9. Strings

9.1. Strings Revisited

9.2. A Collection Data Type

9.3. Operations on Strings

9.4. Index Operator: Working with the Characters of a String

9.5. String Methods

9.5.1. String Format Method

9.6. Length

9.7. The Slice Operator

9.8. String Comparison

9.9. Strings are Immutable

9.10. Traversal and the for Loop: By Item

9.11. Traversal and the for Loop: By Index

9.12. Traversal and the while Loop

9.13. The in and not in operators

9.14. The Accumulator Pattern with Strings

9.16. Looping and Counting

9.17. A find function

9.18. Optional parameters

9.19. Character classification

9.20. Summary

9.21. Glossary

9.22. Exercises

9.1. Strings Revisited

Throughout the first chapters of this book we have used strings to represent words or phrases that we wanted to print out. Our definition was simple: a string is simply some characters inside quotes. In this chapter we explore strings in much more detail.

9.2. A Collection Data Type

So far we have seen built-in types like: int, float, bool, str and we’ve seen lists. int, float, and bool are considered to be simple or primitive data types because their values are not composed of any smaller parts. They cannot be broken down. On the other hand, strings and lists are different from the others because they are made up of smaller pieces. In the case of strings, they are made up of smaller strings each containing one character.

Types that are comprised of smaller pieces are called collection data types. Depending on what we are doing, we may want to treat a collection data type as a single entity (the whole), or we may want to access its parts. This ambiguity is useful.

Strings can be defined as sequential collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right.

A string that contains no characters, often referred to as the empty string, is still considered to be a string. It is simply a sequence of zero characters and is represented by ‘’ or “” (two single or two double quotes with nothing in between).

9.3. Operations on Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

message - 1

"Hello" / 123

message \* "Hello"

"15" + 2

Interestingly, the + operator does work with strings, but for strings, the + operator represents concatenation, not addition. Concatenation means joining the two operands by linking them end-to-end. For example:

1 fruit = "banana"

2 bakedGood = " nut bread"

3 print(fruit + bakedGood)

4

​

The output of this program is banana nut bread. The space before the word nut is part of the string and is necessary to produce the space between the concatenated strings. Take out the space and run it again.

The \* operator also works on strings. It performs repetition. For example, 'Fun'\*3 is 'FunFunFun'. One of the operands has to be a string and the other has to be an integer.

1 print("Go" \* 6)

2

​3 name = "Packers"

4 print(name \* 3)

5

​6 print(name + "Go" \* 3)

7

​8 print((name + "Go") \* 3)

9

​

Activity: 9.3.2 ActiveCode (ch08\_mult)

This interpretation of + and \* makes sense by analogy with addition and multiplication. Just as 4\*3 is equivalent to 4+4+4, we expect "Go"\*3 to be the same as "Go"+"Go"+"Go", and it is. Note also in the last example that the order of operations for \* and + is the same as it was for arithmetic. The repetition is done before the concatenation. If you want to cause the concatenation to be done first, you will need to use parenthesis.

Check your understanding

strings-3-3: What is printed by the following statements?

s = "python"

t = "rocks"

print(s + t)

A. python rocks

B. python

C. pythonrocks

D. Error, you cannot add two strings together.

strings-3-4: What is printed by the following statements?

s = "python"

excl = "!"

print(s+excl\*3)

A. python!!!

B. python!python!python!

C. pythonpythonpython!

D. Error, you cannot perform concatenation and repetition at the same time.

9.4. Index Operator: Working with the Characters of a String

The indexing operator (Python uses square brackets to enclose the index) selects a single character from a string. The characters are accessed by their position or index value. For example, in the string shown below, the 14 characters are indexed left to right from position 0 to position 13.

index values

It is also the case that the positions are named from right to left using negative numbers where -1 is the rightmost index and so on. Note that the character at index 6 (or -8) is the blank character.

1 school = "Luther College"

2 m = school[2]

3 print(m)

4

​5 lastchar = school[-1]

6 print(lastchar)

7

​

The expression school[2] selects the character at index 2 from school, and creates a new string containing just this one character. The variable m refers to the result.

Remember that computer scientists often start counting from zero. The letter at index zero of "Luther College" is L. So at position [2] we have the letter t.

If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. Give it a try.

The expression in brackets is called an index. An index specifies a member of an ordered collection. In this case the collection of characters in the string. The index indicates which character you want. It can be any integer expression so long as it evaluates to a valid index value.

Note that indexing returns a string — Python has no special type for a single character. It is just a string of length 1.

Check your understanding

strings-4-2: What is printed by the following statements?

s = "python rocks"

print(s[3])

A. t

B. h

C. c

D. Error, you cannot use the [ ] operator with a string.

strings-4-3: What is printed by the following statements?

s = "python rocks"

print(s[2] + s[-5])

A. tr

B. ps

C. nn

D. Error, you cannot use the [ ] operator with the + operator.

9.5. String Methods

We previously saw that each turtle instance has its own attributes and a number of methods that can be applied to the instance. For example, we wrote tess.right(90) when we wanted the turtle object tess to perform the right method to turn to the right 90 degrees. The “dot notation” is the way we connect the name of an object to the name of a method it can perform.

Strings are also objects. Each string instance has its own attributes and methods. The most important attribute of the string is the collection of characters. There are a wide variety of methods. Try the following program.

1 ss = "Hello, World"

2 print(ss.upper())

3

​4 tt = ss.lower()

5 print(tt)

6

​

In this example, upper is a method that can be invoked on any string object to create a new string in which all the characters are in uppercase. lower works in a similar fashion changing all characters in the string to lowercase. (The original string ss remains unchanged. A new string tt is created.)

In addition to upper and lower, the following table provides a summary of some other useful string methods. There are a few code examples that follow so that you can try them out.

Method

Parameters

Description

upper

none

Returns a string in all uppercase

lower

none

Returns a string in all lowercase

capitalize

none

Returns a string with first character capitalized, the rest lower

strip

none

Returns a string with the leading and trailing whitespace removed

lstrip

none

Returns a string with the leading whitespace removed

rstrip

none

Returns a string with the trailing whitespace removed

count

item

Returns the number of occurrences of item

replace

old, new

Replaces all occurrences of old substring with new

center

width

Returns a string centered in a field of width spaces

ljust

width

Returns a string left justified in a field of width spaces

rjust

width

Returns a string right justified in a field of width spaces

find

item

Returns the leftmost index where the substring item is found, or -1 if not found

rfind

item

Returns the rightmost index where the substring item is found, or -1 if not found

index

item

Like find except causes a runtime error if item is not found

rindex

item

Like rfind except causes a runtime error if item is not found

format

substitutions

Involved! See String Format Method, below

You should experiment with these methods so that you understand what they do. Note once again that the methods that return strings do not change the original. You can also consult the Python documentation for strings.

1 ss = " Hello, World "

2

​3 els = ss.count("l")

4 print(els)

5

​6 print("\*\*\*" + ss.strip() + "\*\*\*")

7 print("\*\*\*" + ss.lstrip() + "\*\*\*")

8 print("\*\*\*" + ss.rstrip() + "\*\*\*")

9

​10 news = ss.replace("o", "\*\*\*")

11 print(news)

12

​

1 food = "banana bread"

2 print(food.capitalize())

3

​4 print("\*" + food.center(25) + "\*")

5 print("\*" + food.ljust(25) + "\*") # stars added to show bounds

6 print("\*" + food.rjust(25) + "\*")

7

​8 print(food.find("e"))

9 print(food.find("na"))

10 print(food.find("b"))

11

​12 print(food.rfind("e"))

13 print(food.rfind("na"))

14 print(food.rfind("b"))

15

​16 print(food.index("e"))

17

Check your understanding

strings-5-4: What is printed by the following statements?

s = "python rocks"

print(s.count("o") + s.count("p"))

A. 0

B. 2

C. 3

strings-5-5: What is printed by the following statements?

s = "python rocks"

print(s[1] \* s.index("n"))

A. yyyyy

B. 55555

C. n

D. Error, you cannot combine all those things together.

9.5.1. String Format Method

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,

Hello \_\_\_\_\_!

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. We use this as an analogy: Python has a similar construction, better called fill-in-the-braces. The string method format, makes substitutions into places in a string enclosed in braces. Run this code:

1 person = input('Your name: ')

2 greeting = 'Hello {}!'.format(person)

3 print(greeting)

4

​

There are several new ideas here!

The string for the format method has a special form, with braces embedded. Such a string is called a format string. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method. There are many variations on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier greeting, and then the string is printed.

The identifier greeting was introduced to break the operations into a clearer sequence of steps. However, since the value of greeting is only referenced once, it can be eliminated with the more concise version:

1 person = input('Enter your name: ')

2 print('Hello {}!'.format(person))

3

There can be multiple substitutions, with data of any type. Next we use floats. Try original price $2.50 with a 7% discount:

1 origPrice = float(input('Enter the original price: $'))

2 discount = float(input('Enter discount percentage: '))

3 newPrice = (1 - discount/100)\*origPrice

4 calculation = '${} discounted by {}% is ${}.'.format(origPrice, discount, newPrice)

5 print(calculation)

6

The parameters are inserted into the braces in order.

If you used the data suggested, this result is not satisfying. Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.

Format strings can give further information inside the braces showing how to specially format data. In particular floats can be shown with a specific number of decimal places. For two decimal places, put :.2f inside the braces for the monetary values:

1 origPrice = float(input('Enter the original price: $'))

2 discount = float(input('Enter discount percentage: '))

3 newPrice = (1 - discount/100)\*origPrice

4 calculation = '${:.2f} discounted by {}% is ${:.2f}.'.format(origPrice, discount, newPrice)

5 print(calculation)

6

The 2 in the format modifier can be replaced by another integer to round to that specified number of digits.

This kind of format string depends directly on the order of the parameters to the format method. There are other approaches that we will skip here, explicitly numbering substitutions and taking substitutions from a dictionary.

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final formatted string. The rule is to double the braces: { { and }}. For example mathematical set notation uses braces. The initial and final doubled braces in the format string below generate literal braces in the formatted string:

a = 5

b = 9

setStr = 'The set is { { { }, {} } }.'.format(a, b)

print(setStr)

strings-5-10: What is printed by the following statements?

x = 2

y = 6

print('sum of {} and {} is {}; product: {}.'.format( x, y, x+y, x\*y))

A. Nothing - it causes an error

B. sum of {} and {} is {}; product: {}. 2 6 8 12

C. sum of 2 and 6 is 8; product: 12.

D. sum of {2} and {6} is {8}; product: {12}.

strings-5-11: What is printed by the following statements?

v = 2.34567

print('{:.1f} {:.2f} {:.7f}'.format(v, v, v))

A. 2.34567 2.34567 2.34567

B. 2.3 2.34 2.34567

C. 2.3 2.35 2.3456700

9.6. Length

The len function, when applied to a string, returns the number of characters in a string.

1 fruit = "Banana"

2 print(len(fruit))

3

​

To get the last letter of a string, you might be tempted to try something like this:

1 fruit = "Banana"

2 sz = len(fruit)

3 last = fruit[sz] # ERROR!

4 print(last)

5

​

That won’t work. It causes the runtime error IndexError: string index out of range. The reason is that there is no letter at index position 6 in "Banana". Since we started counting at zero, the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length. Give it a try in the example above.

1 fruit = "Banana"

2 sz = len(fruit)

3 lastch = fruit[sz-1]

4 print(lastch)

5

​

Alternatively in Python, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on. Try it! Most other languages do not allow the negative indices, but they are a handy feature of Python!

Check your understanding

strings-6-4: What is printed by the following statements?

s = "python rocks"

print(len(s))

A. 11

B. 12

strings-6-5: What is printed by the following statements?

s = "python rocks"

print(s[len(s)-5])

A. o

B. r

C. s

D. Error, len(s) is 12 and there is no index 12.

strings-6-6: What is printed by the following statements?

s = "python rocks"

print(s[-3])

A. c

B. k

C. s

D. Error, negative indices are illegal.

9.7. The Slice Operator

A substring of a string is called a slice. Selecting a slice is similar to selecting a character:

1 singers = "Peter, Paul, and Mary"

2 print(singers[0:5])

3 print(singers[7:11])

4 print(singers[17:21])

5

The slice operator [n:m] returns the part of the string from the n’th character to the m’th character, including the first but excluding the last. In other words, start with the character at index n and go up to but do not include the character at index m. This behavior may seem counter-intuitive but if you recall the range function, it did not include its end point either.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

There is no Index Out Of Range exception for a slice. A slice is forgiving and shifts any offending index to something legal.

1 fruit = "banana"

2 print(fruit[:3])

3 print(fruit[3:])

4 print(fruit[3:-10])

5 print(fruit[3:99])

6

​

What do you think fruit[:] means?

Check your understanding

strings-7-3: What is printed by the following statements?

s = "python rocks"

print(s[3:8])

A. python

B. rocks

C. hon r

D. Error, you cannot have two numbers inside the [ ].

strings-7-4: What is printed by the following statements?

s = "python rocks"

print(s[7:11] \* 3)

A. rockrockrock

B. rock rock rock

C. rocksrocksrocks

D. Error, you cannot use repetition with slicing.

9.8. String Comparison

The comparison operators also work on strings. To see if two strings are equal you simply write a boolean expression using the equality operator.

1 word = "banana"

2 if word == "banana":

3 print("Yes, we have bananas!")

4 else:

5 print("Yes, we have NO bananas!")

6

Other comparison operations are useful for putting words in lexicographical order. This is similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

1 word = "zebra"

2

​3 if word < "banana":

4 print("Your word, " + word + ", comes before banana.")

5 elif word > "banana":

6 print("Your word, " + word + ", comes after banana.")

7 else:

8 print("Yes, we have no bananas!")

9

It is probably clear to you that the word apple would be less than (come before) the word banana. After all, a is before b in the alphabet. But what if we consider the words apple and Apple? Are they the same?

1 print("apple" < "banana")

2

​3 print("apple" == "Apple")

4 print("apple" < "Apple")

5

It turns out, as you recall from our discussion of variable names, that uppercase and lowercase letters are considered to be different from one another. The way the computer knows they are different is that each character is assigned a unique integer value. “A” is 65, “B” is 66, and “5” is 53. The way you can find out the so-called ordinal value for a given character is to use a character function called ord.

1 print(ord("A"))

2 print(ord("B"))

3 print(ord("5"))

4

​5 print(ord("a"))

6 print("apple" > "Apple")

7

​

When you compare characters or strings to one another, Python converts the characters into their equivalent ordinal values and compares the integers from left to right. As you can see from the example above, “a” is greater than “A” so “apple” is greater than “Apple”.

Humans commonly ignore capitalization when comparing two words. However, computers do not. A common way to address this issue is to convert strings to a standard format, such as all lowercase, before performing the comparison.

There is also a similar function called chr that converts integers into their character equivalent.

1 print(chr(65))

2 print(chr(66))

3

​4 print(chr(49))

5 print(chr(53))

6

​7 print("The character for 32 is", chr(32), "!!!")

8 print(ord(" "))

9

​

One thing to note in the last two examples is the fact that the space character has an ordinal value (32). Even though you don’t see it, it is an actual character. We sometimes call it a nonprinting character.

Check your understanding

strings-8-6: Evaluate the following comparison:

"Dog" < "Doghouse"

A. True

B. False

strings-8-7: Evaluate the following comparison:

"dog" < "Dog"

A. True

B. False

C. They are the same word

strings-8-8: Evaluate the following comparison:

"dog" < "Doghouse"

A. True

B. False

9.9. Strings are Immutable

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example, in the following code, we would like to change the first letter of greeting.

1 greeting = "Hello, world!"

2 greeting[0] = 'J' # ERROR!

3 print(greeting)

4

Instead of producing the output Jello, world!, this code produces the runtime error TypeError: 'str' object does not support item assignment.

Strings are immutable, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

1 greeting = "Hello, world!"

2 newGreeting = 'J' + greeting[1:]

3 print(newGreeting)

4 print(greeting) # same as it was

5

The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

Check your understanding

strings-9-3: What is printed by the following statements:

s = "Ball"

s[0] = "C"

print(s)

A. Ball

B. Call

C. Error

9.10. Traversal and the for Loop: By Item

A lot of computations involve processing a collection one item at a time. For strings this means that we would like to process one character at a time. Often we start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal.

We have previously seen that the for statement can iterate over the items of a sequence (a list of names in the case below).

1 for aname in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:

2 invitation = "Hi " + aname + ". Please come to my party on Saturday!"

3 print(invitation)

4

Recall that the loop variable takes on each value in the sequence of names. The body is performed once for each name. The same was true for the sequence of integers created by the range function.

1 for avalue in range(10):

2 print(avalue)

3

Since a string is simply a sequence of characters, the for loop iterates over each character automatically.

1 for achar in "Go Spot Go":

2 print(achar)

3

​

The loop variable achar is automatically reassigned each character in the string “Go Spot Go”. We will refer to this type of sequence iteration as iteration by item. Note that it is only possible to process the characters one at a time from left to right.

Check your understanding

strings-10-4: How many times is the word HELLO printed by the following statements?

s = "python rocks"

for ch in s:

print("HELLO")

A. 10

B. 11

C. 12

D. Error, the for statement needs to use the range function.

strings-10-5: How many times is the word HELLO printed by the following statements?

s = "python rocks"

for ch in s[3:8]:

print("HELLO")

A. 4

B. 5

C. 6

D. Error, the for statement cannot use slice.

9.11. Traversal and the for Loop: By Index

It is also possible to use the range function to systematically generate the indices of the characters. The for loop can then be used to iterate over these positions. These positions can be used together with the indexing operator to access the individual characters in the string.

Consider the following codelens example.

1 fruit = "apple"

2 for idx in range(5):

3 currentChar = fruit[idx]

4 print(currentChar)

The index positions in “apple” are 0,1,2,3 and 4. This is exactly the same sequence of integers returned by range(5). The first time through the for loop, idx will be 0 and the “a” will be printed. Then, idx will be reassigned to 1 and “p” will be displayed. This will repeat for all the range values up to but not including 5. Since “e” has index 4, this will be exactly right to show all of the characters.

In order to make the iteration more general, we can use the len function to provide the bound for range. This is a very common pattern for traversing any sequence by position. Make sure you understand why the range function behaves correctly when using len of the string as its parameter value.

1 fruit = "apple"

2 for idx in range(len(fruit)):

3 print(fruit[idx])

4

You may also note that iteration by position allows the programmer to control the direction of the traversal by changing the sequence of index values. Recall that we can create ranges that count down as well as up so the following code will print the characters from right to left.

Check your understanding

strings-11-4: How many times is the letter o printed by the following statements?

s = "python rocks"

for idx in range(len(s)):

if idx % 2 == 0:

print(s[idx])

A. 0

B. 1

C. 2

D. Error, the for statement cannot have an if inside.

9.12. Traversal and the while Loop

The while loop can also control the generation of the index values. Remember that the programmer is responsible for setting up the initial condition, making sure that the condition is correct, and making sure that something changes inside the body to guarantee that the condition will eventually fail.

1 fruit = "apple"

2

​3 position = 0

4 while position < len(fruit):

5 print(fruit[position])

6 position = position + 1

7

The loop condition is position < len(fruit), so when position is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

Check your understanding

strings-12-3: How many times is the letter o printed by the following statements?

s = "python rocks"

idx = 1

while idx < len(s):

print(s[idx])

idx = idx + 2

A. 0

B. 1

C. 2

9.13. The in and not in operators

The in operator tests if one string is a substring of another:

1 print('p' in 'apple')

2 print('i' in 'apple')

3 print('ap' in 'apple')

4 print('pa' in 'apple')

5

​

Note that a string is a substring of itself, and the empty string is a substring of any other string. (Also note that computer scientists like to think about these edge cases quite carefully!)

1 print('a' in 'a')

2 print('apple' in 'apple')

3 print('' in 'a')

4 print('' in 'apple')

5

​

Activity: 9.13.2 ActiveCode (chp8\_in2)

The not in operator returns the logical opposite result of in.

1 print('x' not in 'apple')

2

​

9.14. The Accumulator Pattern with Strings

Combining the in operator with string concatenation using + and the accumulator pattern, we can write a function that removes all the vowels from a string. The idea is to start with a string and iterate over each character, checking to see if the character is a vowel. As we process the characters, we will build up a new string consisting of only the non-vowel characters. To do this, we use the accumulator pattern.

Remember that the accumulator pattern allows us to keep a “running total”. With strings, we are not accumulating a numeric total. Instead we are accumulating characters onto a string.

1 def removeVowels(s):

2 vowels = "aeiouAEIOU"

3 sWithoutVowels = ""

4 for eachChar in s:

5 if eachChar not in vowels:

6 sWithoutVowels = sWithoutVowels + eachChar

7 return sWithoutVowels

8

​9 print(removeVowels("compsci"))

10 print(removeVowels("aAbEefIijOopUus"))

11

Line 5 uses the not in operator to check whether the current character is not in the string vowels. The alternative to using this operator would be to write a very large if statement that checks each of the individual vowel characters. Note we would need to use logical and to be sure that the character is not any of the vowels.

if eachChar != 'a' and eachChar != 'e' and eachChar != 'i' and

eachChar != 'o' and eachChar != 'u' and eachChar != 'A' and

eachChar != 'E' and eachChar != 'I' and eachChar != 'O' and

eachChar != 'U':

sWithoutVowels = sWithoutVowels + eachChar

Look carefully at line 6 in the above program (sWithoutVowels = sWithoutVowels + eachChar). We will do this for every character that is not a vowel. This should look very familiar. As we were describing earlier, it is an example of the accumulator pattern, this time using a string to “accumulate” the final result. In words it says that the new value of sWithoutVowels will be the old value of sWithoutVowels concatenated with the value of eachChar. We are building the result string character by character.

Take a close look also at the initialization of sWithoutVowels. We start with an empty string and then begin adding new characters to the end.

Check your understanding

strings-14-3: What is printed by the following statements:

s = "ball"

r = ""

for item in s:

r = item.upper() + r

print(r)

A. Ball

B. BALL

C. LLAB

9.16. Looping and Counting

We will finish this chapter with a few more examples that show variations on the theme of iteration through the characters of a string. We will implement a few of the methods that we described earlier to show how they can be done.

The following program counts the number of times a particular letter, aChar, appears in a string. It is another example of the accumulator pattern that we have seen in previous chapters.

1 def count(text, aChar):

2 lettercount = 0

3 for c in text:

4 if c == aChar:

5 lettercount = lettercount + 1

6 return lettercount

7

​8 print(count("banana","a"))

9

​

The function count takes a string as its parameter. The for statement iterates through each character in the string and checks to see if the character is equal to the value of aChar. If so, the counting variable, lettercount, is incremented by one. When all characters have been processed, the lettercount is returned.

9.17. A find function

Here is an implementation for a restricted find method, where the target is a single character.

1 def find(astring, achar):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = 0

7 found = False

8 while ix < len(astring) and not found:

9 i f astring[ix] == achar:

10 found = True

11 else:

12 ix = ix + 1

13 if found:

14 return ix

15 else:

16 return -1

17

​18 print(find("Compsci", "p"))

19 print(find("Compsci", "C"))

20 print(find("Compsci", "i"))

21 print(find("Compsci", "x"))

22

​

In a sense, find is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears for the first time. If the character is not found, the function returns -1.

The while loop in this example uses a slightly more complex condition than we have seen in previous programs. Here there are two parts to the condition. We want to keep going if there are more characters to look through and we want to keep going if we have not found what we are looking for. The variable found is a boolean variable that keeps track of whether we have found the character we are searching for. It is initialized to False. If we find the character, we reassign found to True.

The other part of the condition is the same as we used previously to traverse the characters of the string. Since we have now combined these two parts with a logical and, it is necessary for them both to be True to continue iterating. If one part fails, the condition fails and the iteration stops.

When the iteration stops, we must ask a question to find out the individual condition that caused the termination, and then return the proper value. This is a pattern for dealing with while loops with compound conditions.

Note

This pattern of computation is sometimes called a eureka traversal because as soon as we find what we are looking for, we can cry Eureka! and stop looking. The way we stop looking is by setting found to True which causes the condition to fail.

9.18. Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the find function, adding a third parameter for the starting position in the search string:

1 def find2(astring, achar, start):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = start

7 found = False

8 while ix < len(astring) and not found:

9 if astring[ix] == achar:

10 found = True

11 else:

12 ix = ix + 1

13 if found:

14 return ix

15 else:

16 return -1

17

​18 print(find2('banana', 'a', 2))

19

​

The call find2('banana', 'a', 2) now returns 3, the index of the first occurrence of ‘a’ in ‘banana’ after index 2. What does find2('banana', 'n', 3) return? If you said, 4, there is a good chance you understand how find2 works. Try it.

Better still, we can combine find and find2 using an optional parameter.

1 def find3(astring, achar, start=0):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = start

7 found = False

8 while ix < len(astring) and not found:

9 if astring[ix] == achar:

10 found = True

11 else:

12 ix = ix + 1

13 if found:

14 return ix

15 else:

16 return -1

17

​18 print(find3('banana', 'a', 2))

19

​

The call find3('banana', 'a', 2) to this version of find behaves just like find2, while in the call find3('banana', 'a'), start will be set to the default value of 0.

Adding another optional parameter to find makes it search from a starting position, up to but not including the end position.

1 def find4(astring, achar, start=0, end=None):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = start

7 if end == None:

8 end = len(astring)

9

​10 found = False

11 while ix < end and not found:

12 if astring[ix] == achar:

13 found = True

14 else:

15 ix = ix + 1

16 if found:

17 return ix

18 else:

19 return -1

20

​21 ss = "Python strings have some interesting methods."

22

​23 print(find4(ss, 's'))

24 print(find4(ss, 's', 7))

The optional value for end is interesting. We give it a default value None if the caller does not supply any argument. In the body of the function we test what end is and if the caller did not supply any argument, we reassign end to be the length of the string. If the caller has supplied an argument for end, however, the caller’s value will be used in the loop.

The semantics of start and end in this function are precisely the same as they are in the range function.

9.18. Optional parameters

To find the locations of the second or third occurrence of a character in a string, we can modify the find function, adding a third parameter for the starting position in the search string:

1 def find2(astring, achar, start):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = start

7 found = False

8 while ix < len(astring) and not found:

9 if astring[ix] == achar:

10 found = True

11 else:

12 ix = ix + 1

13 if found:

14 return ix

15 else:

16 return -1

17

​18 print(find2('banana', 'a', 2))

19

​

Activity: 9.18.1 ActiveCode (ch08\_fun4)

The call find2('banana', 'a', 2) now returns 3, the index of the first occurrence of ‘a’ in ‘banana’ after index 2. What does find2('banana', 'n', 3) return? If you said, 4, there is a good chance you understand how find2 works. Try it.

Better still, we can combine find and find2 using an optional parameter.

1 def find3(astring, achar, start=0):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 x = start

7 found = False

8 while ix < len(astring) and not found:

9 if astring[ix] == achar:

10 found = True

11 else:

12 ix = ix + 1

13 if found:

14 return ix

15 else:

16 return -1

17

​18 print(find3('banana', 'a', 2))

19

The call find3('banana', 'a', 2) to this version of find behaves just like find2, while in the call find3('banana', 'a'), start will be set to the default value of 0.

Adding another optional parameter to find makes it search from a starting position, up to but not including the end position.

1 def find4(astring, achar, start=0, end=None):

2 """

3 Find and return the index of achar in astring.

4 Return -1 if achar does not occur in astring.

5 """

6 ix = start

7 if end == None:

8 end = len(astring)

9

​10 found = False

11 while ix < end and not found:

12 if astring[ix] == achar:

13 found = True

14 else:

15 ix = ix + 1

16 if found:

17 return ix

18 else:

19 return -1

20

​21 ss = "Python strings have some interesting methods."

22

​

23 print(find4(ss, 's'))

24 print(find4(ss, 's', 7))

The optional value for end is interesting. We give it a default value None if the caller does not supply any argument. In the body of the function we test what end is and if the caller did not supply any argument, we reassign end to be the length of the string. If the caller has supplied an argument for end, however, the caller’s value will be used in the loop.

The semantics of start and end in this function are precisely the same as they are in the range function.

9.20. Summary

This chapter introduced a lot of new ideas. The following summary may prove helpful in remembering what you learned.

indexing ([])

Access a single character in a string using its position (starting from 0). Example: 'This'[2] evaluates to 'i'.

length function (len)

Returns the number of characters in a string. Example: len('happy') evaluates to 5.

for loop traversal (for)

Traversing a string means accessing each character in the string, one at a time. For example, the following for loop:

for ix in 'Example':

...

executes the body of the loop 7 times with different values of ix each time.

slicing ([:])

A slice is a substring of a string. Example: 'bananas and cream'[3:6] evaluates to ana (so does 'bananas and cream'[1:4]).

string comparison (>, <, >=, <=, ==, !=)

The six common comparison operators work with strings, evaluating according to lexigraphical order. Examples: 'apple' < 'banana' evaluates to True. 'Zeta' < 'Appricot' evaluates to False. 'Zebra' <= 'aardvark' evaluates to True because all upper case letters precede lower case letters.

in and not in operator (in, not in)

The in operator tests whether one string is contained inside another string. Examples: 'heck' in "I'll be checking for you." evaluates to True. 'cheese' in "I'll be checking for you." evaluates to False.

9.21. Glossary

collection data type

A data type in which the values are made up of components, or elements, that are themselves values.

default value

The value given to an optional parameter if no argument for it is provided in the function call.

dot notation

Use of the dot operator, ., to access functions inside a module, or to access methods and attributes of an object.

immutable data type

A data type whose values cannot be changed. Modifying functions create a totally new object that does not change the original one.

index

A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from a list.

optional parameter

A parameter written in a function header with an assignment to a default value which it will receive if no corresponding argument is given for it in the function call.

slice

A part of a string (substring) specified by a range of indices. More generally, a subsequence of any sequence type in Python can be created using the slice operator (sequence[start:stop]).

traverse

To iterate through the elements of a collection, performing a similar operation on each.

whitespace

Any of the characters that move the cursor without printing visible characters. The constant string.whitespace contains all the white-space characters.