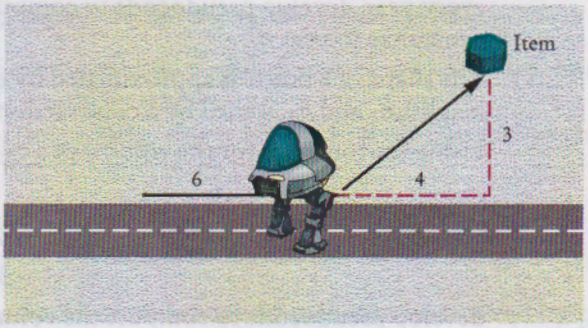
The total time is the time for traversing both segments. The time to traverse the first segment is simply the length of the segment divided by the speed: 6 km divided by 5 km/h, or 1.2 hours.



To compute the time for the second segment, we first need to know its length. It is the hypotenuse of a right triangle with side lengths 3 and 4.

Therefore, its length is $\sqrt{3^2 + 4^2} = 5$. At 2 km/h, it takes 2.5 hours to traverse it. That makes the total travel time 3.7 hours.

This computation gives us enough information to devise an algorithm for the total travel time with arbitrary arguments:

Time for segment 1 = l_1 / s_1
Length of segment 2 = square root of $(dx - l_1)^2 + dy^2$
Time for segment 2 = length of segment 2 / s_2
Total time = time for segment 1 + time for segment 2

Translated into Python, the computations are

```
segment1Time = segment1Length / segment1Speed
segment2Length = sqrt((xDistance - segment1Length) ** 2 + yDistance ** 2)
segment2Time = segment2Length / segment2Speed
totalTime = segment1Time + segment2Time
```

Note that we use variable names that are longer and more descriptive than dx or s_1. When you do hand calculations, it is convenient to use the shorter names, but you should change them to descriptive names in your program.

2.4 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string `"Hello"` is a sequence of five characters.

2.4.1 The String Type

You have already seen strings in print statements such as

```
print("Hello")
```

A string can be stored in a variable

```
greeting = "Hello"
```

and later accessed when needed just as numerical values can be:

```
print(greeting)
```

A string literal denotes a particular string.

A **string literal** denotes a particular string (such as `"Hello"`), just as a number literal (such as 2) denotes a particular number. In Python, string literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.

```
print("This is a string.", 'So is this.')
```

By allowing both types of delimiters, Python makes it easy to include an apostrophe or quotation mark within a string.

```
message = 'He said "Hello"'
```

In this book, we use double quotation marks around strings because this is a common convention in many other programming languages. However, the interactive Python interpreter always displays strings with single quotation marks.

The `len` function returns the number of characters in a string.

The number of characters in a string is called the *length* of the string. For example, the length of `"Harry"` is 5. You can compute the length of a string using Python's `len` function:

```
length = len("World!")    # length is 6
```

A string of length 0 is called the *empty string*. It contains no characters and is written as `""` or `''`.

2.4.2 Concatenation and Repetition

Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.

Given two strings, such as `"Harry"` and `"Morgan"`, you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Python, you use the + operator to concatenate two strings. For example,

```
firstName = "Harry"
lastName = "Morgan"
name = firstName + lastName
```

results in the string

```
"HarryMorgan"
```

What if you'd like the first and last name separated by a space? No problem:

```
name = firstName + " " + lastName
```

This statement concatenates three strings: `firstName`, the string literal `" "`, and `lastName`. The result is

```
"Harry Morgan"
```

When the expression to the left or the right of a + operator is a string, the other one must also be a string or a syntax error will occur. You cannot concatenate a string with a numerical value.

You can also produce a string that is the result of repeating a string multiple times. For example, suppose you need to print a dashed line. Instead of specifying a literal string with 50 dashes, you can use the * operator to create a string that is comprised of the string `"-"` repeated 50 times. For example,

```
dashes = "-" * 50
```

results in the string

```
"----------------------------------------------"
```

A string can be repeated using the * operator.

A string of any length can be repeated using the * operator. For example, the statements

```
message = "Echo..."
print(message * 5)
```

display

```
Echo...Echo...Echo...Echo...Echo...
```

The factor by which the string is replicated must be an integer value. The factor can appear on either side of the * operator, but it is common practice to place the string on the left side and the integer factor on the right.

2.4.3 Converting Between Numbers and Strings

Sometimes it is necessary to convert a numerical value to a string. For example, suppose you need to append a number to the end of a string. You cannot concatenate a string and a number:

```
name = "Agent " + 1729    # Error: Can only concatenate strings
```

Because string concatenation can only be performed between two strings, we must first convert the number to a string.

To produce the string representation of a numerical value, use the `str` function. The statement

```
str(1729)
```

converts the integer value 1729 to the string `"1729"`. The `str` function solves our problem:

```
id = 1729
name = "Agent " + str(id)
```

The `str` function can also be used to convert a floating-point value to a string.

The `int` and `float` functions convert a string containing a number to the numerical value.

Conversely, to turn a string containing a number into a numerical value, use the `int` and `float` functions:

```
id = int("1729")
price = float("17.29")
```

This conversion is important when the strings come from user input (see Section 2.5.1).

The string passed to the `int` or `float` functions can only consist of those characters that comprise a literal value of the indicated type. For example, the statement

```
value = float("17x29")
```

will generate a run-time error because the letter "x" cannot be part of a floating-point literal.

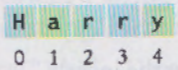
Blank spaces at the front or back will be ignored: `int("  1729  ")` is still 1729.

2.4.4 Strings and Characters

Strings are sequences of **Unicode** characters (see Computing & Society 2.1). You can access the individual characters of a string based on their position within the string. This position is called the *index* of the character.

String positions are counted starting with 0.

The first character has index 0, the second has index 1, and so on.



A string is a sequence of characters.

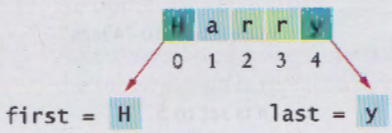
An individual character is accessed using a special subscript notation in which the position is enclosed within square brackets. For example, if the variable `name` is defined as

```
name = "Harry"
```

the statements

```
first = name[0]
last = name[4]
```

extract two different characters from the string. The first statement extracts the first character as the string `"H"` and stores it in variable `first`. The second statement extracts the character at position 4, which in this case is the last character, and stores it in variable `last`.



The index value must be within the valid range of character positions or an "index out of range" exception will be generated at run-time. The `len` function can be used to determine the position of the last index, or the last character in a string.

```
pos = len(name) - 1   # Length of "Harry" is 5
last = name[pos]      # last is set to "y"
```

Initials are formed from the first letter of each name.

The following program puts these concepts to work. The program initializes two variables with strings, one with your name and the other with that of your significant other. It then prints out your initials.

The operation `first[0]` makes a string consisting of one character, taken from the start of `first`. The operation `second[0]` does the same for the second name. Finally, you concatenate the resulting one-character strings with the string literal `"&"` to get a string of length 3, the `initials` string. (See Figure 4.)

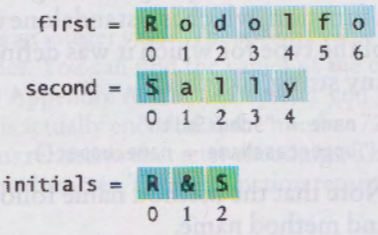


Figure 4
Building the `initials` String

ch02/initials.py

```
1  ##
2  #  This program prints a pair of initials.
3  #
4
```

```
# Set the names of the couple.
first = "Rodolfo"
second = "Sally"

# Compute and display the initials.
initials = first[0] + "&" + second[0]
print(initials)
```

Table 7 String Operations

Statement	Result	Comment
`string = "Py"` `string = string + "thon"`	`string` is set to `"Python"`	When applied to strings, + denotes concatenation.
`print("Please" +` `"enter your name: ")`	Prints `Please enter your name:`	Use concatenation to break up strings that don't fit into one line.
`team = str(49) + "ers"`	`team` is set to `"49ers"`	Because 49 is an integer, it must be converted to a string.
`greeting = "H & S"` `n = len(greeting)`	`n` is set to 5	Each space counts as one character.
`string = "Sally"` `ch = string[1]`	`ch` is set to `"a"`	Note that the initial position is 0.
`last = string[len(string) - 1]`	`last` is set to the string containing the last character in `string`	The last character has position `len(string) - 1`.

2.4.5 String Methods

In computer programming, an **object** is a software entity that represents a value with certain behavior. The value can be simple, such as a string, or complex, like a graphical window or data file. You will learn much more about objects in Chapter 9. For now, you need to master a small amount of notation for working with string objects.

The behavior of an object is given through its **methods**. A method, like a function, is a collection of programming instructions that carry out a particular task. But unlike a function, which is a standalone operation, a method can only be applied to an object of the type for which it was defined. For example, you can apply the `upper` method to any string, like this:

```
name = "John Smith"
uppercaseName = name.upper()   # Sets uppercaseName to "JOHN SMITH"
```

Note that the method name follows the object, and that a dot (.) separates the object and method name.

There is another string method called `lower` that yields the lowercase version of a string:

```
print(name.lower())   # Prints john smith
```

It is a bit arbitrary when you need to call a function (such as `len(name)`) and when you need to call a method (`name.lower()`). You will simply need to remember or look it up.

Just like function calls, method calls can have arguments. For example, the string method `replace` creates a new string in which every occurrence of a given substring is replaced with a second string. Here is a call to that method with two arguments:

```
name2 = name.replace("John", "Jane")   # Sets name2 to "Jane Smith"
```

Note that none of the method calls change the contents of the string on which they are invoked. After the call `name.upper()`, the `name` variable still holds `"John Smith"`. The method call returns the uppercase version. Similarly, the `replace` method returns a new string with the replacements, without modifying the original.

Table 8 lists the string methods introduced in this section.

Table 8 Useful String Methods

Method	Returns
`s.lower()`	A lowercase version of string *s*.
`s.upper()`	An uppercase version of *s*.
`s.replace(old, new)`	A new version of string *s* in which every occurrence of the substring *old* is replaced by the string *new*.

18. What is the length of the string `"Python Program"`?

19. Given this string variable, give a method call that returns the string `"gram"`.

```
title = "Python Program"
```

20. Use string concatenation to turn the string variable `title` from Self Check 19 into `"Python Programming"`.

21. What does the following statement sequence print?

```
string = "Harry"
n = len(string)
mystery = string[0] + string[n - 1]
print(mystery)
```

Practice It Now you can try these exercises at the end of the chapter: R2.7, R2.11, P2.15, P2.22.

Special Topic 2.4

Character Values

A character is stored internally as an integer value. The specific value used for a given character is based on a standard set of codes. You can find the values of the characters that are used in Western European languages in Appendix A. For example, if you look up the value for the character `"H"`, you can see that it is actually encoded as the number 72.

Python provides two functions related to character encodings. The `ord` function returns the number used to represent a given character. The `chr` function returns the character associated with a given code. For example,

```
print("The letter H has a code of", ord("H"))
print("Code 97 represents the character", chr(97))
```

produces the following output

```
The letter H has a code of 72
Code 97 represents the character a
```

Special Topic 2.5

Escape Sequences

To include a quotation mark in a literal string, precede it with a backslash (\), like this:

```
"He said \"Hello\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence \" is called an **escape sequence**.

To include a backslash in a string, use the escape sequence \\, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is \n, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
print("*\n**\n***")
```

prints the characters

```
*
**
***
```

on three separate lines.

Computing & Society 2.1 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, ä, ö, ü, and a *double-s* character ß. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.

The German Keyboard Layout

Many countries don't use the Roman script at all. Russian, Greek, Hebrew, Arabic, and Thai letters, to name just a few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has about as many characters as the English alphabet.

Hebrew, Arabic, and English

The Chinese languages as well as Japanese and Korean use Chinese characters. Each character represents an idea or thing. Words are made up of one or more of these ideographic characters. Over 70,000 ideographs are known.

Starting in 1988, a consortium of hardware and software manufacturers developed a uniform encoding scheme called Unicode that is capable of encoding text in essentially all written languages of the world.

Today Unicode defines over 100,000 characters. There are even plans to add codes for extinct languages, such as Egyptian hieroglyphics.

The Chinese Script