00001110

CHAPTER 14

Sequences

Learning Objectives

- *Define the term sequence*
- *Explain the differences between mutable and immutable sequences*
- *Explain the purpose of a list*
- *Process multiple lists in parallel with the zip() function*
- *Explain the purpose of a tuple*
- *Explain the differences between a list and a tuple*
- *List and describe common sequence operations*
- *List and describe the methods supported by list types*
- *Use list comprehensions to create a list*
- *Add items to a list*
- *Remove items from a list*
- *Convert a list into a JSON string*
- *Convert a JSON string into a list*
- *Iterate over strings, lists, and tuples*
- *Access tuple elements via indexers*

Introduction

As soon as you begin to create programs that go beyond simply printing "Hello, World!" to the console, you'll find the need to create, store, and process sequentially arranged data such as a list of numbers, a string of characters, or a row in a spreadsheet, just to name a few. Such data in Python is referred to as a *sequence*. Sequences are both a fundamental concept in Python as well as a fundamental data type. A deep understanding of sequences is vital to understanding *strings*, *lists*, *tuples*, and *ranges*.

Sequences come in two flavors: *mutable* and *immutable*. A *mutable sequence* is one whose elements can be added, modified, and removed. Conversely, the elements contained within an *immutable sequence* cannot be modified. Understanding the differences between mutable and immutable sequences unlocks the key to understanding the differences between lists and strings.

Python lists are similar in concept to arrays in languages like C or C++. The difference between arrays and lists is that arrays are lightweight and close to the hardware whereas Python lists are complex objects that support a wide range of operations.

Along the way you will learn how to use Python's built-in functions `len()`, `min()`, and `max()`, and how to perform sequence *slicing*. The primary purpose of this chapter is to show you how to create and process sequences the Pythonic way.

I will also introduce you to *JavaScript Object Notation* (JSON). JSON has emerged as the de facto standard for the exchange of information between computer systems. I'll show you how to generate JSON from lists and how to convert JSON lists to Python lists. Understanding JSON opens you up to a world of online data resources.

1 An Overview Of Sequences

Python's sequence types include *lists*, *tuples*, and *ranges*. Python also includes specialized types for storing and processing text *strings* and *binary data*. You have already encountered lists and ranges in earlier parts of the book. You have also used strings, which represent a specialized sequence type. All you really need to know to quickly gain proficiency using sequences is what they are, how to create them, the differences between mutable and immutable sequences, and the operations each supports.

1.1 What Is A Sequence?

A sequence is an ordered set of elements that can be accessed individually via a positive or negative integer index. As the saying goes, a picture is worth a thousand words, so let's take a look at figure 14-1.

Referring to figure 14-1 — The string literal `"Hello World!"` is assigned to `string_variable`. A string is an immutable sequence of characters. The `"Hello World!"` string contains 12 character elements, each of which can be accessed individually via a positive or negative integer index number enclosed in square brackets. Thus, `string_variable[0]` accesses the first element of the sequence, `string_variable[4]` accesses the fifth element of the sequence, and `string_variable[11]` accesses the last element of the sequence.

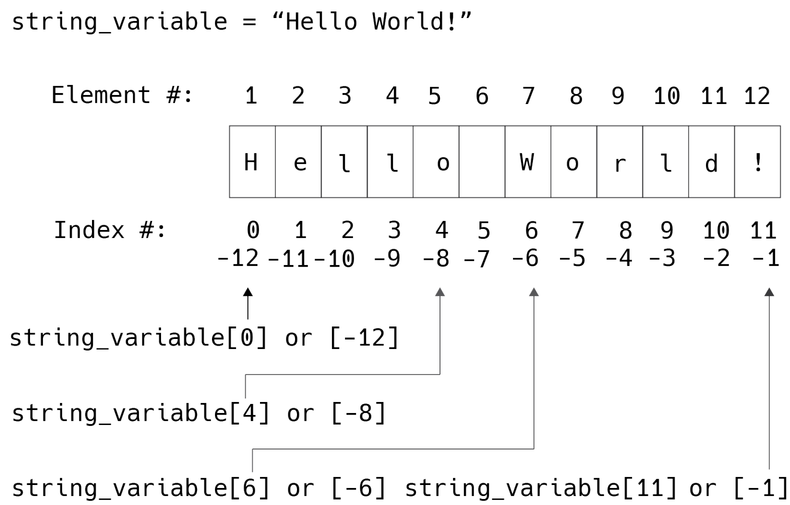

Figure 14-1: Sequence Showing Element vs. Index Numbering

The three most common types of sequences you will use in your programs include strings, lists, and ranges, with tuples coming in a close fourth. I actually went quite a while programming in Python without seeing the need for tuples, but they do come in very handy in many situations.

If you intend to process binary data then you’ll use *byte sequences* with the most common use cases being digital image manipulation and data transfer between client & server applications.

1.2 Immutable Vs. Mutable Sequences

Sequences are either *immutable* or *mutable*. The term immutable means that once a sequence is created, its elements cannot be modified. A mutable sequence is one whose elements can be modified and deleted. Strings are immutable sequences as are tuples and bytes objects. Lists and byte arrays are mutable. I discuss strings, lists, and tuples in greater detail later in this chapter. I’ll introduce you to byte sequences later in the book.

1.3 Common Sequence Operations

Table 14-1 lists common operations supported by both mutable and immutable sequences.

Operation	Result
`x in sequence`	Returns `True` if item in sequence equals x; otherwise `False`
`x not in sequence`	Returns `False` if item in sequence equals x; otherwise `True`
`sequence_1 + sequence_2`	The concatenation of `sequence_1` and `sequence_2`
`sequence * n` *or* `n * sequence`	Add `sequence` to itself `n` times

Table 14-1: Operations Common To Sequences

Operation	Result
`sequence[i]`	Returns `i`th element of `sequence`
`sequence[i:j]`	Returns *slice* of `sequence` from `i` to `j`
`sequence[i:j:k]`	Returns *slice* of `sequence` from `i` to `j` every `k`th element (step)
`len(sequence)`	Returns length of `sequence` (element count)
`min(sequence)`	Returns smallest item in `sequence`
`max(sequence)`	Returns largest item in `sequence`
`sequence.index(x[, i[, j]])`	Returns the index of the first occurrence of `x` in `sequence`, at or after index `i` and before index `j`.
`sequence.count(x)`	Returns total number of occurrences of x in `sequence`
Reference: *https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range*	

Table 14-1: Operations Common To Sequences (Continued)

Referring to table 14-1 — The behavior of a few of these operators varies depending on the type of sequence to which they are applied. I'll bring your attention to special cases as they arise.

Quick Review

Python's sequence types include *lists*, *tuples*, and *ranges*. Python also includes specialized types for storing and processing text *strings* and *binary data*.

A sequence is an ordered set of elements that can be accessed individually via a positive or negative integer index. The three most common types of sequences you will use in your programs include strings, lists, and ranges.

Sequences are either *immutable* or *mutable*. The term immutable means that once a sequence is created, its elements cannot be modified. A mutable sequence is one whose elements can be modified and deleted.

2 Strings

Strings are immutable sequences. Their immutable nature can cause confusion at times, especially if you're a novice programmer. The primary reason for the confusion is because an operation applied to a string results in new string, leaving the original string unaffected. If you keep this in mind when working with strings you should be good to go.

2.1 What Is A String?

A string (`str`) is an immutable text sequence of Unicode characters. Each element of a string is actually a string of length 1 and points to a Unicode code point. Strings have many uses and a deep understanding of how strings behave is crucial to becoming a proficient Python programmer. Before going further, let's look at a few ways to create strings within your program.

14.1 creating_strings.py

```
1    """Demonstrate string creation."""
2
3    def main():
4       # Initialize with empty string
5       empty_string = ''
6
7       # Initialize with string literal in single quotes
8       first_name = 'Rick'
9
10      # Initialize with string literal in double quotes
11      last_name = "Miller"
12
13      # Initialize with a Unicode character code
14      copyright = '\u00A92024'
15
16      # Initialize with emoji
17      emoji = '\N{face with tears of joy} \N{smiling face with halo}' + \
18      '\N{kiss mark} \N{yawning face}'
19
20      print(f'Empty String: {empty_string}')
21      print(f'First Name: {first_name}')
22      print(f'Last Name: {last_name}')
23      print(f'Symbols: {copyright}')
24      print(f'Emoji: {emoji}')
25
26      for character in emoji:
27         print(f'{character}  ', end='')
28
29
30   if __name__ == '__main__':
31      main()
32
```

Referring to example 14.1 — This short program demonstrates various ways to initialize string variables. Note first that string literals can be enclosed in either single or double quotes. On line 5, I'm initializing a variable named `empty_string` with two single quotes and no space between them. On line 8, I define the `first_name` variable and assign to it the string literal `'Rick'`. On line 10, I define the `last_name` variable and assign the string literal `"Miller"`. On line 14, I define a variable named `copyright` and assign to it a Unicode coded character `'\u00a9'`, which is the copyright symbol `'©'` followed by the year 2024. Next, on lines 17 and 18, I define a variable named `emoji` and assign to it several emoji characters referenced by their CLDR short names, a list of which is located here:

https://www.unicode.org/emoji/charts/full-emoji-list.html

Note that which CLDR short names will actually work in your program depends on which version of Python you are running and which version of the CLDR short names is available. Here's a plain text list of the Unicode character set:

https://www.unicode.org/Public/UNIDATA/UnicodeData.txt

Note that the characters used in the string literals on lines 8 and 11 are Unicode characters (Basic Latin) and also part of the ASCII character set:

https://www.unicode.org/charts/PDF/U0000.pdf

If you intend for your code to be portable across different operating systems, I recommend you stick with the Basic Latin (ASCII) characters. You can reference the Unicode Code Charts here:

http://www.unicode.org/charts/

Figure 14-2 shows the results of running this program in iTerm on MacOS.

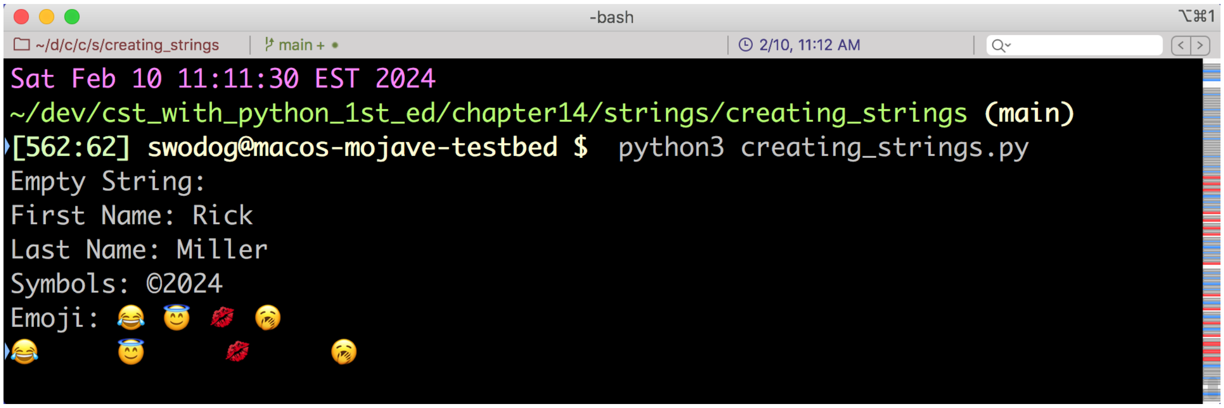

Figure 14-2: Results of Running Example 14.1 in iTerm on MacOS

Referring to figure 14-2 — The emoji's print in iTerm running in MacOS, but you will not have much luck with Git Bash or Linux terminals without custom configuration, which is beyond the scope of this book, and generally unnecessary for most all programming tasks you are likely to attempt, unless you plan to process emojis. Figure 14-3 shows the same code running on Git Bash in Windows.

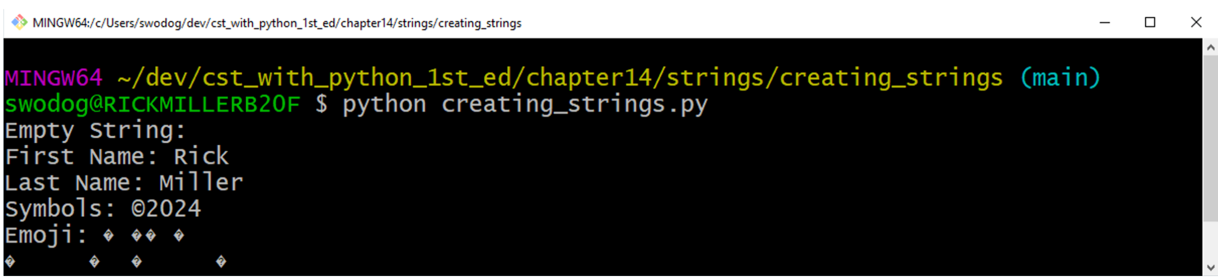

Figure 14-3: Results of Running Example 14.1 in Git Bash on Windows

Note that if you have a burning desire to show emojis in Git Bash, you may, as an exercise, follow the tips provided here: *https://github.com/mintty/mintty/wiki/Tips#emojis*.

Pro Tip: If you want your Python console program to be portable across operating systems, stick with the Basic Latin (ASCII) Unicode Code Chart.

2.2 Common Operations On Strings

In this section, I will demonstrate a few of the most common operations you are likely to perform on strings. To demonstrate every operation possible would bore you to tears, so I'll spare you that terrible fate. Besides, I will introduce you to additional string operations not specifically discussed in this section as you progress through the book and as the use case demands.

The most common operations you will perform on strings include *concatenating* (*joining* or *building*), *accessing individual elements*, *searching*, *slicing*, *determining length*, and *changing case*.

2.3 Concatenation

Concatenation is the million dollar word for joining one string object with another. Note that regardless of the method used, concatenating two immutable sequences together results in a new immutable sequence. (i.e., joining two strings objects together results in a new string object.) Example 14.2 demonstrates the most common ways to concatenate strings.

14.2 string_concatenation.py

```
1    """Demonstrate string concatenation."""
2
3    def main():
4        first_name = 'Rick'
5        last_name = 'Miller'
6        middle_initial = 'W'
7        # Use Concatenation Operator '+'
8        full_name = first_name + ' ' + middle_initial + ' ' + last_name
9        print(full_name)
10
11       age = 35
12       # Use Formatted String a.k.a. 'f' String
13       full_name_and_age = f'{first_name} {middle_initial} {last_name} {age}'
14       print(full_name_and_age)
15
16       # Long Strings with Concatenation Operator '+'
17       passage_one = 'This is an example of a string that must be broken apart ' \
18       'and spread across multiple lines of code.' \
19       '\n\tAuthor:' + full_name + ' ' + str(age)
20       print(passage_one)
21
22       # Long Strings with F Strings
23       passage_two = f'This is another long string being spread over ' \
24       f'multiple lines of code.\n\tAuthor: {full_name_and_age}'
25       print(passage_two)
26
27       # Long Strings with Three Double Quotes
28       passage_three = """This is a moderately long passage. The ancients knew
29   the secret to long life:
30       1. Eat a healthy diet,
31       2. Stay active, and
32       3. Get plenty of rest.
33       – Live – Love – Laugh –"""
34       print(passage_three)
35
36
37   if __name__ == '__main__':
38       main()
39
```

Referring to example 14.2 — There's a lot going on in this short example. I start off by declaring three string variables `first_name`, `last_name`, and `middle_initial`. I then use the concatenation operator `'+'` to join these variables together and assign the resulting string to the `full_name` variable. I then print the `full_name` string to the console.

On line 11, I create an integer variable `age` with value 35. On line 13, I create the `full_name_and_age` string variable and initialize it with the help of a formatted string (a.k.a., an `'f'` string).

On line 17, I declare a variable named `passage_one` and spread the initialization over three lines of code. Important items of note here are the use of the backslash `'\'` character to span lines and the use of the escape characters new line and tab `'\n\t'` to insert a new line and tab respectively. Another important point to note is that when concatenating the integer variable `age` with a string using the concatenation operator, you must first convert it into a string with the built-in `str()` function.

On line 23, I use formatted strings to spread the initialization of the variable `passage_two` across multiple lines.

Lastly, on line 28, I use a long string enclosed in triple-quotes to initialize the variable named `passage_three`. Note that when you use triple quoted strings, the formatting of the string is literally interpreted. This means there is no need to include escaped new lines or tabs. One drawback to triple-quoted strings is that they wrap all the way to the left-most margin (see line 29) unless otherwise formatted.

Figure 14-4 shows the results of running this program.

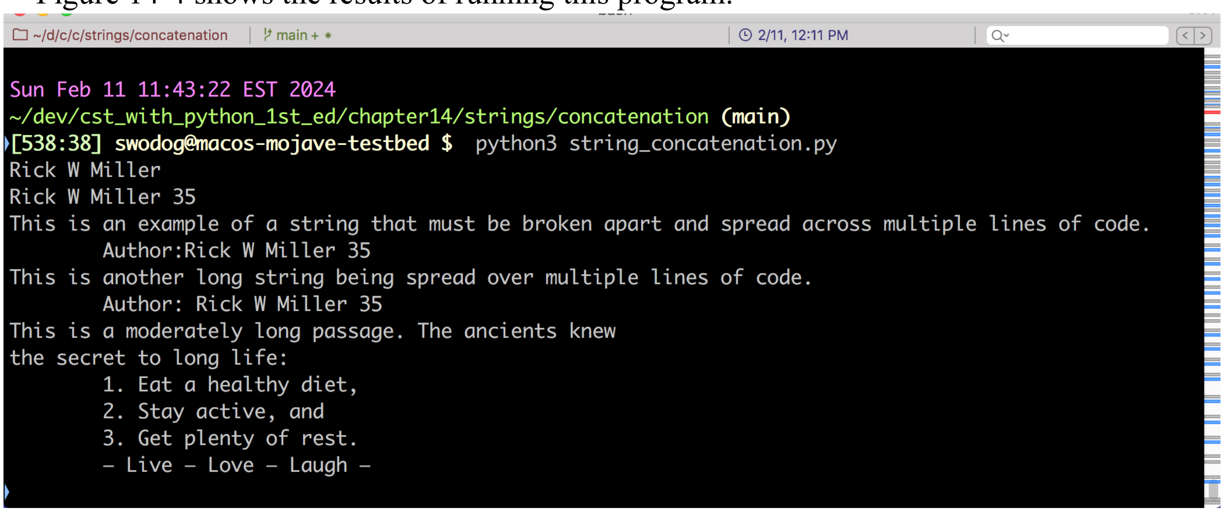

Figure 14-4: Results of Running Example 14.2

2.4 Accessing Individual Characters

You can access individual characters within a string using array notation. Keep in mind that strings are immutable sequences, you can only access individual characters as read-only. Example 14.3 shows how to access individual string characters using positive and negative index values.

14.3 element_access.py

```
"""Demonstrate String Character Access."""

def main():
    s = "These are the times that try men's souls. " \
    "The summer soldier and the sunshine patriot will, " \
    "in this crisis, shrink from the service of their country; " \
    "but he that stands by it now, deserves the love and thanks of " \
    "man and woman. (Thomas Pain, \"The Crisis\", 23 December 1776)"
    # Access characters with array notation
    print(f'{s[0]}')
    print(f'{s[-len(s)]}')

    # Iterate over each character -- Non-Pythonic
    for i in range(len(s)):
        print(f'{s[i]}', end='')

```

```
17      print()
18
19      # Iterate over each character -- Pythonic
20      for c in s:
21         print(f'{c.upper()}', end='')
22
23   if __name__ == '__main__':
24      main()
25
```

Referring to example 14.3 — On line 4, I initialize a string with the famous opening lines of Thomas Paine's *The Crisis*. Lines 10 and 11 illustrate how to access individual string elements using positive and negative index values. Note on line 11 the use of the built-in `len()` function which returns the length, or the number of characters in the string. This value is then negated to yield a negative index value, which yields the string's first character.

Line 14 shows how to iterate over a string's characters in a rather non-Pythonic manner. By non-Pythonic I mean via the use of index values obtained with the `range()` and `len()` functions. Line 20, on the other hand, illustrates the idiomatic Pythonic approach to sequence iteration via the use of an iterator. All Python sequence objects provide an iterator. Note that the `upper()` method merely returns the upper case version of the character; the original character is left unmodified. Figure 14-5 shows the results of running this program.

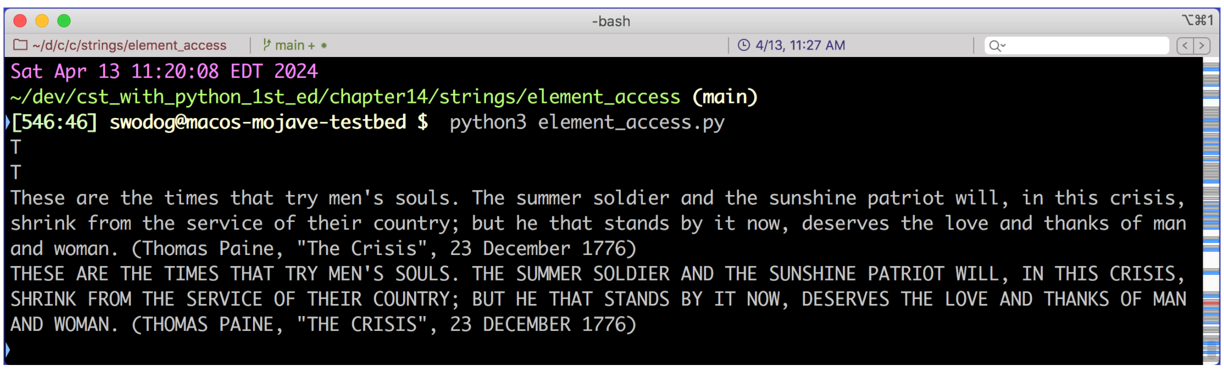

Figure 14-5: Results of Running Example 14.3

A few thoughts before moving on. I've never needed to access individual string characters, and by extension list elements in general, via negative index values, however, you never know when it may come in handy.

As shown above, use the `upper()` method to obtain upper-case characters, and use the `lower()` method to obtain lower-case characters. Always remember — strings are immutable — so calls to `upper()` and `lower()` return a new string in the desired format and leave the original string unchanged.

2.5 Searching

Many times you need to search for substrings and patterns within a string, or count the occurrences of a substring within a string. In this section I will demonstrate the use of the `in` operator and the `str.find()` and `str.count()` methods. Example 14.4 lists the example code.

14.4 string_search.py

```
"""Demonstrate simple string searching."""

def main():
```

```
4      s = 'Now friendship may be thus defined: a complete accord on all subjects\n' \
5   'human and divine, joined with mutual goodwill and affection. And with\n' \
6   'the exception of wisdom, I am inclined to think nothing better than this\n' \
7   'has been given to man by the immortal gods. There are people who give\n' \
8   'the palm to riches or to good health, or to power and office, many even\n' \
9   'to sensual pleasures. This last is the ideal of brute beasts; and of the\n' \
10  'others we may say that they are frail and uncertain, and depend less on\n' \
11  'our own prudence than on the caprice of fortune. Then there are those\n' \
12  'who find the "chief good" in virtue. Well, that is a noble doctrine. But\n' \
13  'the very virtue they talk of is the parent and preserver of friendship,\n' \
14  'and without it friendship cannot possibly exist. "Cicero"\n'
15
16     print(s)
17
18     # Is substring in string
19     if 'friendship' in s:
20        print('Yes, the word \"friendship\" is in the passage.')
21     else:
22        print('The word \"friendship\" not found.')
23
24     # At what index position does substring begin
25     # Start searching from 0 index (start of string)
26     print(f'The word \"friendship\" begins at index {s.find("friendship")}')
27
28     # At what index position does substring begin
29     # Start searching at index position n
30     print(f'The next occurence of \"friendship\" begins at index \
31        {s.find("friendship",5)}')
32
33     # Count the occurrences of the word friendship
34     print(f'The word \"friendship\" appears {s.count("friendship")} times.')
35
36
37  if __name__ == '__main__':
38     main()
39
```

Referring to example 14.4 — On line 4, I declare a string variable named `'s'` initialized with a quote about friendship penned by the ancient Roman senator and writer *Cicero*. On line 19, I use the `in` operator to see if the word "friendship" appears within the passage. On line 26, I use the `str.find()` method to search for the substring "friendship" starting at the beginning of the passage. (i.e., `s.find("friendship")`) The `find()` method returns the starting index of the first character of the substring within the containing string. In this case, the call to `s.find("friendship")` returns the value 4. On line 30, I use another version of the `str.find()` method to start searching at a particular index for the substring "friendship", in this case 5. Finally, on line 34, I call the `str.count()` method to count the occurrences of the substring "friendship". Figure 14-6 shows the results of running this program.

Pro Tip: Use the **in** operator to see if a string contains a substring. You can also check for the absence of a substring by negating the **in** operator. (i.e., **not in**)

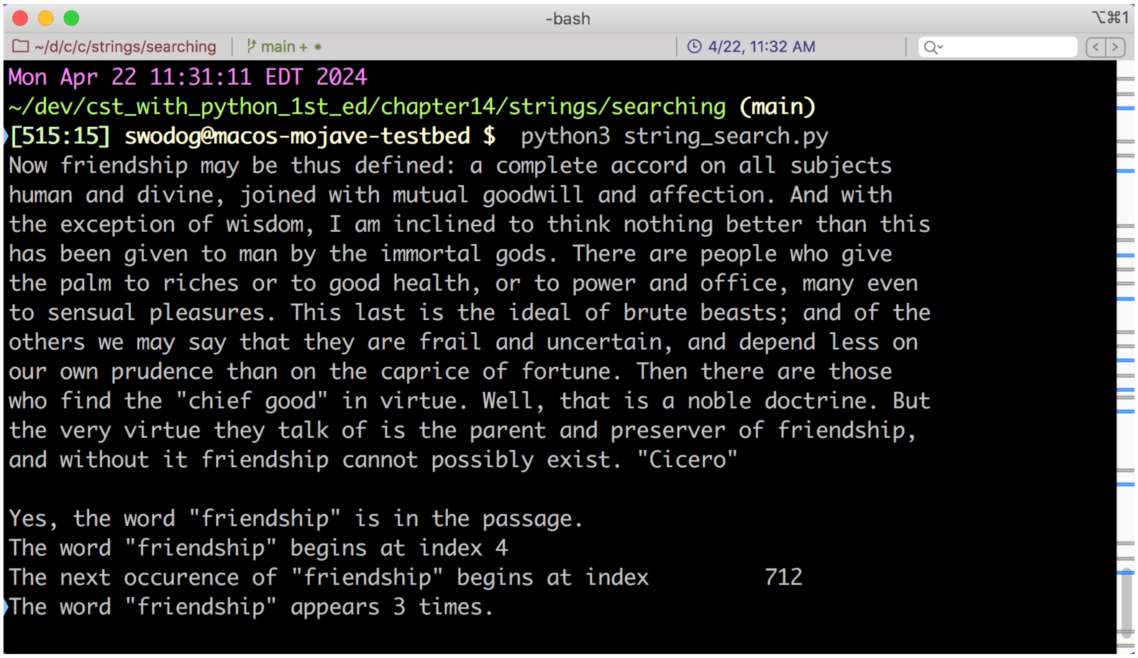

Figure 14-6: Results of Running Example 14.4

2.6 Slicing

You will encounter many times the need to extract sections of a string either from the beginning up to some index position or from a designated index position to another. You can do these types of operations using *sequence slicing*. Example 14.5 shows how to perform various types of slicing operations on a string.

14.5 string_slicing.py

```
"""Demonstrate slicing operations on strings."""

def main():
    s = 'As in books on geography, Sossius Senecio, the writers crowd the\n' \
'countries of which they know nothing into the furthest margins of their\n' \
'maps, and write upon them legends such as, "In this direction lie\n' \
'waterless deserts full of wild beasts;" or, "Unexplored morasses;" or,\n' \
'"Here it is as cold as Scythia;" or, "A frozen sea;" so I, in my\n' \
'writings on Parallel Lives, go through that period of time where history\n' \
'rests on the firm basis of facts, and may truly say, "All beyond this is\n' \
'portentous and fabulous, inhabited by poets and mythologers, and there\n' \
'is nothing true or certain."\n' \
'\tFrom "Life of Theseus", Plutarch\'s Lives, Vol. I\n'

    # Print entire string
    print(f'Print entire string:\n {s}\n')

    # Print first 10 characters (index values 0 - 9)
    print(f'Print first 10 characters:\n {s[:10]}\n')

    # Print section of string from index 10 to 19
    print(f'Print section of string from index 10 to 19:\n {s[10:20]}\n')

    newline = '\n'
```

```
25      # Print first line
26      print(f'Print first line:\n {s[:s.find(newline)]}')
27
28  if __name__ == '__main__':
29      main()
30
```

Referring to example 14.5 — On line 4, I've initialized the variable `'s'` with the opening paragraph from the "*Life of Theseus*", Plutarch's Lives, Volume I. The first thing I do on line 16 is to print the entire string to the console for reference. Next, on line 19, I print the first 10 characters of the string. The expression `s[:10]` yields the characters from index position `0` through `9`. On line 22, I print the characters from index position 10 through 19 using the expression `s[10:20]`. Finally, on line 26, I print all the characters in the first line by slicing from the beginning of the string up to the first occurrence of the newline character `'\n'`. Note that on line 24 I have declared a variable named `newline` and initialized it with `'\n'`. This is required because backslash characters are not allowed in formatted string expressions. The slicing expression `s[:s.find(newline)]` returns all characters from the beginning of the string up to the first occurrence of `'\n'`. Figure 14-7 shows the results of running this program.

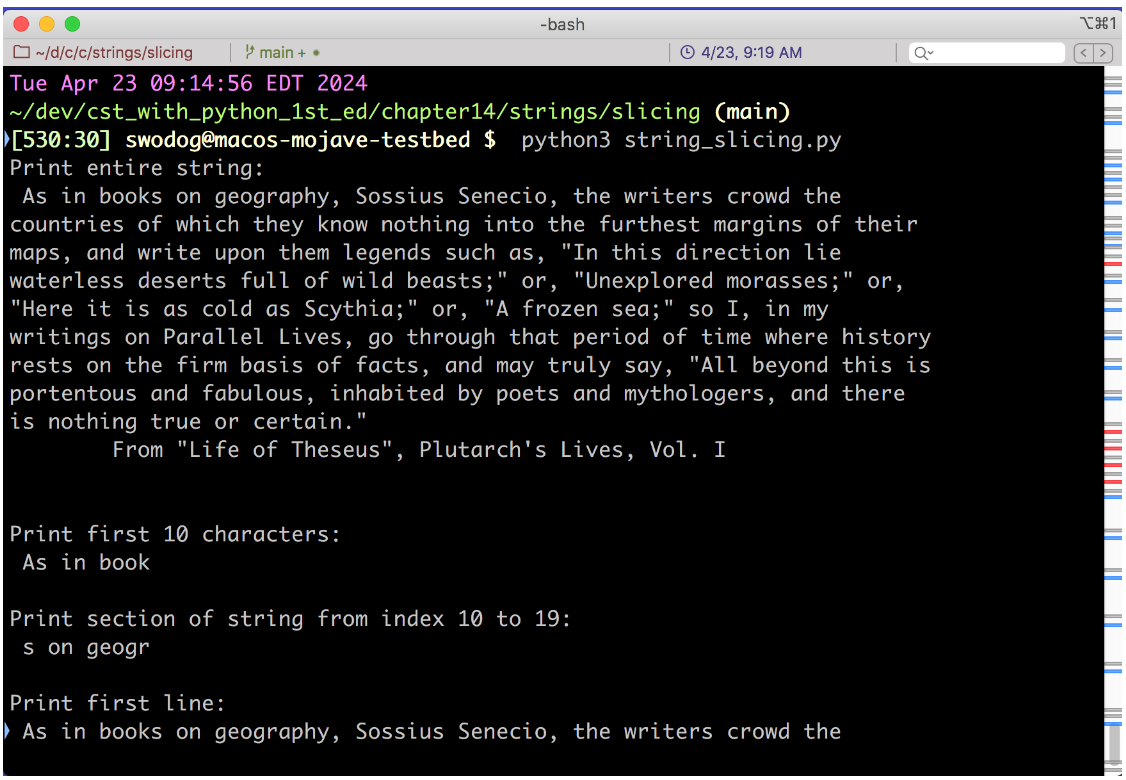

Figure 14-7: Results of Running Example 14.5

Quick Review

Strings are immutable sequences and support many of the same operations as their mutable list counterparts. Common operations performed on strings include *joining/building*, *accessing individual characters*, *searching*, and *slicing*. Always keep in mind that methods like `string.upper()` and `string.lower()` return new strings and leave the original string untouched.

3 Lists

A list is a *mutable* sequence of elements. You can *add* elements to a list, *modify* list elements, and *delete* elements from a list. Unlike a string, whose elements are all single-length Unicode code points, a list's elements can be a mix of different types, however, a list with mixed-type elements is an unusual use case. More often than not, you will create lists to hold elements of the same type. (i.e., lists of strings, lists of integers, lists of some class-type objects, etc.) Enough jabbering! Let's get into it!

3.1 Creating and Initializing Lists

Example 14.6 shows various ways to create and initialize lists.

14.6 create_lists.py

```
1    """Demonstrate various ways to create and initialize lists."""
2
3    def main():
4        # Create empty list
5        breakfast_club = []
6        # Then add elements
7        breakfast_club.append('Rick')
8        breakfast_club.append('Tri')
9        breakfast_club.append('Alex')
10       breakfast_club.append('Raffi')
11       print(f'Breakfast Club Has {len(breakfast_club)} \
12       Members: {breakfast_club}')
13
14       # Create and initialize with a list literal
15       it_566 = ['Jawaher', 'Bader', 'Matthew', 'Anthony',
16           'Davis', 'Lewis', 'Joseph']
17       print(f'IT-566 Class has {len(it_566)} Students: {it_566}')
18
19       print('*' * 20)
20
21       # Create another empty list
22       list_of_lists = []
23       # Then add the existing lists
24       list_of_lists.append(breakfast_club)
25       list_of_lists.append(it_566)
26       print(f'List of Lists: {list_of_lists}')
27
28   if __name__ == '__main__':
29       main()
30
```

Referring to example 14.6 — On line 5, I create an empty list named `breakfast_club` with the help of a set of opening and closing brackets "`[]`". Armed with an empty list, you can add items to the list with the `append()` method as shown on lines 7 through 10. On line 11, I print the number of items in the list with the help of the built-in `len()` function followed by the list's items. Next, on line 15, I create a new list named `it-566` and initialize it with a list literal. A list literal is a set of comma-separated elements contained within a set of square brackets. On line 17, I print the number of items in the list followed by the list's items. Finally, on line 22, I create another empty list named `list_of_lists`, append the previous two lists, and print the list. Figure 14-8 shows the results of running this program.

```
Sun May  5 08:42:31 EDT 2024
~/dev/cst_with_python_1st_ed/chapter14/lists/creating_lists (main)
[530:30] swodog@macos-mojave-testbed $  python3 create_lists.py
Breakfast Club Has 4    Members: ['Rick', 'Tri', 'Alex', 'Raffi']
IT-566 Class has 7 Students: ['Jawaher', 'Bader', 'Matthew', 'Anthony', 'Davis', 'Lewis', 'Joseph']
********************
List of Lists: [['Rick', 'Tri', 'Alex', 'Raffi'], ['Jawaher', 'Bader', 'Matthew', 'Anthony', 'Davis', 'Lewis', 'Joseph']]
```

Figure 14-8: Results of Running Example 14.6

Referring to figure 14-8 — Note how printing a list renders the list items on the console. The items appear as a set of comma-separated elements within square brackets. The square brackets indicate that what is printed is a list. Note also how the `list_of_lists` prints to the console. It renders as two lists contained within a list. (i.e., `[[] []]`)

3.2 List Comprehensions

Another way to create and initialize a list is via a *list comprehension*. It can take some effort to wrap your head around list comprehensions, so let's go straight to an example. Example 14.7 shows how one might create a list of Latin alphabet consonants with the help of a list comprehension.

14.7 list_comprehensions.py

```
1    """Demonstrates list comprehensions."""
2
3    def main():
4        # Create list of vowels
5        vowels = ['A', 'E', 'I', 'O', 'U']
6        # Create list of consonants
7        consonants = [chr(c) for c in range(65,91) if chr(c) not in vowels]
8        print(f'Consonants: {consonants}')
9
10   if __name__ == '__main__':
11       main()
12
```

Referring to example 14.7 — I start on line 5 by creating a list of capitalized vowels. On line 7, I create a list of consonants with the help of a list comprehension. A list comprehension is contained within a set of square brackets. It begins with an *expression* followed by one or more `for` and `if` clauses. The `chr()` built-in function converts an integer into a character and serves as the expression. The `range()` function generates integers from `65` to `90`. These correspond to the ASCII characters `'A'` - `'Z'`. Thus, the list comprehension on line 7 can be read like so:

"*Create a list of characters, `chr(c)`, that contains values of `c` between 65 through 90 whose character values are not in the list of vowels.*"

Figure 14-9 shows the results of running this program.

```
Sun May  5 10:56:27 EDT 2024
~/dev/cst_with_python_1st_ed/chapter14/lists/list_comprehensions (main)
[549:49] swodog@macos-mojave-testbed $  python3 list_comprehensions.py
Consonants: ['B', 'C', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'X', 'Y', 'Z']
```

Figure 14-9: Results of Running Example 14.7

You could create the same list of consonants without using a list comprehension as shown in example 14.8.

14.8 consonants.py

```
1    """Create list of consonants using ordinary means."""
2
3    def main():
4       # Create list of vowels
5       vowels = ['A', 'E', 'I', 'O', 'U']
6       # Create empty list to hold consonants
7       consonants = []
8       # Step through ASCII values 65 through 90
9       #   If c not a vowel add to list
10      for c in range(65, 91):
11         if chr(c) not in vowels:
12            consonants.append(chr(c))
13
14      print(f'Consonants: {consonants}')
15
16
17   if __name__ == '__main__':
18      main()
19
```

Referring to example 14.8 — As in the previous example, I start by creating a list of vowels. Next, I create an empty list named `consonants`. The `for` statement beginning on line 10 steps through the range of values `65` through 90. If the value `chr(c)` is **not in** the list of values, it is added to the consonants list. Note the order of the `for` and `if` statements are the same here as they are in the list comprehension of the previous example.

3.2.1 Thoughts On List Comprehensions

Personally, I don't reach for list comprehensions on my first attempt at writing code. I will write code and then review the results to see if it can be distilled and clarified with a list comprehension. Complex list comprehensions can be tough for mere mortals to decipher and tend to obfuscate rather than clarify. This leads to the following Pro Tip:

Pro Tip: Avoid overly complex list comprehensions, especially if they obfuscate the meaning of your code rather than clarify.

3.3 Processing Lists

Up till now in this section, I've only printed the entire list to the console. Sometimes this may be what you want to do, but generally you will need to iterate over a list and perform some type of processing on each element. The following example prints a list of student names and other information in a tabular format using the PrettyTable package.

14.9 process_names.py

```
"""Demonstrate list processing."""

from prettytable import PrettyTable

def main():
   students = list()
   students.append("Anthony Alston, Masters, IT")
```

```
8        students.append("Phillip Behrns, Masters, IT")
9        students.append("Lkhagvasuren Dechinlkhundev, Masters, Cybersecurity")
10       students.append("Samantha King, PhD, Cybersecurity")
11       students.append("Claire Madison, Masters, Cybersecurity")
12
13       table = PrettyTable()
14       table.field_names = ["Student Name", "Degree", "Major"]
15       table.align = 'l'
16
17       for student in students:
18           info = student.split(',')
19           table.add_row([info[0], info[1], info[2]])
20
21       print(table)
22
23   if __name__ == '__main__':
24       main()
25
```

Referring to example 14.9 — There's a lot going on in this short example. First, on line 3, I import PrettyTable. For more information about how to install and use PrettyTable see the package page on PyPI: *https://pypi.org/project/prettytable/* Next, on line 6, I create an empty list using the `list()` constructor. I then add several students and their information to the list. Note that I'm adding comma separated strings. On line 13, I create a `table` and populate the `table.field_names` property with a list of strings representing each field name. I then set the table alignment to left. The bulk of the processing occurs in the `for` statement starting on line 17. I step through each `student` string in the `students` list and split the string into three separate strings. The result of the `split()` method is a list of three strings using a comma as a field delimiter. I then add a row to the table and populate it with data from the `info` list. (i.e., `info[0]` yields the student's name, `info[1]` yields the degree, and `info[2]` yields the major) Figure 14-10 shows the results of running this program.

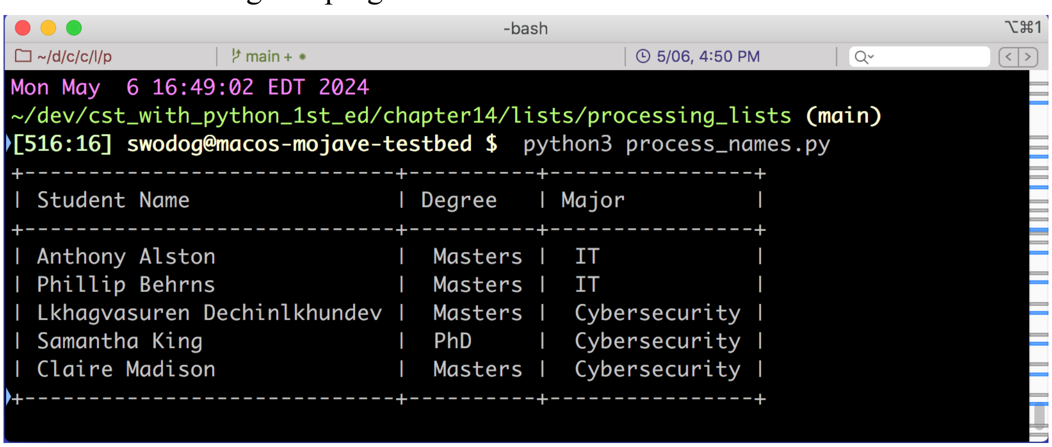

Figure 14-10: Results of Running Example 14.9

3.3.1 The enumerate Function

Earlier in the section on strings in example 14.3, I showed you how to iterate over a sequence in a Pythonic and non-Pythonic way. The non-Pythonic way is considered non-Pythonic because it employs the `range()` and `len()` functions to generate an index with which to access each sequence element. Using these functions to iterate over a sequence is considered non-Pythonic by Python snobs mainly because, they claim, it shows a lack of understanding of Python sequence iteration idioms. Usually, it's programmers who come to Python from other languages like C, C++, and Java, who are most likely to iterate over a sequence in Python using idiomatic expressions supported by those languages. By idiomatic expressions, I'm talking about `for` loops as shown in the following brief example written in the C programming language.

14.10 main.c

```
1    /* A Short C Program to Process an Array of Integers. */
2
3    #include <stdio.h>
4
5    int main(){
6
7        int total = 0;
8        int numbers[] = {1,2,3,4,5,6,7,8,9,10};
9
10       for(int i = 0; i < 10; i++){
11          total += numbers[i];
12       }
13
14       printf("Total = %d", total);
15
16       return 0;
17   }
18
```

Referring to example 14.10 — Even if you've never seen a C program, you should be able to nuke out what's happening in the code. All the action takes place within a function named `main()`. On line 7, I declare an integer variable named `total` and assign to it the value `0`. On the following line, I declare and initialize an array of ten integers. The `for` statement that begins on line 10 uses an indexer named `i` to step through each element of the array and add the value at the i^{th} position to the `total`.

Look closely at the way the `for` statement is written in the example above. This is what I mean when I refer to an idiomatic looping statement in languages like C, C++, etc. If you are an experienced programmer in a programming language with similar `for` statements, writing them this way is a tough habit to break.

Conversely, iterating over a sequence the Pythonic way in Python is akin to using a `foreach` statement in C#. If you're not familiar with that language, don't worry, it's very similar to how the Python `for` statement works. Sometimes, however, you want to iterate over a sequence and extract both the *value* of each element and the *indexer* associated with each value. This is where the built-in `enumerate()` function comes in handy as shown in the following example.

14.11 enumerating_lists.py

```
"""Demonstrate the enumerate() function."""

from prettytable import PrettyTable

def main():
    names = ['Sarah', 'Sapna', 'Laura', 'Anita', 'Diana', 'Katerina']
    table = PrettyTable()
    table.align = 'l'
    table.field_names = ['Index', 'Name']

    for index, value in enumerate(names):
        table.add_row([index, value])
```

```
13
14        print(table)
15
16    if __name__ == '__main__':
17        main()
18
```

Referring to example 14.11 — This example uses the PrettyTable package again to format the output in a more human-readable manner. On line 6, I declare and initialize a list of `names`. The `for` statement on line 11 uses the `enumerate()` function to extract both the `index` and `value` of each list element. I then add each `index` and `value` to a new row in the table. Finally, on line 14, I print the table to the console. Figure 14-11 shows the results of running this program.

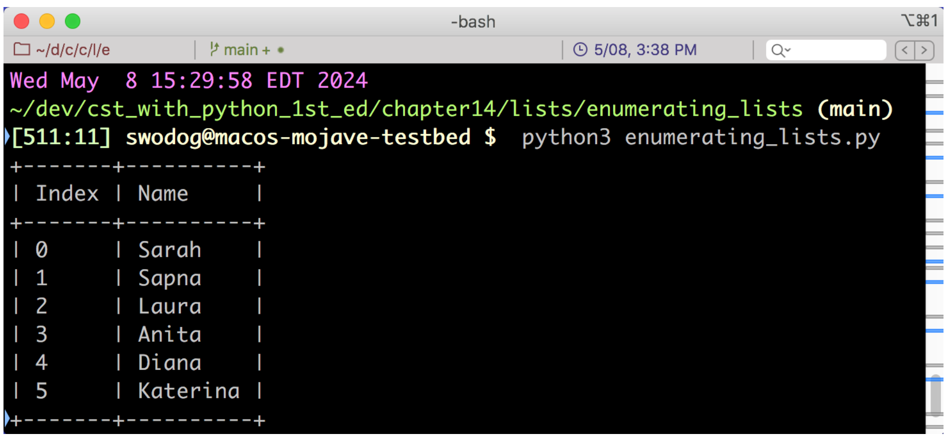

Figure 14-11: Results of Running Example 14.11

Note that the hardest part about using the `enumerate()` function is remembering the order it returns the `index` and the `value`. Just remember, "*I before V she whispered to me!*" (;-o)

3.4 More List Operations

Table 14-2 lists operations you can perform on a list (mutable sequence), including the `list.append()` and `list.count()` methods you're already seen in action.

Operation	Result
`list.append(item)`	Add an item to the end of the list. Increases len(list) by one.
`list.extend(iterable)`	Add all the items in iterable to the end of the list. Increases len(list) by number of items in iterable.
`list.insert(index, item)`	Insert item at given index position.
`list.remove(item)`	Remove the first element from the list with value equal to item. Throws a ValueError exception if the item is not in the list.

Table 14-2: List (Mutable Sequence) Operations

Operation	Result
`list.pop()` or `list.pop(index)`	list.pop() removes and returns the last element from the list. list.pop(index) removes and returns the element from given index. Throws IndexError if the list is empty or index is invalid.
`list.clear()`	Removes all list elements.
`list.index(x[, i[, j]])`	Returns the index of the first occurrence of `x` in `sequence`, at or after index `i` and before index `j`.
`list.count(item)`	Returns the number of times item appears in the list.
`list.sort(*, key=none, reverse=False)`	Sort list elements in place. (See example below.)
`list.reverse()`	Reverse list elements in place.
`list.copy()`	Returns shallow copy of list.

Table 14-2: List (Mutable Sequence) Operations (Continued)

I'll leave most of these operations for you to explore on your own, but I will demonstrate the `sort()` method and show you how the `key` parameter allows you to customize the sorting operation as shown in example 14.12.

14.12 sort_list.py

```
1    """Demonstrate list.sort() method."""
2
3    def main():
4        letters = ['z', 'W', 'Z', 'Q', 'a', 'A', 'm']
5        print(f'Original Order: {letters}')
6        letters.sort()
7        print(f'Default sort(): {letters}')
8        letters.sort(key=str.upper)
9        print(f'Specify sort key param: {letters}')
10
11   if __name__ == '__main__':
12       main()
13
```

Referring to example 14.12 — In this short example, I create a list of mixed-case letters in higgledy-piggledy order. My intention is to sort the list in ascending order, however, if you look at an ASCII chart, you'll notice that the upper case letters come before the lower case letters. A call to `letters.sort()` without the optional key parameter (i.e., a default sort) confirms this. On line 8, I assign the name of the `str.upper` method to the `key` parameter, which calls the `str.upper()` method on each character in the list. Figure 14-12 shows the results of running this program.

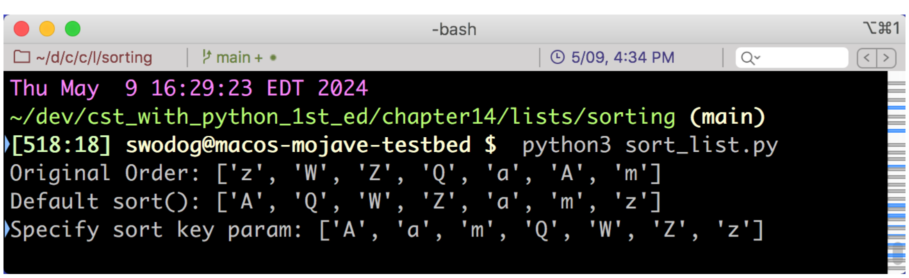

Figure 14-12: Results of Running Example 14.12

3.5 Converting List To JSON

JavaScript Object Notation (JSON) is considered the de facto standard for IT systems data interchange, and Python makes it easy to convert lists into JSON strings and back again. Example 14.13 offers a short example that shows you how to perform JSON encoding and decoding with the help of the `json` package and the `json.dumps()` and `json.loads()` methods.

14.13 json_encoding_decoding.py

```
1    """Demonstrate how to perform JSON encoding and decoding on a list."""
2
3    import json
4
5    def main():
6        students = ['Davis', 'Badar', 'Matthew', 'Lewis', 'Joseph']
7        print(f'students list: {students}')
8        # Use json.dumps() to convert list to JSON string
9        students_json = json.dumps(students)
10       print(f'students_json: {students_json}')
11       # Use json.loads() to create Python object from JSON string
12       students_two = json.loads(students_json)
13       print(f'students_two list: {students_two}')
14
15
16   if __name__ == '__main__':
17       main()
18
```

Referring to example 14.13 — Starting from the top, on line 3, I import the `json` package, which is part of the Python standard library. Next, on line 6, I create a list of student names and print the list to the console. On line 9, I convert the list into a JSON string and print it to the console. Note the difference between how a Python list prints to the console vs. a JSON list. The JSON list items are surrounded by double-quotes whereas Python list items are surrounded by single-quotes. Finally, on line 10, I convert the JSON string back into a Python object, and print the list to the console again as a check. Figure 14-13 shows the results of running this program.

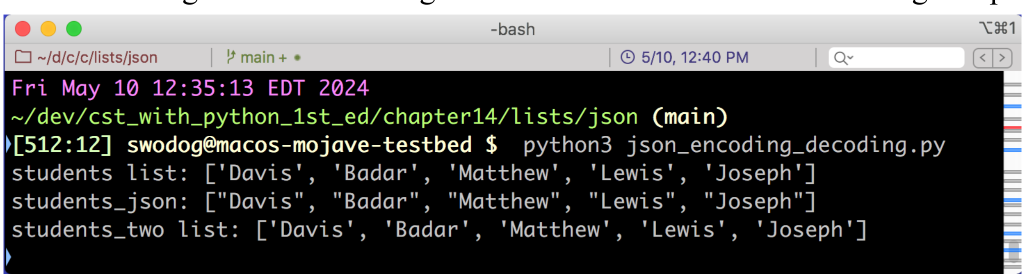

Figure 14-13: Results of Running Example 14.13

You'll learn more about JSON and how to use it to share data between applications as you progress through the book.

Quick Review

A list is a *mutable* sequence of elements. You can *add* elements to a list, *modify* list elements, and *delete* elements from a list. Unlike strings, lists can have elements of any data type including, but by no means limited to, integers, floats, strings, other lists, dictionaries, and user-defined data types.

You can create lists in many different ways including with square brackets `"[]"`, the `list()` constructor, and list comprehensions. Use list comprehensions sparingly as they can be challenging to decipher.

Lists can be processed using `for` statements. If you need to extract both the `index` and `value` from a list during processing use the `enumerate()` function.

Use the `json.dumps()` method to convert a Python list into a JSON list string. Use the `json.loads()` method to convert a JSON list string into a Python list.

4 Ranges

A range behaves like an immutable sequence of numbers. You've already seen ranges in action but I would like to elaborate on exactly what a range is and how they can be used. When you create a range using the `range()` function, you are actually calling the range type's constructor. Let's take a look at a short example that highlights some of the more common uses of ranges.

14.14 range_common_uses.py

```python
"""Demonstrate common uses of ranges."""

def main():
    # To loop n number of times
    for i in range(10):
        print(f'{i}, ', end='')

    print()

    # Specify start, stop, and step
    for i in range(2, 100, 10):
        print(f'{i}, ', end='')

    print()

    # Assign range to variable
    range_one = range(10)
    range_two = range(10, 21)
    print(f'{range_one}')
    print(f'{range_two}')

    # Create list from range
    num_list = list(range_one)
    print(f'{num_list}')

    # Extend list with range
    num_list.extend(range_two)
    print(f'{num_list}')

    # Check for membership
    print(f'5 in range_one: {5 in range_one}')
    print(f'5 in range_two: {5 in range_two}')


if __name__ == '__main__':
    main()

```

Referring to example 14.14 — On line 5, I use a range to generate ten integer values from 0 - 9 and print the value of `i` in the body of the `for` loop. This is perhaps the most frequent way you'll see a range employed. On line 11, I specify the `start`, `stop`, and `step` parameters to generate every 10th number between 2 and 99. On lines 17 and 18, I create two ranges and assign them to the corresponding variables `range_one` and `range_two`. Next, on line 23, I create a list named `num_list` from `range_one`. On line 27, I extend `num_list` with `range_two`. Finally, on lines 31 and 32, I search for values in each range using the `in` operator. Figure 14-14 shows the results of running this program.

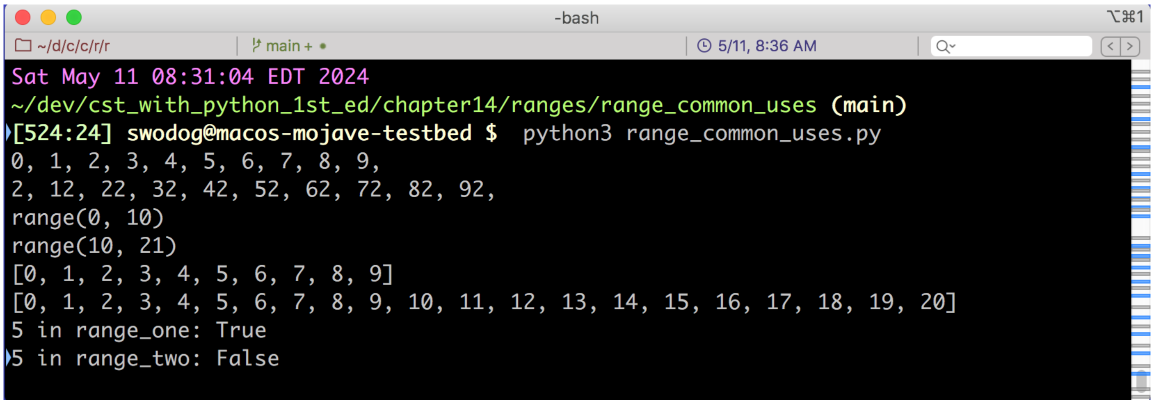

Figure 14-14: Results of Running Example 14.14

Referring to figure 14-14 — Notice when I print the range variables, only their type and settings print to the console, not the values they generate. Although a range can be treated like a sequence, it does not contain all of its possible values, like a list does. A range object will generate its specified values one-at-a-time, as, for example, when a range is used to supply values to a `for` loop. Thus, a range is a generator with super powers. You'll learn more about generators later in the book.

Quick Review

A range behaves like an immutable sequence of numbers. It's a cross between an immutable sequence and a generator. A generator is function or object that returns values one-at-a-time. When creating a range with the `range()` function, you are actually calling the constructor on a range object. Ranges are used for a variety of purposes from generating index values in `for` loops to populating lists with numeric values.

5 Tuples

A tuple is a immutable list. Tuples are created using parentheses `"()"` or by simply separating tuple items with commas as shown in example 14.15.

14.15 tuples.py

```
"""Demonstrate how to create and use tuples."""

def main():
    # Create tuple with parentheses
    person_1 = ('Steven', 'Hester')
    # Create tuple with commas
```



```
7        person_2 = 'David', 'Miller'
8        # Access tuple elements via indexers
9        print(f'{person_1[0]} {person_1[1]}')
10       print(f'{person_2[0]} {person_2[1]}')
11
12       # Process tuple with for loop
13       for s in person_1:
14           print(f'{s} ', end='')
15
16
17   if __name__ == '__main__':
18       main()
19
```

Referring to example 14.15 — On line 5, I'm creating a tuple with two string elements using parentheses. On line 7, I creating a tuple with two string elements using commas. Both methods of creating tuples work fine, but I prefer to use parentheses. Once you have a tuple, you can use it like a list, but since it's immutable, you can't add, modify, or delete its elements. Figure 14-15 shows the results of running this program.

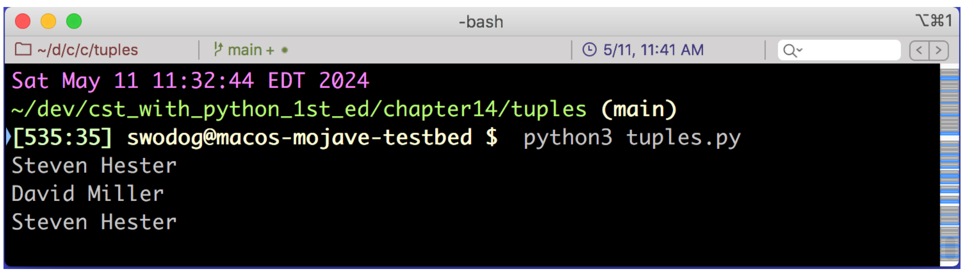

Figure 14-15: Results of Running Example 14.15

That really covers the basics of tuples. You'll learn more about how and when to use tuples as you progress through the book. Just remember they are immutable. What's the big deal about immutability, anyway? Good question. I will answer it in the next chapter.

Quick Review

A tuple is an immutable list. Tuples can contain any type of data. You can create tuples with parentheses or with comma separated items. Once you have a tuple, you can use it like a list, but since it's immutable, you cannot add, modify, or delete tuple items.

Summary

Sequences come in two flavors: *mutable* and *immutable*. A *mutable sequence* is one whose elements can be added, modified, and removed. Conversely, the elements contained within an *immutable sequence* cannot be modified. Understanding the differences between mutable and immutable sequences unlocks the key to understanding the differences between lists and strings.

Python's sequence types include *lists*, *tuples*, and *ranges*. Python also includes specialized types for storing and processing text *strings* and *binary data*.

A sequence is an ordered set of elements that can be accessed individually via a positive or negative integer index. The three most common types of sequences you will use in your programs include strings, lists, and ranges.

Strings are immutable sequences and support many of the same operations as their mutable list counterparts. Common operations performed on strings include *joining/building*, *accessing individual characters*, *searching*, and *slicing*. Always keep in mind that methods like `string.upper()` and `string.lower()` return new strings and leave the original string untouched.

A list is a *mutable* sequence of elements. You can *add* elements to a list, *modify* list elements, and *delete* elements from a list. Unlike strings, lists can have elements of any data type including, but by no means limited to, integers, floats, strings, other lists, dictionaries, and user-defined data types.

You can create lists in many different ways including with square brackets `"[]"`, the `list()` constructor, and list comprehensions. Use list comprehensions sparingly as they can be challenging to decipher.

Lists can be processed using `for` statements. If you need to extract both the `index` and `value` from a list during processing use the `enumerate()` function.

Use the `json.dumps()` method to convert a Python list into a JSON list string. Use the `json.loads()` method to convert a JSON list string into a Python list.

A range behaves like an immutable sequence of numbers. It's a cross between an immutable sequence and a generator. A generator is function or object that returns values one-at-a-time. When creating a range with the `range()` function, you are actually calling the constructor on a range object. Ranges are used for a variety of purposes from generating index values in `for` loops to populating lists with numeric values.

A tuple is an immutable list. Tuples can contain any type of data. You can create tuples with parentheses or with comma separated items. Once you have a tuple, you can use it like a list, but since it's immutable, you cannot add, modify, or delete tuple items.

Skill-Building Exercises

1. **String Manipulation:** Given a string, reverse it without using built-in reverse functions.

2. **List Operations:** Write a program that removes duplicates from a list while preserving the original order of elements.

3. **Tuple Manipulation:** Create a tuple of tuples representing student information (name, age, grade). Write a function to sort the students by grade in descending order.

4. **Range Practice:** Generate a list of even numbers between 1 and 100 using a range object and list comprehension.

5. **String Formatting:** Write a program that takes a sentence as input and capitalizes the first letter of each word.

6. **List Sorting:** Given a list of tuples containing (name, age), sort the list based on age in ascending order.

7. **Tuple Unpacking:** Create a tuple of coordinates (x, y). Write a function that takes this tuple as input and returns the distance of the point from the origin (0, 0).

8. **String Searching:** Write a program that takes a string and a substring as input and counts the number of occurrences of the substring within the string.

9. **List Comprehension:** Generate a list of squares of numbers from 1 to 10 using list comprehension.

10. **Tuple Concatenation:** Create two tuples of integers. Write a program to concatenate these tuples and sort the resulting tuple in ascending order.

Suggested Projects

1. **Sequence Manipulation Tool:** Create a tool that allows users to perform various operations on sequences like lists, tuples, and strings. Operations can include sorting, reversing, merging, and searching within sequences.

2. **Sequence Pattern Matcher:** Develop a program that takes a sequence and a pattern as input and identifies all occurrences of the pattern within the sequence. This could be applied to strings, lists, or tuples.

3. **Sequence Generator:** Build a sequence generator program that generates different types of sequences (e.g., Fibonacci sequence, prime numbers, arithmetic sequences) based on user input parameters.

4. **Sequence Analyzer:** Create a program that analyzes sequences to determine properties such as length, uniqueness of elements, presence of duplicates, and frequency distribution of elements.

5. **Sequence Converter:** Develop a tool that converts sequences between different types (e.g., converting a string to a list or a tuple) while preserving the order and integrity of elements.

6. **Sequence Compression Utility:** Build a program that compresses sequences by identifying repeating patterns and replacing them with a shorter representation, thus reducing the overall size of the sequence.

7. **Sequence Comparator:** Develop a program that compares two sequences and identifies similarities, differences, and common elements between them. This could be useful for tasks like plagiarism detection or version control.

8. **Sequence Alignment Tool:** Create a tool for aligning sequences (e.g., DNA or protein sequences) by identifying regions of similarity and dissimilarity, helping in tasks like sequence comparison and evolutionary analysis.

9. **Sequence Visualization Tool:** Build a program that visualizes sequences using graphs, histograms, or other graphical representations to help users better understand the distribution and patterns within the sequence.

10. **Sequence Encryption/Decryption:** Develop a program that encrypts and decrypts sequences using cryptographic techniques, providing a secure way to transmit and store sensitive information within sequences.10.

Self-Test Questions

1. What are the two primary types of Python sequences?

2. How do you declare an empty list in Python?

3. What are the fundamental characteristics of Python lists, and how do they support various operations like indexing and slicing?

4. What three sequence types are considered immutable?

5. How do tuples differ from lists in Python?

6. Discuss the role of indexing and slicing in accessing elements within Python sequences, highlighting any notable differences between various sequence types.

7. What are the advantages of using Python's built-in sequence functions like `len()`, `min()`, and `max()` over manual implementations?

8. How do you iterate over sequences in Python using loops, comprehensions, and built-in functions like `enumerate()`?

9. Explain the concept of sequence unpacking in Python, and provide examples demonstrating its usage with tuples and lists. (**Note:** *I did not specifically discuss this topic in this chapter, so you may need to do some additional research online.*)

10. Can you describe scenarios where Python's sequence types, such as lists, tuples, and strings, are used interchangeably, and when it's beneficial to use one over the others?

References

The Python Data Model, *https://docs.python.org/3/reference/datamodel.html*

Sequence Types — `list`, `tuple`, and `range`, *https://docs.python.org/3/library/std-*

types.html#sequence-types-list-tuple-range

Emoji Chats, Unicode.org, *https://www.unicode.org/emoji/charts/full-emoji-list.html*

Unicode HowTo, Python.org, *https://docs.python.org/3/howto/unicode.html*

Notes