s" % ("list", x)
"The list contains: [1, 2, 'three']"
```

Using formatting sequences

All formatting sequences are substrings contained in the string on the left side of the central [%]. Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is to be accomplished. The [%s] formatting sequence used previously is the simplest formatting sequence; it indicates that the corresponding string from the tuple on the right side of the central [%] should be substituted in place of the [%s].

Other formatting sequences can be more complex. The following sequence specifies the field width (total number of characters) of a printed number to be six, specifies the number of characters after the decimal point to be two, and

left-justifies the number in its field. I've put this formatting sequence in angle brackets so you can see where extra spaces are inserted into the formatted string:

```
>>> "Pi is <%-6.2f>" % 3.14159 # use of the formatting sequence: %-6.2f
'Pi is <3.14  >'
```

All the options for characters that are allowable in formatting sequences are given in the documentation. There are quite a few options, but none is particularly difficult to use. Remember that you can always try a formatting sequence interactively in Python to see whether it does what you expect it to do.

Named parameters and formatting sequences

Finally, one additional feature available with the [%] operator can be useful in certain circumstances. Unfortunately, to describe it, I have to employ a Python feature that I haven't yet discussed in detail: *dictionaries*, commonly called *hash tables* or *associative arrays* in other languages. You can skip ahead to [chapter 7] to learn about dictionaries; skip this section for now and come back to it later; or read straight through, trusting the examples to make things clear.

Formatting sequences can specify what should be substituted for them by name rather than by position. When you do this, each formatting sequence has a name in parentheses immediately following the initial [%] of the formatting sequence, like so:

```
"%(pi).2f"
```

In addition, the argument to the right of the [%] operator is no longer given as a single value or tuple of values to be printed, but as a dictionary of values to be printed, with each named formatting sequence having a correspondingly named key in the dictionary. Using the previous formatting sequence with the string modulus operator, you might produce code like this:

```
>>> num_dict = {'e': 2.718, 'pi': 3.14159}
>>> print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
3.14 - 3.1416 - 2.72
```

This code is particularly useful when you're using format strings that perform a large number of substitutions, because you no longer have to keep track of the positional correspondences of the right-side tuple of elements with the formatting sequences in the format string. The order in which elements are defined in the [dict] argument is irrelevant, and the template string may use values from [dict] more than once (as it does with the ['pi'] entry).

Controlling output with the print function

Python's built-in [print] function also has some options that can make handling simple string output easier. When used with one parameter, [print] prints the value and a newline character, so that a series of calls to [print] prints each value on a separate line:

```
>>> print("a")
a
>>> print("b")
b
```

But [print] can do more. You can also give the [print] function several arguments, and those arguments are printed on the same line, separated by spaces and ending with a newline:

```
>>> print("a", "b", "c")
a b c
```

If that's not quite what you need, you can give the [print] function additional parameters to control what separates each item and what ends the line:

```
>>> print("a", "b", "c", sep="|")
a|b|c
>>> print("a", "b", "c", end="\n\n")
a b c


>>>
```

Finally, the [print] function can be used to print to files as well as console output.

```
>>> print("a", "b", "c", file=open("testfile.txt", "w")
```

Using the [print] function's options gives you enough control for simple text output, but more complex situations are best served by using the [format] method.

Quick Check: Formatting strings with %

What would be in the variable [x] after the following snippets of code have executed?

```
x = "%.2f" % 1.1111
x = "%(a).2f" % {'a':1.1111}
x = "%(a).08f" % {'a':1.1111}
```

String interpolation

Starting in Python 3.6, there's a way to create string constants containing arbitrary values, which is called *string interpolation*. String interpolation is a way to include the values of Python expressions inside literal strings. These f-strings, as they're commonly called because they are prefixed with [f], use a syntax similar to that of the format method, but with a little less overhead. The following examples should give you a basic idea of how f-strings work:

```
>>> value = 42
>>> message = f"The answer is {value}"
>>> print(message)
The answer is 42
```

Just as with the format method, format specifiers may be added:

```
>>> pi = 3.1415
>>> print(f"pi is {pi:{10}.{2}}")
pi is        3.1
```

Because string interpolation is a new feature, it's not yet clear how it will be used. For the complete documentation on f-strings and format specifiers, refer to PEP-498 in the online Python documentation.

Bytes

A [bytes] object is similar to a [string] object but with an important difference: A [string] is an immutable sequence of Unicode characters, whereas a [bytes] object is a sequence of integers with values from 0 to 256. Bytes can be necessary when you're dealing with binary data, such as reading from a binary data file.

The key thing to remember is that [bytes] objects may look like strings, but they can't be used exactly like strings or combined with strings:

```
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> xb = unicode_a_with_acute.encode()         1
>>> xb
b'\xc3\xa1'                                    2
>>> xb += 'A'                                  3
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    xb += 'A'
TypeError: can't concat str to bytes
>>> xb.decode()                                4
'á'
```

The first thing you can see is that to convert from a regular (Unicode) string to [bytes], you need to call the string's [encode] method ***1***. After it's encoded to a [bytes] object, the character is 2 bytes and no longer prints the same way that the string did ***2***. Further, if you attempt to add a [bytes] object and a string object together, you get a type error because the two types are incompatible ***3***. Finally, to convert a [bytes] object back to a string, you need to call that object's [decode] method ***4***.

Most of the time, you shouldn't need to think about Unicode or bytes at all. But when you need to deal with international character sets (an increasingly common issue), you must understand the difference between regular strings and [bytes].

Quick Check: Bytes

For which of the following kinds of data would you want to use a string? For which could you use bytes?

- [***1*** Data file storing binary data]
- [***2*** Text in a language with accented characters]
- [***3*** Text with only uppercase and lowercase roman characters]
- [***4*** A series of integers no larger than 255]

Lab 6: Preprocessing Text

In processing raw text, it's quite often necessary to clean and normalize the text before doing anything else. If you want to find the frequency of words in text, for example, you can make the job easier if, before you start counting, you make sure that everything is lowercase (or uppercase, if you prefer) and that all punctuation has been removed. You can also make things easier by breaking the text into a series of words. In this lab, the task is to read the first part of the first chapter of *Moby Dick* (found in the book's source code), make sure that everything is one case, remove all punctuation, and write the words one per line to a second file. Because I haven't yet covered reading and writing files, here's the code for those operations:

```
with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w") as outfile:
    for line in infile:
        # make all one case
        # remove punctuation
        # split into words
        # write all words for line
        outfile.write(cleaned_words)
```

Summary

- [Python strings have powerful text-processing features, including searching and replacing, trimming characters, and changing case.]
- [Strings are immutable; they can't be changed in place.]

- [Operations that appear to change strings actually return a copy with the changes.]
-