Module 2 - Chapter 5.1. Strings

Summary

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 ch in "Example":
    ...
```

executes the body of the loop 7 times with different values of `ch` 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 *lexicographical* 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 for membership. In the case of strings, it 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`.

Glossary

compound 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.

docstring A string constant on the first line of a function or module definition (and as we will see later, in class and method definitions as well). Docstrings provide a convenient way to associate documentation with code. Docstrings are also used by programming tools to provide interactive help.

dot notation Use of the **dot operator**, `.`, to access methods and attributes of an object.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

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.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

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.

short-circuit evaluation A style of programming that shortcuts extra work as soon as the outcome is know with certainty. In this chapter our `find` function returned as soon as it found what it was looking for; it didn't traverse all the rest of the items in the string.

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.

Exercises

1. What is the result of each of the following:

```
>>> "Python"[1]
>>> "Strings are sequences of characters."[5]
>>> len("wonderful")
>>> "Mystery"[:4]
>>> "p" in "Pineapple"
>>> "apple" in "Pineapple"
>>> "pear" not in "Pineapple"
>>> "apple" > "pineapple"
>>> "pineapple" < "Peach"
```

2. Modify:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
    print(letter + suffix)
```

so that `Ouack` and `Quack` are spelled correctly.

3. Encapsulate

```
word = "banana"
count = 0
for letter in word:
    if letter == "a":
        count += 1
print(count)
```

in a function named `count_letters`, and generalize it so that it accepts the string and the letter as arguments. Make the function return the number of characters, rather than print the answer. The caller should do the printing.

4. Now rewrite the `count_letters` function so that instead of traversing the string, it repeatedly calls the `find` method, with the optional third parameter to locate new occurrences of the letter being counted.

5. Assign to a variable in your program a triple-quoted string that contains your favourite paragraph of text — perhaps a poem, a speech, instructions to bake a cake, some inspirational verses, etc.

   Write a function which removes all punctuation from the string, breaks the string into a list of words, and counts the number of words in your text that contain the letter "e". Your program should print an analysis of the text like this:

```
Your text contains 243 words, of which 109 (44.8%) contain an "e".
```

6. Print a neat looking multiplication table like this:

```
      1   2   3   4   5   6   7   8   9  10  11  12
   :-----------------------------------------------
 1:   1   2   3   4   5   6   7   8   9  10  11  12
 2:   2   4   6   8  10  12  14  16  18  20  22  24
 3:   3   6   9  12  15  18  21  24  27  30  33  36
 4:   4   8  12  16  20  24  28  32  36  40  44  48
 5:   5  10  15  20  25  30  35  40  45  50  55  60
 6:   6  12  18  24  30  36  42  48  54  60  66  72
 7:   7  14  21  28  35  42  49  56  63  70  77  84
 8:   8  16  24  32  40  48  56  64  72  80  88  96
 9:   9  18  27  36  45  54  63  72  81  90  99 108
10:  10  20  30  40  50  60  70  80  90 100 110 120
11:  11  22  33  44  55  66  77  88  99 110 121 132
12:  12  24  36  48  60  72  84  96 108 120 132 144
```

7. Write a function that reverses its string argument, and satisfies these tests:

```
reverse("happy") == "yppah"
reverse("Python") == "nohtyP"
reverse("") == ""
reverse("a") == "a"
```

8. Write a function that mirrors its argument:

```
mirror("good") == "gooddoog"
mirror("Python") == "PythonnohtyP"
mirror("") == ""
mirror("a") == "aa"
```

9. Write a function that removes all occurrences of a given letter from a string:

```
remove_letter("a", "apple") == "pple"
remove_letter("a", "banana") == "bnn"
remove_letter("z", "banana") == "banana"
remove_letter("i", "Mississippi") == "Mssssppp"
remove_letter("b", "") = ""
remove_letter("b", "c") = "c"
```

10. Write a function that recognizes palindromes. (Hint: use your `reverse` function to make this easy!):

```
is_palindrome("abba")
not is_palindrome("abab")
is_palindrome("tenet")
not is_palindrome("banana")
is_palindrome("straw warts")
is_palindrome("a")
# is_palindrome(""))     # Is an empty string a palindrome?
```

11. Write a function that counts how many times a substring occurs in a string:

```
count("is", "Mississippi") == 2
count("an", "banana") == 2
count("ana", "banana") == 2
count("nana", "banana") == 1
count("nanan", "banana") == 0
count("aaa", "aaaaaa") == 4
```

12. Write a function that removes the first occurrence of a string from another string:

```
remove("an", "banana") == "bana"
remove("cyc", "bicycle") == "bile"
remove("iss", "Mississippi") == "Missippi"
remove("eggs", "bicycle") == "bicycle"
```

13. Write a function that removes all occurrences of a string from another string:

```
remove_all("an", "banana") == "ba"
remove_all("cyc", "bicycle") == "bile"
remove_all("iss", "Mississippi") == "Mippi"
remove_all("eggs", "bicycle") == "bicycle"
```

There are only four *really* important operations on strings, and we'll be able to do just about anything. There are many more nice-to-have methods (we'll call them sugar coating) that can make life easier, but if we can work with the basic four operations smoothly, we'll have a great grounding.

- len(str) finds the length of a string.
- str[i] the subscript operation extracts the i'th character of the string, as a new string.
- str[i:j] the slice operation extracts a substring out of a string.
- str.find(target) returns the index where target occurs within the string, or -1 if it is not found.

So if we need to know if "snake" occurs as a substring within `s`, we could write

```
if s.find("snake") >= 0:  ...
if "snake" in s: ...              # Also works, nice-to-know sugar coating!
```

It would be wrong to split the string into words unless we were asked whether the *word* "snake" occurred in the string.

Suppose we're asked to read some lines of data and find function definitions, e.g.: `def some_function_name(x, y):`, and we are further asked to isolate and work with the name of the function. (Let's say, print it.)

```
s = "..."                    # Get the next line from somewhere
def_pos = s.find("def ")     # Look for "def " in the line
if def_pos == 0:             # If it occurs at the left margin
  op_index = s.find("(")     # Find the index of the open parenthesis
  fnname = s[4:op_index]     # Slice out the function name
  print(fnname)              # ... and work with it.
```

One can extend these ideas:

- What if the function def was indented, and didn't start at column 0? The code would need a bit of adjustment, and we'd probably want to be sure that all the characters in front of the `def_pos` position were spaces. We would not want to do the wrong thing on data like this: `# I def initely like Python!`
- We've assumed on line 3 that we will find an open parenthesis. It may need to be checked that we did!
- We have also assumed that there was exactly one space between the keyword `def` and the start of the function name. It will not work nicely for `def f(x)`
- Suppose any line of text can contain at most one url that starts with "http://" and ends at the next space in the line. Write a fragment of code to extract and print the full url if it is present. (Hint: read the documentation for `find`. It takes some extra arguments, so you can set a starting point from which it will search.)
- Suppose a string contains at most one substring "< ... >". Write a fragment of code to extract and print the portion of the string between the angle brackets.

5.2 Tuples Glossary

data structure An organization of data for the purpose of making it easier to use.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

tuple An immutable data value that contains related elements. Tuples are used to group together related data, such as a person's name, their age, and their gender.

tuple assignment An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs *simultaneously* rather than in sequence, making it useful for swapping values.

Exercises

1. We've said nothing in this chapter about whether you can pass tuples as arguments to a function. Construct a small Python example to test whether this is possible, and write up your findings.
2. Is a pair a generalization of a tuple, or is a tuple a generalization of a pair?
3. Is a pair a kind of tuple, or is a tuple a kind of pair?

5.3 Lists Glossary

aliases Multiple variables that contain references to the same object.

clone To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

delimiter A character or string used to indicate where a string should be split.

element One of the values in a list (or other sequence). The bracket operator selects elements of a list. Also called *item*.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

index An integer value that indicates the position of an item in a list. Indexes start from 0.

list A collection of values, each in a fixed position within the list. Like other types `str`, `int`, `float`, etc. there is also a `list` type-converter function that tries to turn whatever argument you give it into a list.

list traversal The sequential accessing of each element in a list.

modifier A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

nested list A list that is an element of another list.

object A thing to which a variable can refer.

pattern A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the patterns and algorithms that form your toolkit. Patterns often correspond to your "mental chunking".

promise An object that promises to do some work or deliver some values if they're eventually needed, but it lazily puts off doing the work immediately. Calling `range` produces a promise.

pure function A function which has no side effects. Pure functions only make changes to the calling program through their return values.

sequence Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

side effect A change in the state of a program made by calling a function. Side effects can only be produced by modifiers.

step size The interval between successive elements of a linear sequence. The third (and optional argument) to the `range` function is called the step size. If not specified, it defaults to 1.

Exercises

1. What is the Python interpreter's response to the following?

```
>>> list(range(10, 0, -2))
```

The three arguments to the *range* function are *start*, *stop*, and *step*, respectively. In this example, `start` is greater than `stop`. What happens if `start < stop` and `step < 0`? Write a rule for the relationships among `start`, `stop`, and `step`.

2. Consider this fragment of code:

```
1 import turtle
2
3 tess = turtle.Turtle()
4 alex = tess
5 alex.color("hotpink")
```

Does this fragment create one or two turtle instances? Does setting the color of `alex` also change the color of `tess`? Explain in detail.

3. Draw a state snapshot for `a` and `b` before and after the third line of the following Python code is executed:

```
1 a = [1, 2, 3]
2 b = a[:]
3 b[0] = 5
```

4. What will be the output of the following program?

```
1 this = ["I", "am", "not", "a", "crook"]
2 that = ["I", "am", "not", "a", "crook"]
3 print("Test 1: {0}".format(this is that))
4 that = this
5 print("Test 2: {0}".format(this is that))
```

Provide a *detailed* explanation of the results.

5. Lists can be used to represent mathematical *vectors*. In this exercise and several that follow you will write functions to perform standard operations on vectors. Create a script named `vectors.py` and write Python code to pass the tests in each case.

Write a function `add_vectors(vector1,vector2)` that takes two lists of numbers of the same length, and returns a new list containing the sums of the corresponding elements of each:

```
1 add_vectors([1, 1], [1, 1]) == [2, 2]
2 add_vectors([1, 2], [1, 4]) == [2, 6]
3 add_vectors([1, 2, 1], [1, 4, 3]) == [2, 6, 4]
```

6. Write a function `scalar_mult(scalar, vector)` that takes a number, `scalar`, and a list, `vector` and returns the scalar multiple of `vector` by `scalar.`:

```
1 scalar_mult(5, [1, 2]) == [5, 10]
2 scalar_mult(3, [1, 0, -1]) == [3, 0, -3]
3 scalar_mult(7, [3, 0, 5, 11, 2]) == [21, 0, 35, 77, 14]
```

7. Write a function `dot_product(vec1,vec2)` that takes two lists of numbers of the same length, and returns the sum of the products of the corresponding elements of each (the dot_product).

```
1 dot_product([1, 1], [1, 1]) ==  2
2 dot_product([1, 2], [1, 4]) ==  9
3 dot_product([1, 2, 1], [1, 4, 3]) == 12
```

8. *Extra challenge for the mathematically inclined*: Write a function `cross_product(vec1, vec2)` that takes two lists of numbers of length 3 and returns their cross product. You should write your own tests.

9. Describe the relationship between `" ".join(song.split())` and `song` in the fragment of code below. Are they the same for all strings assigned to `song`? When would they be different?

```
1 song = "The rain in Spain..."
```

10. Write a function `replace(s, old, new)` that replaces all occurrences of `old` with `new` in a string `s`:

```
replace("Mississippi", "i", "I") == "MIssIssIppI"
song = "I love spom! Spom is my favorite food. Spom, spom, yum!"
replace(song, "om", "am") ==
    "I love spam! Spam is my favorite food. Spam, spam, yum!"
replace(song, "o", "a") ==
    "I lave spam! Spam is my favarite faad. Spam, spam, yum!"
```

Hint: use the `split` and `join` methods.

5.4 Dictionaries Glossary

call graph A graph consisting of nodes which represent function frames (or invocations), and directed edges (lines with arrows) showing which frames gave rise to other frames.

dictionary A collection of key:value pairs that maps from keys to values. The keys can be any immutable value, and the associated value can be of any type.

immutable data value A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

key A data item that is *mapped to* a value in a dictionary. Keys are used to look up values in a dictionary. Each key must be unique across the dictionary.

key:value pair One of the pairs of items in a dictionary. Values are looked up in a dictionary by key.

mapping type A mapping type is a data type comprised of a collection of keys and associated values. Python's only built-in mapping type is the dictionary. Dictionaries implement the associative array abstract data type.

memo Temporary storage of precomputed values to avoid duplicating the same computation.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

Exercises

1. Write a program that reads a string and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample output of the program when the user enters the data "ThiS is String with Upper and lower case Letters", would look this this:

```
a  2   g  1   n  2   s  5
c  1   h  2   o  1   t  5
d  1   i  4   p  2   u  1
e  5   l  2   r  4   w  2
```

2. Give the Python interpreter's response to each of the following from a continuous interpreter session:

a.
```
>>> dictionary = {"apples": 15, "bananas": 35, "grapes": 12}
>>> dictionary["bananas"]
```

b.
```
>>> dictionary["oranges"] = 20
>>> len(dictionary)
```

c.
```
>>> "grapes" in dictionary
```

d.
```
>>> dictionary["pears"]
```

e.
```
>>> dictionary.get("pears", 0)
```

f.
```
>>> fruits = list(dictionary.keys())
>>> fruits.sort()
>>> print(fruits)
```

g.
```
>>> del dictionary["apples"]
>>> "apples" in dictionary
```

Be sure you understand why you get each result. Then apply what you have learned to fill in the body of the function below:

```
def add_fruit(inventory, fruit, quantity=0):
    return None
new_inventory = {}
add_fruit(new_inventory, "strawberries", 10)
print("strawberries" in new_inventory)
print(new_inventory["strawberries"] == 10)
add_fruit(new_inventory, "strawberries", 25)
print(new_inventory["strawberries"] == 35)
```

3. Write a program called `alice_words.py` that creates a text file named `alice_words.txt` containing an alphabetical listing of all the words, and the number of times each occurs, in the text version of *Alice's Adventures in Wonderland*. (You can obtain a free plain text version of the book, along with many others, from http://www.gutenberg.org.) The first 10 lines of your output file should look something like this:

```
Word                 Count
======================
a                    631
a-piece              1
abide                1
able                 1
about                94
above                3
absence              1
absurd               2
```

How many times does the word `alice` occur in the book?

4. What is the longest word in Alice in Wonderland? How many characters does it have?