Object Oriented Programming II

Dr. Burak Kaleci

April 27, 2019

Content

Introduction

- Many of the programs that you have written so far have worked with strings, but only in a limited way.
- The operations that you have performed with strings so far have primarily involved only input and output. For example, you have read strings as input from the keyboard and from files, and sent strings as output to the screen and to files.
- There are many types of programs that not only read strings as input and write strings as output, but also perform operations on strings.
- Word processing programs, for example, manipulate large amounts of text, and thus work extensively with strings. Email programs and search engines are other examples of programs that perform operations on strings.

Accessing the Individual Characters in a String

- Some programming tasks require that you access the individual characters in a string.
- For example, you are probably familiar with websites that require you to set up a password.
- For security reasons, many sites require that your password have at least one uppercase letter, at least one lowercase letter, and at least one digit.
- When you set up your password, a program examines each character to ensure that the password meets these qualifications.

Accessing the Individual Characters in a String

- One of the easiest ways to access the individual characters in a string is to use the for loop.
- Here is the general format:

```
for variable in string:
    statement
    statement
    etc.
```

- In the general format, variable is the name of a variable, and string is either a string literal or a variable that references a string.
- Each time the loop iterates, variable will reference a copy of a character in string, beginning with the first character. We say that the loop iterates over the characters in the string.

Accessing the Individual Characters in a String

- Here is an example:
```
name = 'Juliet'
for ch in name:
    print(ch)
```
- The name variable references a string with six characters, so this loop will iterate six times.
- The first time the loop iterates, the ch variable will reference 'J', the second time the loop iterates the ch variable will reference 'u', and so forth.
- Notice that ch variable references a copy of a character from the string as the loop iterates.
- If we change the value that ch references in the loop, it has no effect on the string referenced by name.

Accessing the Individual Characters in a String

- To demonstrate, look at the following:

```
name = 'Juliet'
for ch in name:
    ch = 'X'
print(name)
```

- The statement in line 3 merely reassigns the ch variable to a different value each time the loop iterates.
- It has no effect on the string 'Juliet' that is referenced by name, and it has no effect on the number of times the loop iterates.
- When this code executes, the statement in line 4 will print Juliet.

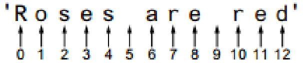
Accessing the Individual Characters in a String

```
# This program counts the number of times
# the letter T (uppercase or lowercase)
# appears in a string.

def main():
    # Create a variable to use to hold the count.
    # The variable must start with 0.
    count = 0

    # Get a string from the user.
    my_string = input('Enter a sentence: ')

    # Count the Ts.
    for ch in my_string:
        if ch == 'T' or ch == 't':
            count += 1

    # Print the result.
    print('The letter T appears', count, 'times.')

# Call the main function.
main()
```

- This program asks the user to enter a string.
- It then uses a for loop to iterate over the string, counting the number of times that the letter T (uppercase or lowercase) appears.

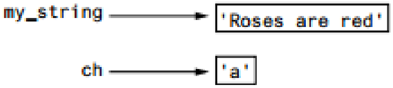
Indexing

- Another way that you can access the individual characters in a string is with an index.
- Each character in a string has an index that specifies its position in the string.
- Indexing starts at 0, so the index of the first character is 0, the index of the second character is 1, and so forth.
- The index of the last character in a string is 1 less than the number of characters in the string.
- Figure shows the indexes for each character in the string 'Roses are red'.
- The string has 13 characters, so the character indexes range from 0 through 12.



Indexing

- You can use an index to retrieve a copy of an individual character in a string, as shown here:
  my_string = 'Roses are red'
  ch = my_string[6]
- The expression my_string[6] in the second statement returns a copy of the character at index 6 in my_string.
- After this statement executes, ch will reference 'a' as shown in Figure.



Indexing

- Here is another example:
  ```
  my_string = 'Roses are red'
  print(my_string[0], my_string[6], my_string[10])
  ```
- This code will print the following:
  ```
  R a r
  ```
- You can also use negative numbers as indexes, to identify character positions relative to the end of the string.
- The Python interpreter adds negative indexes to the length of the string to determine the character position. The index 1 identifies the last character in a string, 2 identifies the next to last character, and so forth.
- The following code shows an example:
  ```
  my_string = 'Roses are red'
  print(my_string[-1], my_string[-2], my_string[-13])
  ```
- This code will print the following:
  ```
  d e R
  ```

IndexError Exceptions

- An IndexError exception will occur if you try to use an index that is out of range for a particular string.
- For example, the string 'Boston' has 6 characters, so the valid indexes are 0 through 5. (The valid negative indexes are 1 through 6.)
- The following is an example of code that causes an IndexError exception:

```
city = 'Boston'
print(city[6])
```

- This type of error is most likely to happen when a loop incorrectly iterates beyond the end of a string, as shown here:

```
city = 'Boston'
index = 0
while index < 7:
    print(city[index])
    index +=1
```

- The last time that this loop iterates, the index variable will be assigned the value 6, which is an invalid index for the string 'Boston'.
- As a result, the print function will cause an IndexError exception to be raised.

The len Function

- The len function can also be used to get the length of a string.
- The following code demonstrates:

  ```
  city = 'Boston'
  size = len(city)
  ```
- The second statement calls the len function, passing the city variable as an argument. The function returns the value 6, which is the length of the string 'Boston'. This value is assigned to the size variable.
- The len function is especially useful to prevent loops from iterating beyond the end of a string, as shown here:

  ```
  city = 'Boston'
  index = 0
  while index < len(city):
      print(city[index])
      index +=1
  ```
- Notice the loop iterates as long as index is less than the length of the string. This is because the index of the last character in a string is always 1 less than the length of the string.

String Concatenation

- A common operation that performed on strings is concatenation, or appending one string to the end of another string.
- The + operator produces a string that is the combination of the two strings used as its operands.
- The following code demonstrates:

```
message = 'Hello ' + 'world'
print(message)
```

- Line 1 concatenates the strings 'Hello' and 'world' to produce the string 'Hello world'.
- The string 'Hello world' is then assigned to the message variable. Line 2 prints the string that is referenced by the message variable.

String Concatenation

- Here is another code that demonstrates concatenation:

```
first_name = 'Emily'
last_name = 'Yeager'
full_name = first_name + ' ' + last_name
print(full_name)
```

- Line 1 assigns the string 'Emily' to the first_name variable.
- Line 2 assigns the string 'Yeager' to the last_name variable.
- Line 3 produces a string that is the concatenation of first_name, followed by a space, followed by last_name. The resulting string is assigned to the full_name variable.
- Line 4 prints the string referenced by full_name.

String Concatenation

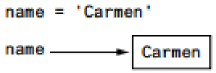
- You can also use the +=operator to perform concatenation.
  ```
  letters = 'abc'
      letters += 'def'
      print(letters)
  ```
- The statement in line 2 performs string concatenation.
- After the statement in line 2 executes, the letters variable will reference the string 'abcdef'.
- Keep in mind that the operand on the left side of the += operator must be an existing variable.
- If you specify a nonexistent variable, an exception is raised.

String Are Immutable

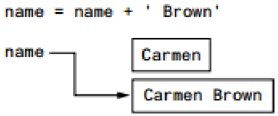
```
1  # This program concatenates strings.
2
3  def main():
4      name  = 'Carmen'
5      print('The name is', name)
6      name  = name + ' Brown'
7      print('Now the name is', name)
8
9  # Call the main function.
10 main()
```

- In Python, strings are immutable, which means once they are created, they cannot be changed.
- Some operations, such as concatenation, give the impression that they modify strings, but in reality they do not.

- The statement in line 4 assigns the string 'Carmen'to the name variable, as shown in Figure.



- The statement in line 6 concatenates the string ' Brown' to the string 'Carmen' and assigns the result to the name variable, as shown in Figure.



- As you can see from the figure, the original string 'Carmen' is not modified. Instead, a new string containing 'Carmen Brown' is created and assigned to the name variable.

String Are Immutable

- Because strings are immutable, you cannot use an expression in the form string[index] on the left side of an assignment operator.
- For example, the following code will cause an error:

```
friend = 'Bill'
friend[0] = 'J'
```

- The last statement in this code will raise an exception because it attempts to change the value of the first character in the string 'Bill'.

String Slicing

- When you take a slice from a string, you get a span of characters from within the string.
- String slices are also called substrings.
- To get a slice of a string, you write an expression in the following general format:

  string[start : end]
- In the general format, start is the index of the first character in the slice, and end is the index marking the end of the slice.
- The expression will return a string containing a copy of the characters from start up to (but not including) end.

String Slicing

- For example, suppose we have the following:
  ```
  full_name = 'Patty Lynn Smith'
  middle_name = full_name[6:10]
  ```
- The second statement assigns the string 'Lynn' to the middle_name variable.
- If you leave out the start index in a slicing expression, Python uses 0 as the starting index.
- Here is an example:
  ```
  full_name = 'Patty Lynn Smith'
  first_name = full_name[:5]
  ```
- The second statement assigns the string 'Patty' to first_name.
- If you leave out the end index in a slicing expression, Python uses the length of the string as the end index.
- Here is an example:
  ```
  full_name = 'Patty Lynn Smith'
  last_name = full_name[11:]
  ```
- The second statement assigns the string 'Smith' to last_name.

String Slicing

- The following codes assign entire string:
  ```
  full_name = 'Patty Lynn Smith'
  my_string = full_name[:]
  ```
- The slicing examples we have seen so far get slices of consecutive characters from strings.
- Slicing expressions can also have step value, which can cause characters to be skipped in the string.
- Here is an example of code that uses a slicing expression with a step value:
  ```
  letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  print(letters[0:26:2])
  ```
- The third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second character from the specified range in the string.
- The code will print the following:
  ```
  ACEGIKMOQSUWY
  ```

String Slicing

- You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the string.
- Here is an example:

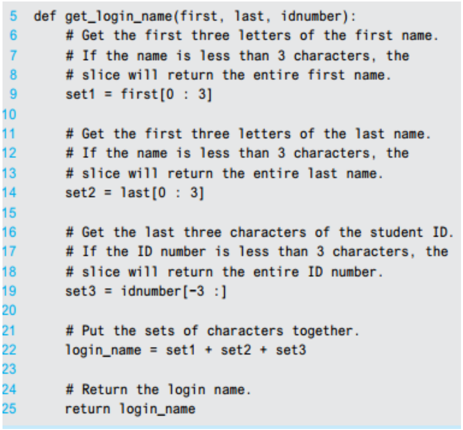
```
full_name = 'Patty Lynn Smith'
last_name = full_name[5:]
```

- The second statement in this code assigns the string 'Smith' to the last_name variable.

Extracting Characters from a String

- At a university, each student is assigned a system login name, which the student uses to log into the campus computer system.
- As part of your internship with the university's Information Technology department, you have been asked to write the code that generates system login names for students.
- You will use the following algorithm to generate a login name:
  - Get the first three characters of the student's first name.
  - Get the first three characters of the student's last name.
  - Get the last three characters of the student's ID number.
  - If the names and ID number are less than three characters in length, use the entire names and numbers
  - Concatenate the three sets of characters to generate the login name.

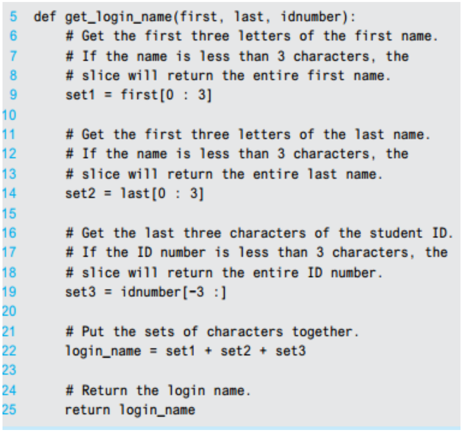
Extracting Characters from a String



```
def get_login_name(first, last, idnumber):
    # Get the first three letters of the first name.
    # If the name is less than 3 characters, the
    # slice will return the entire first name.
    set1 = first[0 : 3]

    # Get the first three letters of the last name.
    # If the name is less than 3 characters, the
    # slice will return the entire last name.
    set2 = last[0 : 3]

    # Get the last three characters of the student ID.
    # If the ID number is less than 3 characters, the
    # slice will return the entire ID number.
    set3 = idnumber[-3 :]

    # Put the sets of characters together.
    login_name = set1 + set2 + set3

    # Return the login name.
    return login_name
```

- For example, if a student's name is Amanda Spencer, and her ID number is ENG6721, her login name would be AmaSpe721.
- You decide to write a function named get_login_name that accepts a student's first name, last name, and ID number as arguments, and returns the student's login name as a string.
- You will save the function in a module named login.py. This module can then be imported into any Python program that needs to generate a login name.

Extracting Characters from a String



```
 5  def get_login_name(first, last, idnumber):
 6      # Get the first three letters of the first name.
 7      # If the name is less than 3 characters, the
 8      # slice will return the entire first name.
 9      set1 = first[0 : 3]
10
11      # Get the first three letters of the last name.
12      # If the name is less than 3 characters, the
13      # slice will return the entire last name.
14      set2 = last[0 : 3]
15
16      # Get the last three characters of the student ID.
17      # If the ID number is less than 3 characters, the
18      # slice will return the entire ID number.
19      set3 = idnumber[-3 :]
20
21      # Put the sets of characters together.
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
```

- The get_login_name function accepts three string arguments: a first name, a last name, and an ID number.
- The statement in line 9 uses a slicing expression to get the first three characters of the string referenced by first and assigns those characters, as a string, to the set1 variable.
- If the string referenced by first is less than three characters long, then the value 3 will be an invalid ending index. If this is the case, Python will use the length of the string as the ending index, and the slicing expression will return the entire string.

Extracting Characters from a String

```
 5  def get_login_name(first, last, idnumber):
 6      # Get the first three letters of the first name.
 7      # If the name is less than 3 characters, the
 8      # slice will return the entire first name.
 9      set1 = first[0 : 3]
10
11      # Get the first three letters of the last name.
12      # If the name is less than 3 characters, the
13      # slice will return the entire last name.
14      set2 = last[0 : 3]
15
16      # Get the last three characters of the student ID.
17      # If the ID number is less than 3 characters, the
18      # slice will return the entire ID number.
19      set3 = idnumber[-3 :]
20
21      # Put the sets of characters together.
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
```

- The statement in line 19 uses a slicing expression to get the last three characters of the string referenced by idnumber and assigns those characters, as a string, to the set3 variable.
- If the string referenced by idnumber is less than three characters, then the value 3 will be an invalid starting index. If this is the case, Python will use 0 as the starting index.
- The statement in line 22 assigns the concatenation of set1, set2, and set3to the login_name variable.
- The variable is returned in line 25.

Extracting Characters from a String

```python
import login

def main():
    # Get the user's first name, last name, and ID number.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')
    idnumber = input('Enter your student ID number: ')

    # Get the login name.
    print('Your system login name is:')
    print(login.get_login_name(first, last, idnumber))

# Call the main function.
main()
```

Testing Strings with in and not in

- In Python, you can use the in operator to determine whether one string is contained in another string.
- Here is the general format of an expression using the in operator with two strings:
  ```
  string1 in string2
  ```
- string1 and string2 can be either string literals or variables referencing strings.
- The expression returns true if string1 is found in string2.
- For example, look at the following code:
  ```
  text = 'Four score and seven years ago'
  if 'seven' in text:
      print('The string "seven" was found.')
  else:
      print('The string "seven" was not found.')
  ```
- This code determines whether the string 'Four score and seven years ago' contains the string 'seven'.
- If we run this code, it will display:
  ```
  The string "seven" was found.
  ```

Testing Strings with in and not in

- You can use the not in operator to determine whether one string is not contained in another string.
- Here is an example:

```
names = 'Bill Joanne Susan Chris Juan Katie'
if 'Pierre' not in names:
    print('Pierre was not found.')
else:
    print('Pierre was found.')
```

- If we run this code, it will display:

```
Pierre was not found.
```

String Methods

- Strings in Python have numerous methods.In this section, we will discuss several string methods for performing the following types of operations:
  - Testing the values of strings
  - Performing various modifications
  - Searching for substrings and replacing sequences of characters
- Here is the general format of a string method call:
  stringvar.method(arguments)
- In the general format, stringvar is a variable that references a string, method is the name of the method that is being called, and arguments is one or more arguments being passed to the method.

String Testing Methods

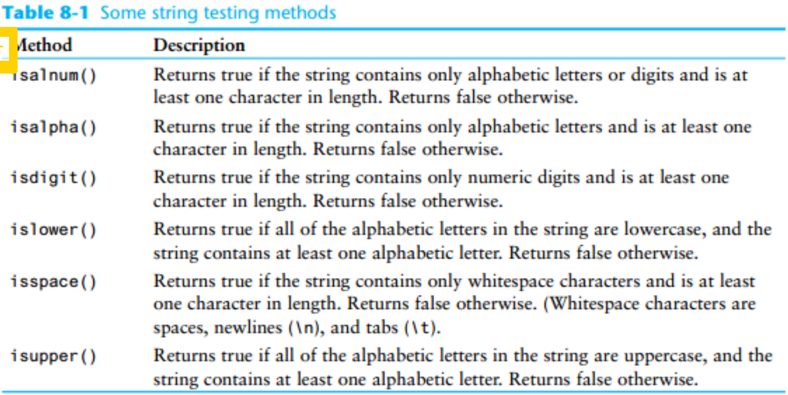


Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

String Testing Methods

- The string methods shown in Table 8-1 test a string for specific characteristics.
- For example, the isdigit method returns true if the string contains only numeric digits. Otherwise, it returns false.
- Here is an example:

```
string1 = '1200'
if string1.isdigit():
    print(string1, 'contains only digits.')
else:
    print(string1, 'contains characters other than digits.')
```

- This code will display

```
1200 contains only digits.
```

String Testing Methods

- Here is another example:
```
string2 = '123abc'
if string2.isdigit():
    print(string2, 'contains only digits.')
else:
    print(string2, 'contains characters other than digits.')
```
- This code will display
```
123abc contains characters other than digits.
```

String Testing Methods

```
# This program demonstrates several string testing methods.

def main():
    # Get a string from the user.
    user_string = input('Enter a string: ')

    print('This is what I found about that string:')

    # Test the string.
    if user_string.isalnum():
        print('The string is alphanumeric.')
    if user_string.isdigit():
        print('The string contains only digits.')
    if user_string.isalpha():
        print('The string contains only alphabetic characters.')
    if user_string.isspace():
        print('The string contains only whitespace characters.')
    if user_string.islower():
        print('The letters in the string are all lowercase.')
    if user_string.isupper():
        print('The letters in the string are all uppercase.')

# Call the string.
main()
```

Modification Methods

- Although strings are immutable, meaning they cannot be modified, they do have a number of methods that return modified versions of themselves.

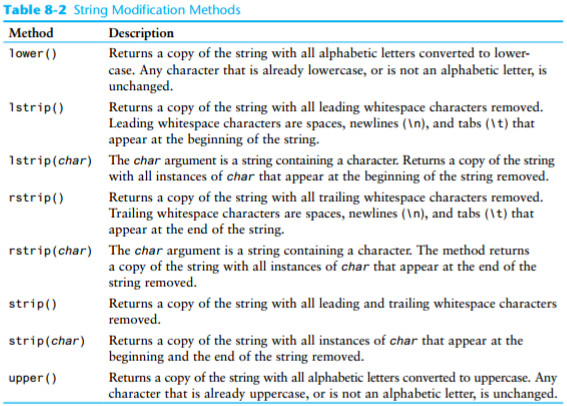


Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(`*`char`*`)`	The *char* argument is a string containing a character. Returns a copy of the string with all instances of *char* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(`*`char`*`)`	The *char* argument is a string containing a character. The method returns a copy of the string with all instances of *char* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(`*`char`*`)`	Returns a copy of the string with all instances of *char* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

Modification Methods

- For example, the lower method returns a copy of a string with all of its alphabetic letters converted to lowercase.
- Here is an example:

```
letters = 'WXYZ'
print(letters, letters.lower())
```

- This code will print

```
WXYZ wxyz
```

- The upper method returns a copy of a string with all of its alphabetic letters converted to uppercase.
- Here is an example:

```
letters = 'abcd'
print(letters, letters.upper())
```

- This code will print

```
abcd ABCD
```

Modification Methods

- The lower and upper methods are useful for making case-insensitive string comparisons.
- String comparisons are case-sensitive, which means the uppercase characters are distinguished from the lowercase characters.
- For example, in a case-sensitive comparison, the string 'abc' is not considered the same as the string 'ABC' or the string 'Abc' because the case of the characters are different.
- Sometimes it is more convenient to perform a case-insensitive comparison, in which the case of the characters is ignored.
- In a case-insensitive comparison, the string 'abc' is considered the same as 'ABC' and 'Abc'.

Modification Methods

- For example, look at the following code:

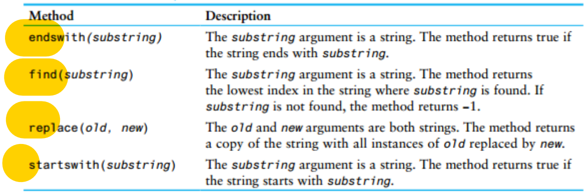
```
again = 'y'
while again.lower() == 'y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

- Notice the last statement in the loop asks the user to enter y to see the message displayed again. The loop iterates as long as the expression again.lower()=='y' is true.
- The expression will be true if the again variable references either 'y' or 'Y'.
- Similar results can be achieved by using the upper method, as shown here:

```
again = 'y'
while again.upper() == 'Y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Searching and Replacing

- Programs commonly need to search for substrings, or strings that appear within other strings.
- For example, suppose you have a document opened in your word processor, and you need to search for a word that appears somewhere in it. The word that you are searching for is a substring that appears inside a larger string, the document.



Method	Description
`endswith(substring)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(substring)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns -1.
`replace(old, new)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(substring)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

Searching and Replacing

- The endswith method determines whether a string ends with a specified substring.
- Here is an example:

```
filename = input('Enter the filename: ')
if filename.endswith('.txt'):
    print('That is the name of a text file.')
elif filename.endswith('.py'):
   print('That is the name of a Python source file.')
elif filename.endswith('.doc'):
    print('That is the name of a word processing document.')
else:
    print('Unknown file type.')
```

- The startswith method works like the endswith method, but determines whether a string begins with a specified substring.

Searching and Replacing

- The find method searches for a specified substring within a string.
- The method returns the lowest index of the substring, if it is found. If the substring is not found, the method returns 1.
- Here is an example:

```
string = 'Four score and seven years ago'
position = string.find('seven')
if position != 1:
    print('The word "seven" was found at index', position)
else:
    print('The word "seven" was not found.')
```

- This code will display

```
The word "seven" was found at index 15
```

Searching and Replacing

- The replac emethod returns a copy of a string, where every occurrence of a specified substring has been replaced with another string.
- For example, look at the following code:

```
string = 'Four score and seven years ago'
new_string = string.replace('years', 'days')
print(new_string)
```

- This code will display

```
Four score and seven days ago
```

The Repetition Operator

- The repetition operator works with strings as well.
- Here is the general format:

  string_to_copy * n
- The repetition operator creates a string that contains n repeated copies of string_to_copy. Here is an example:

  my_string = 'w' * 5
- After this statement executes, my_string will reference the string 'wwwww'.
- Here is another example:

  print('Hello' * 5)
- This statement will print:

  HelloHelloHelloHelloHello

The Repetition Operator

```python
# This program demonstrates the repetition operator.

def main():
    # Print nine rows increasing in length.
    for count in range(1, 10):
        print('Z' * count)

    # Print nine rows decreasing in length.
    for count in range(8, 0, -1):
        print('Z' * count)

# Call the main function.
main()
```

Splitting a String

```
# This program demonstrates the split method.

def main():
    # Create a string with multiple words.
    my_string = 'One two three four'

    # Split the string.
    word_list = my_string.split()

    # Print the list of words.
    print(word_list)

# Call the main function.
main()
```

- Strings in Python have a method named split that returns a list containing the words in the string.
- By default, the split method uses spaces as separators (that is, it returns a list of the words in the string that are separated by spaces).

Splitting a String

```
# This program calls the split method, using the
# '/' character as a separator.

def main():
    # Create a string with a date.
    date_string = '11/26/2018'

    # Split the date.
    date_list = date_string.split('/')

    # Display each piece of the date.
    print('Month:', date_list[0])
    print('Day:', date_list[1])
    print('Year:', date_list[2])

# Call the main function.
main()
```

- You can specify a different separator by passing it as an argument to the split method.

Introduction

- In Python, a dictionary is an object that stores a collection of data.
- Each element that is stored in a dictionary has two parts: a key and a value.
- In fact, dictionary elements are commonly referred to as key-value pairs.
- When you want to retrieve a specific value from a dictionary, you use the key that is associated with that value.

Introduction

- For example, suppose each employee in a company has an ID number, and we want to write a program that lets us look up an employee's name by entering that employee's ID number.
- We could create a dictionary in which each element contains an employee ID number as the key, and that employee's name as the value.
- If we know an employee's ID number, then we can retrieve that employee's name.
- Another example would be a program that lets us enter a person's name and gives us that person's phone number.
- The program could use a dictionary in which each element contains a person's name as the key, and that person's phone number as the value.
- If we know a person's name, then we can retrieve that person's phone number.

Creating a Dictionary

- You can create a dictionary by enclosing the elements inside a set of curly braces ( ).
- An element consists of a key, followed by a colon, followed by a value. The elements are separated by commas.
- The following statement shows an example:
  phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
- This statement creates a dictionary and assigns it to the phonebook variable.
- The dictionary contains the following three elements:
  The first element is 'Chris':'555-1111'. In this element, the key is 'Chris' and the value is '555-1111'.
- The values in a dictionary can be objects of any type, but the keys must be immutable objects.
- For example, keys can be strings, integers, floating-point values, or tuples. Keys cannot be lists or any other type of immutable object.

Retrieving a Value from a Dictionary

- The elements in a dictionary are not stored in any particular order.
- For example, look at the code in which a dictionary is created and its elements are displayed:

  phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}

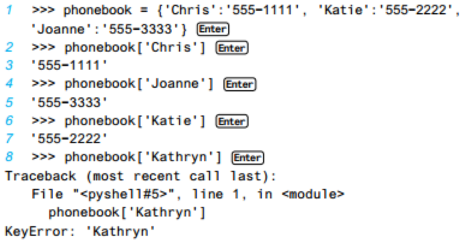
  phonebook
- Notice the order in which the elements are displayed is different than the order in which they were created.
- This illustrates how dictionaries are not sequences, like lists, tuples, and strings.
- As a result, you cannot use a numeric index to retrieve a value by its position from a dictionary.
- Instead, you use a key to retrieve a value

Retrieving a Value from a Dictionary

- To retrieve a value from a dictionary, you simply write an expression in the following general format:

  dictionary_name[key]
- In the general format, dictionary_name is the variable that references the dictionary, and key is a key.
- If the key exists in the dictionary, the expression returns the value that is associated with the key. If the key does not exist, a KeyError exception is raised.

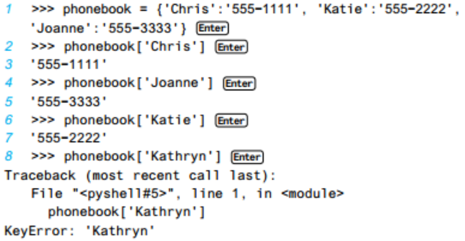
Retrieving a Value from a Dictionary



```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> phonebook['Chris'] Enter
3  '555-1111'
4  >>> phonebook['Joanne'] Enter
5  '555-3333'
6  >>> phonebook['Katie'] Enter
7  '555-2222'
8  >>> phonebook['Kathryn'] Enter
Traceback (most recent call last):
   File "<pyshell#5>", line 1, in <module>
     phonebook['Kathryn']
KeyError: 'Kathryn'
```

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- In line 2, the expression phonebook['Chris'] returns the value from the phonebook dictionary that is associated with the key 'Chris'. The value is displayed in line 3.

Retrieving a Value from a Dictionary



```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> phonebook['Chris'] Enter
3  '555-1111'
4  >>> phonebook['Joanne'] Enter
5  '555-3333'
6  >>> phonebook['Katie'] Enter
7  '555-2222'
8  >>> phonebook['Kathryn'] Enter
Traceback (most recent call last):
   File "<pyshell#5>", line 1, in <module>
     phonebook['Kathryn']
KeyError: 'Kathryn'
```

- In line 4, the expression phonebook['Joanne'] returns the value from the phonebook dictionary that is associated with the key 'Joanne'. The value is displayed in line 5.
- In line 6, the expression phonebook['Katie'] returns the value from the phonebook dictionary that is associated with the key 'Katie'. The value is displayed in line 7.
- In line 8, the expression phonebook['Kathryn'] is entered. There is no such key as 'Kathryn' in the phonebook dictionary, so a KeyError exception is raised.

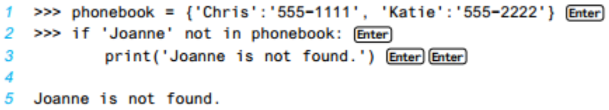
Using the in and not in Operators to Test for a Value in a Dictionary

- As previously demonstrated, a KeyError exception is raised if you try to retrieve a value from a dictionary using a nonexistent key.
- To prevent such an exception, you can use the in operator to determine whether a key exists before you try to use it to retrieve a value.

```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
'Joanne':'555-3333'} Enter
>>> if 'Chris' in phonebook: Enter
        print(phonebook['Chris']) Enter Enter

555-1111
```

Using the in and not in Operators to Test for a Value in a Dictionary

- The if statement in line 2 determines whether the key 'Chris' is in the phonebook dictionary.
- If it is, the statement in line 3 displays the value that is associated with that key.
- You can also use the not in operator to determine whether a key does not exist

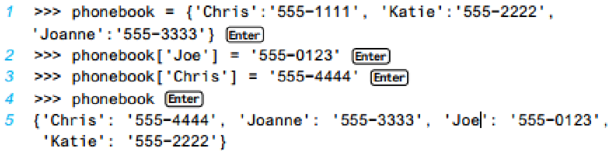


```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
>>> if 'Joanne' not in phonebook: Enter
        print('Joanne is not found.') Enter Enter

Joanne is not found.
```

Adding Elements to an Existing Dictionary

- Dictionaries are mutable objects. You can add new key-value pairs to a dictionary with an assignment statement in the following general format:
  dictionary_name[key] = value
- In the general format, dictionary_name is the variable that references the dictionary, and key is a key.
- If key already exists in the dictionary, its associated value will be changed to value.
- If the key does not exist, it will be added to the dictionary, along with value as its associated value.

Adding Elements to an Existing Dictionary



```
1 >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
  'Joanne':'555-3333'} Enter
2 >>> phonebook['Joe'] = '555-0123' Enter
3 >>> phonebook['Chris'] = '555-4444' Enter
4 >>> phonebook Enter
5 {'Chris': '555-4444', 'Joanne': '555-3333', 'Joe': '555-0123',
   'Katie': '555-2222'}
```

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- The statement in line 2 adds a new key-value pair to the phonebook dictionary. Because there is no key 'Joe' in the dictionary, this statement adds the key 'Joe', along with its associated value '555-0123'.
- The statement in line 3 changes the value that is associated with an existing key.Because the key 'Chris' already exists in the phonebook dictionary, this statement changes its associated value to '555-4444'.
- Line 4 displays the contents of the phonebook dictionary. The output is shown in line 5.

Deleting Elements

- You can delete an existing key-value pair from a dictionary with the del statement.
- Here is the general format:

  del dictionary_name[key]
- In the general format, dictionary_name is the variable that references the dictionary, and key is a key.
- After the statement executes, the key and its associated value will be deleted from the dictionary.
- If the key does not exist, a KeyError exception is raised.

Deleting Elements

```
 1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
 2  >>> phonebook Enter
 3  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
 4  >>> del phonebook['Chris'] Enter
 5  >>> phonebook Enter
 6  {'Joanne': '555-3333', 'Katie': '555-2222'}
 7  >>> del phonebook['Chris'] Enter
 8  Traceback (most recent call last):
 9      File "<pyshell#5>", line 1, in <module>
10          del phonebook['Chris']
11  KeyError: 'Chris'
```

- Line 1 creates a dictionary, and line 2 displays its contents.
- Line 4 deletes the element with the key 'Chris', and line 5 displays the contents of the dictionary. You can see in the output in line 6 that the element no longer exists in the dictionary.
- Line 7 tries to delete the element with the key 'Chris' again. Because the element no longer exists, a KeyError exception is raised.

Getting the Number of Elements in a Dictionary

- You can use the built-in len function to get the number of elements in a dictionary.
- Here is an example:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222'}
num_items = len(phonebook)
print(num_items)
```

- Line 1 creates a dictionary with two elements and assigns it to the phonebook variable.
- Line 2 calls the len function passing the phonebook variable as an argument. The function returns the value 2, which is assigned to the num_items variable.
- Line 3 passes num_items to the print function.

Mixing Data Types in a Dictionary

- As previously mentioned, the keys in a dictionary must be immutable objects, but their associated values can be any type of object. For example, the values can be lists.
- Line 8 retrieves the value that is associated with the key 'Sophie'. The value is displayed in line 9.
- Line 10 retrieves the value that is associated with the key 'Kayla' and assigns it to the kayla_scores variable. After this statement executes, the kayla_scores variable references the list [88, 92, 100].

```
1  >>> test_scores = { 'Kayla' : [88, 92, 100], Enter
2                     'Luis' : [95, 74, 81], Enter
3                     'Sophie' : [72, 88, 91], Enter
4                     'Ethan' : [70, 75, 78] } Enter
5  >>> test_scores Enter
6  {'Kayla': [88, 92, 100], 'Sophie': [72, 88, 91], 'Ethan': [70, 75, 78],
7  'Luis': [95, 74, 81]}
8  >>> test_scores['Sophie'] Enter
9  [72, 88, 91]
10 >>> kayla_scores = test_scores['Kayla'] Enter
11 >>> print(kayla_scores) Enter
12 [88, 92, 100]
```

Mixing Data Types in a Dictionary

- The values that are stored in a single dictionary can be of different types.
- For example, one element's value might be a string, another element's value might be a list, and yet another element's value might be an integer.
- The keys can be of different types, too, as long as they are immutable.

  mixed_up = {'abc':1, 999:'yada yada', (3, 6, 9):[3, 6, 9]}
- The first element is 'abc':1. In this element, the key is the string 'abc' and the value is the integer 1.
- The second element is 999:'yada yada'. In this element, the key is the integer 999 and the value is the string 'yada yada'.
- The third element is (3, 6, 9):[3, 6, 9]. In this element, the key is the tuple (3, 6, 9) and the value is the list [3, 6, 9].

Creating an Empty Dictionary

- Sometimes, you need to create an empty dictionary and then add elements to it as the program executes.
- You can use an empty set of curly braces to create an empty dictionary:

```
phonebook =
phonebook['Chris'] = '555-1111'
phonebook['Katie'] = '555-2222'
phonebook['Joanne'] = '555-3333'
```

- The statement in line 1 creates an empty dictionary and assigns it to the phonebook variable.
- Lines 2 through 4 add key-value pairs to the dictionary.
- You can also use the built-in dict() method to create an empty dictionary, as shown in the following statement:

```
phonebook = dict()
```

- After this statement executes, the phonebook variable will reference an empty dictionary.

Using the for Loop to Iterate over a Dictionary

- You can use the for loop in the following general format to iterate over all the keys in a dictionary:

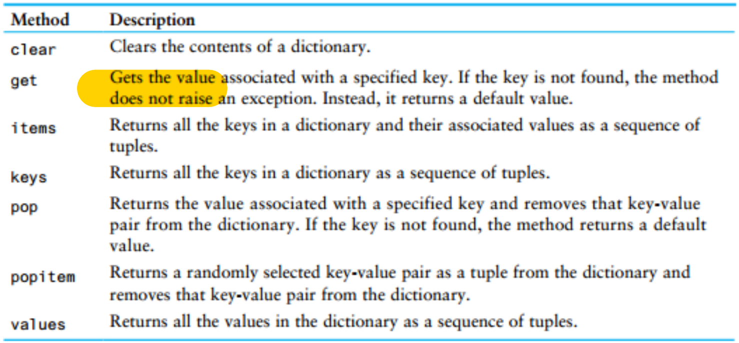
```
for var in dictionary:
    statement
    statement
    etc.
```

- In the general format, var is the name of a variable and dictionary is the name of a dictionary.
- This loop iterates once for each element in the dictionary.
- Each time the loop iterates, var is assigned a key.

Using the for Loop to Iterate over a Dictionary

```
>>> phonebook = {'Chris':'555-1111', Enter
                 'Katie':'555-2222', Enter
                 'Joanne':'555-3333'} Enter
>>> for key in phonebook: Enter
        print(key) Enter Enter


Chris
Joanne
Katie
>>> for key in phonebook: Enter
        print(key, phonebook[key]) Enter Enter


Chris 555-1111
Joanne 555-3333
Katie 555-2222
```

Some Dictionary Methods



Method	Description
`clear`	Clears the contents of a dictionary.
`get`	Gets the value associated with a specified key. If the key is not found, the method does not raise an exception. Instead, it returns a default value.
`items`	Returns all the keys in a dictionary and their associated values as a sequence of tuples.
`keys`	Returns all the keys in a dictionary as a sequence of tuples.
`pop`	Returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value.
`popitem`	Returns a randomly selected key-value pair as a tuple from the dictionary and removes that key-value pair from the dictionary.
`values`	Returns all the values in the dictionary as a sequence of tuples.

The clear Method

- The clear method deletes all the elements in a dictionary, leaving the dictionary empty.
- The method's general format is

  dictionary.clear()
- Notice after the statement in line 4 executes, the phonebook dictionary contains no elements.

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> phonebook Enter
3  {'Chris': '555-1111', 'Katie': '555-2222'}
4  >>> phonebook.clear() Enter
5  >>> phonebook Enter
6  {}
```

The get Method

- You can use the get method as an alternative to the []operator for getting a value from a dictionary.
- The get method does not raise an exception if the specified key is not found.
- Here is the methods general format:

  dictionary.get(key, default)
- In the general format, dictionary is the name of a dictionary, key is a key to search for in the dictionary, and default is a default value to return if the key is not found.
- When the method is called, it returns the value that is associated with the specified key. If the specified key is not found in the dictionary, the method returns default.

The get Method

```
1 >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2 >>> value = phonebook.get('Katie', 'Entry not found') Enter
3 >>> print(value) Enter
4 555-2222
5 >>> value = phonebook.get('Andy', 'Entry not found') Enter
6 >>> print(value) Enter
7 Entry not found
```

- The statement in line 2 searches for the key 'Katie' in the phonebook dictionary. The key is found, so its associated value is returned and assigned to the value variable.
- The statement in line 5 searches for the key 'Andy' in the phonebook dictionary. The key is not found, so the string 'Entry not found' is assigned to the value variable.

The items Method

- The items method returns all of a dictionary's keys and their associated values.
- They are returned as a special type of sequence known as a dictionary view.
- Each element in the dictionary view is a tuple, and each tuple contains a key and its associated value.
- For example, suppose we have created the following dictionary:
  phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
- If we call the phonebook.items() method, it returns the following sequence:
  [('Chris', '555-1111'), ('Joanne', '555-3333'), ('Katie', '555-2222')]

The items Method

- You can use the for loop to iterate over the tuples in the sequence.
- The for loop in lines 4 through 5 calls the phonebook.items() method, which returns a sequence of tuples containing the key-value pairs in the dictionary.
- The loop iterates once for each tuple in the sequence.
- Each time the loop iterates, the values of a tuple are assigned to the key and value variables.
- Line 5 prints the value of the key variable, followed by the value of the value variable.

```
1  >>> phonebook = {'Chris':'555-1111', Enter
2                  'Katie':'555-2222', Enter
3                  'Joanne':'555-3333'} Enter
4  >>> for key, value in phonebook.items(): Enter
5          print(key, value) Enter Enter
```

The keys Method

- The keys method returns all of a dictionary's keys as a dictionary view, which is a type of sequence.
- Each element in the dictionary view is a key from the dictionary.
- For example, suppose we have created the following dictionary:
  phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
- If we call the phonebook.keys() method, it will return the following sequence:
  ['Chris', 'Joanne', 'Katie']

The pop Method

- The pop method returns the value associated with a specified key and removes that key value pair from the dictionary.
- If the key is not found, the method returns a default value.
- Here is the method's general format:

  dictionary.pop(key, default)
- In the general format, dictionary is the name of a dictionary, key is a key to search for in the dictionary, and default is a default value to return if the key is not found.
- When the method is called, it returns the value that is associated with the specified key, and it removes that key-value pair from the dictionary.
- If the specified key is not found in the dictionary, the method returns default.

The pop Method

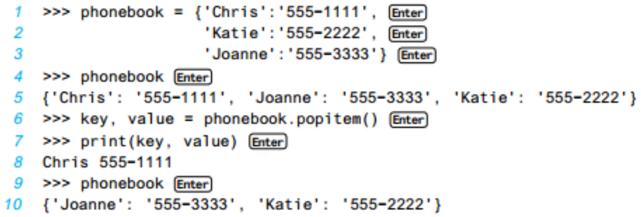
- Line 4 calls the phonebook.pop() method, passing 'Chris' as the key to search for. The value that is associated with the key 'Chris' is returned and assigned to the phone_num variable. The key-value pair containing the key 'Chris' is removed from the dictionary.
- Line 9 calls the phonebook.pop() method, passing 'Andy' as the key to search for. The key is not found, so the string 'Entry not found' is assigned to the phone_num variable.

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                   'Katie':'555-2222', Enter
 3                   'Joanne':'555-3333'} Enter
 4  >>> phone_num = phonebook.pop('Chris', 'Entry not found') Enter
 5  >>> phone_num Enter
 6  '555-1111'
 7  >>> phonebook Enter
 8  {'Joanne': '555-3333', 'Katie': '555-2222'}
 9  >>> phone_num = phonebook.pop('Andy', 'Element not found') Enter
10  >>> phone_num Enter
11  'Element not found'
12  >>> phonebook Enter
13  {'Joanne': '555-3333', 'Katie': '555-2222'}
```

The popitem Method

- The popitem method returns a randomly selected key-value pair, and it removes that key value pair from the dictionary.
- The key-value pair is returned as a tuple.
- Here is the method's general format: dictionary.popitem()
- You can use an assignment statement in the following general format to assign the returned key and value to individual variables:
  k, v= dictionary.popitem()
- This type of assignment is known as a multiple assignment because multiple variables are being assigned at once.
- In the general format, k and vare variables. After the statement executes, k is assigned a randomly selected key from the dictionary, and v is assigned the value associated with that key.
- The key-value pair is removed from the dictionary.

The popitem Method



```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                   'Katie':'555-2222', Enter
 3                   'Joanne':'555-3333'} Enter
 4  >>> phonebook Enter
 5  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
 6  >>> key, value = phonebook.popitem() Enter
 7  >>> print(key, value) Enter
 8  Chris 555-1111
 9  >>> phonebook Enter
10  {'Joanne': '555-3333', 'Katie': '555-2222'}
```

- Line 6 calls the phonebook.popitem()method. The key and value that are returned from the method are assigned to the variables key and value.
- The key-value pair is removed from the dictionary.
- Line 9 displays the contents of the dictionary. The output is shown in line 10. Notice that the key-value pair that was returned from the popitem method in line 6 has been removed.

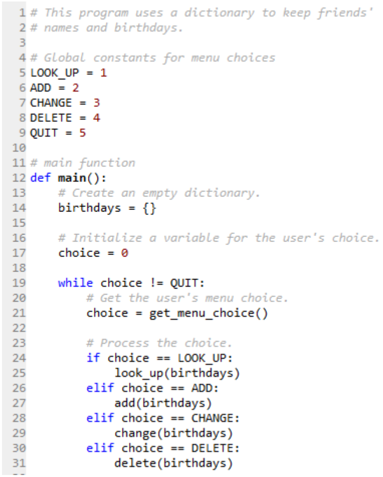
The values Method

- The values method returns all a dictionary's values (without their keys) as a dictionary view, which is a type of sequence.
- Each element in the dictionary view is a value from the dictionary.
- For example, suppose we have created the following dictionary:
  phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
- If we call the phonebook.values() method, it returns the following sequence:
  ['555-1111', '555-2222', '555-3333']

Storing Names and Birthdays in a Dictionary

- In this section, we look at a program that keeps your friends' names and birthdays in a dictionary.
- Each entry in the dictionary uses a friend's name as the key, and that friend's birthday as the value.
- You can use the program to look up your friends' birthdays by entering their names.
- The program displays a menu that allows the user to make one of the following choices:
  - Look up a birthday
  - Add a new birthday
  - Change a birthday
  - Delete a birthday
  - Quit the program
- The program initially starts with an empty dictionary, so you have to choose item 2 from the menu to add a new entry. Once you have added a few entries, you can choose item 1 to look up a specific person's birthday, item 3 to change an existing birthday in the dictionary, item 4 to delete a birthday from the dictionary, or item 5 to quit the program.

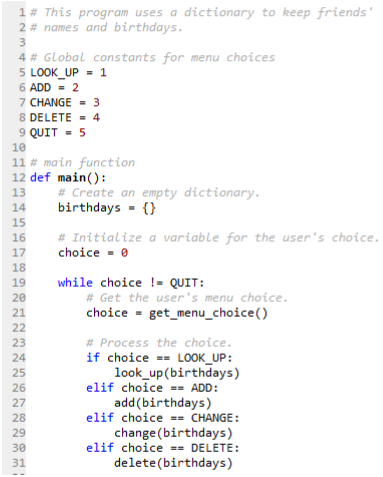
Storing Names and Birthdays in a Dictionary



```
 1 # This program uses a dictionary to keep friends'
 2 # names and birthdays.
 3
 4 # Global constants for menu choices
 5 LOOK_UP = 1
 6 ADD = 2
 7 CHANGE = 3
 8 DELETE = 4
 9 QUIT = 5
10
11 # main function
12 def main():
13     # Create an empty dictionary.
14     birthdays = {}
15
16     # Initialize a variable for the user's choice.
17     choice = 0
18
19     while choice != QUIT:
20         # Get the user's menu choice.
21         choice = get_menu_choice()
22
23         # Process the choice.
24         if choice == LOOK_UP:
25             look_up(birthdays)
26         elif choice == ADD:
27             add(birthdays)
28         elif choice == CHANGE:
29             change(birthdays)
30         elif choice == DELETE:
31             delete(birthdays)
```

- The global constants that are declared in lines 5 through 9 are used to test the user's menu selection.
- Inside the main function, line 14 creates an empty dictionary referenced by the birthdays variable.
- Line 17 initializes the choice variable with the value 0. This variable holds the user's menu selection.
- The while loop that begins in line 19 repeats until the user chooses to quit the program.
- Inside the loop, line 21 calls the get_menu_choice function.
- The get_menu_choice function displays the menu and returns the user's selection.
- The value that is returned is assigned to the choice variable.

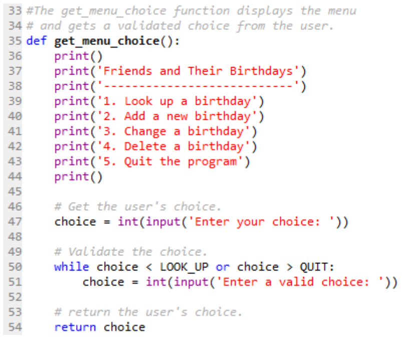
Storing Names and Birthdays in a Dictionary



```
 1 # This program uses a dictionary to keep friends'
 2 # names and birthdays.
 3
 4 # Global constants for menu choices
 5 LOOK_UP = 1
 6 ADD = 2
 7 CHANGE = 3
 8 DELETE = 4
 9 QUIT = 5
10
11 # main function
12 def main():
13     # Create an empty dictionary.
14     birthdays = {}
15
16     # Initialize a variable for the user's choice.
17     choice = 0
18
19     while choice != QUIT:
20         # Get the user's menu choice.
21         choice = get_menu_choice()
22
23         # Process the choice.
24         if choice == LOOK_UP:
25             look_up(birthdays)
26         elif choice == ADD:
27             add(birthdays)
28         elif choice == CHANGE:
29             change(birthdays)
30         elif choice == DELETE:
31             delete(birthdays)
```

- The if-elif statement in lines 24 through 31 processes the user's menu choice.
- If the user selects item 1, line 25 calls the look_up function.
- If the user selects item 2, line 27 calls the add function.
- If the user selects item 3, line 29 calls the change function.
- If the user selects item 4, line 31 calls the delete function.

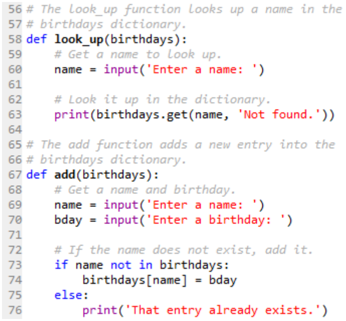
Storing Names and Birthdays in a Dictionary



```
33 #The get_menu_choice function displays the menu
34 # and gets a validated choice from the user.
35 def get_menu_choice():
36     print()
37     print('Friends and Their Birthdays')
38     print('---------------------------')
39     print('1. Look up a birthday')
40     print('2. Add a new birthday')
41     print('3. Change a birthday')
42     print('4. Delete a birthday')
43     print('5. Quit the program')
44     print()
45 
46     # Get the user's choice.
47     choice = int(input('Enter your choice: '))
48 
49     # Validate the choice.
50     while choice < LOOK_UP or choice > QUIT:
51         choice = int(input('Enter a valid choice: '))
52 
53     # return the user's choice.
54     return choice
```

- The statements in lines 36 through 44 display the menu on the screen.
- Line 47 prompts the user to enter his or her choice. The input is converted to an int and assigned to the choice variable.
- The while loop in lines 50 through 51 validates the user's input and, if necessary, prompts the user to reenter his or her choice.
- Once a valid choice is entered, it is returned from the function in line 54.

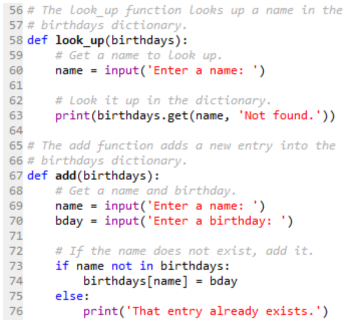
Storing Names and Birthdays in a Dictionary



```
56 # The look_up function looks up a name in the
57 # birthdays dictionary.
58 def look_up(birthdays):
59     # Get a name to look up.
60     name = input('Enter a name: ')
61
62     # Look it up in the dictionary.
63     print(birthdays.get(name, 'Not found.'))
64
65 # The add function adds a new entry into the
66 # birthdays dictionary.
67 def add(birthdays):
68     # Get a name and birthday.
69     name = input('Enter a name: ')
70     bday = input('Enter a birthday: ')
71
72     # If the name does not exist, add it.
73     if name not in birthdays:
74         birthdays[name] = bday
75     else:
76         print('That entry already exists.')
```

- The purpose of the look_up function is to allow the user to look up a friend's birthday.
- It accepts the dictionary as an argument. Line 60 prompts the user to enter a name, and line 63 passes that name as an argument to the dictionary's get function.
- If the name is found, its associated value (the friend's birthday) is returned and displayed.
- If the name is not found, the string Not found. is displayed

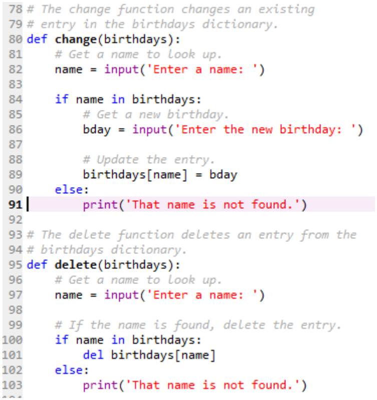
Storing Names and Birthdays in a Dictionary



```
56 # The look_up function looks up a name in the
57 # birthdays dictionary.
58 def look_up(birthdays):
59     # Get a name to look up.
60     name = input('Enter a name: ')
61 
62     # Look it up in the dictionary.
63     print(birthdays.get(name, 'Not found.'))
64 
65 # The add function adds a new entry into the
66 # birthdays dictionary.
67 def add(birthdays):
68     # Get a name and birthday.
69     name = input('Enter a name: ')
70     bday = input('Enter a birthday: ')
71 
72     # If the name does not exist, add it.
73     if name not in birthdays:
74         birthdays[name] = bday
75     else:
76         print('That entry already exists.')
```

- The purpose of the add function is to allow the user to add a new birthday to the dictionary.
- It accepts the dictionary as an argument. Lines 69 and 70 prompt the user to enter a name and a birthday.
- The if statement in line 73 determines whether the name is not already in the dictionary. If not, line 74 adds the new name and birthday to the dictionary.
- Otherwise, a message indicating that the entry already exists is printed in line 76.

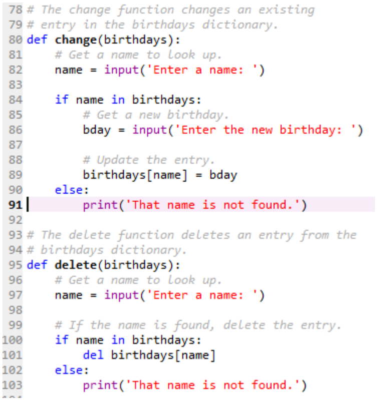
Storing Names and Birthdays in a Dictionary



```
 78 # The change function changes an existing
 79 # entry in the birthdays dictionary.
 80 def change(birthdays):
 81     # Get a name to look up.
 82     name = input('Enter a name: ')
 83 
 84     if name in birthdays:
 85         # Get a new birthday.
 86         bday = input('Enter the new birthday: ')
 87 
 88         # Update the entry.
 89         birthdays[name] = bday
 90     else:
 91         print('That name is not found.')
 92 
 93 # The delete function deletes an entry from the
 94 # birthdays dictionary.
 95 def delete(birthdays):
 96     # Get a name to look up.
 97     name = input('Enter a name: ')
 98 
 99     # If the name is found, delete the entry.
100     if name in birthdays:
101         del birthdays[name]
102     else:
103         print('That name is not found.')
```

- The purpose of the change function is to allow the user to change an existing birthday in the dictionary.
- It accepts the dictionary as an argument. Line 82 gets a name from the user.
- The if statement in line 84 determines whether the name is in the dictionary. If so, line 86 gets the new birthday, and line 89 stores that birthday in the dictionary.
- If the name is not in the dictionary, line 91 prints a message indicating so.

Storing Names and Birthdays in a Dictionary



```
 78 # The change function changes an existing
 79 # entry in the birthdays dictionary.
 80 def change(birthdays):
 81     # Get a name to look up.
 82     name = input('Enter a name: ')
 83 
 84     if name in birthdays:
 85         # Get a new birthday.
 86         bday = input('Enter the new birthday: ')
 87 
 88         # Update the entry.
 89         birthdays[name] = bday
 90     else:
 91         print('That name is not found.')
 92 
 93 # The delete function deletes an entry from the
 94 # birthdays dictionary.
 95 def delete(birthdays):
 96     # Get a name to look up.
 97     name = input('Enter a name: ')
 98 
 99     # If the name is found, delete the entry.
100     if name in birthdays:
101         del birthdays[name]
102     else:
103         print('That name is not found.')
```

- The purpose of the delete function is to allow the user to delete an existing birthday from the dictionary.
- It accepts the dictionary as an argument. Line 97 gets a name from the user.
- The if statement in line 100 determines whether the name is in the dictionary. If so, line 101 deletes it.
- If the name is not in the dictionary, line 103 prints a message indicating so.

Introduction

- A set is an object that stores a collection of data in the same way as mathematical sets. Here are some important things to know about sets:
  - All the elements in a set must be unique. No two elements can have the same value.
  - Sets are unordered, which means that the elements in a set are not stored in any particular order.
  - The elements that are stored in a set can be of different data types.

Creating a Set

- To create a set, you have to call the built-in set function.
- Here is an example of how you create an empty set:

  ```
  myset = set()
  ```
- After this statement executes, the myset variable will reference an empty set.
- You can also pass one argument to the set function. The argument that you pass must be an object that contains iterable elements, such as a list, a tuple, or a string.
- The individual elements of the object that you pass as an argument become elements of the set.
- Here is an example:

  ```
  myset = set(['a', 'b', 'c'])
  ```
- In this example, we are passing a list as an argument to the set function.
- After this statement executes, the myset variable references a set containing the elements 'a', 'b', and 'c'.

Creating a Set

- If you pass a string as an argument to the set function, each individual character in the string becomes a member of the set.
- Here is an example:
  myset = set('abc')
- After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.
- Sets cannot contain duplicate elements. If you pass an argument containing duplicate elements to the set function, only one of the duplicated elements will appear in the set.
- Here is an example:
  myset = set('aaabc')
- The character 'a' appears multiple times in the string, but it will appear only once in the set. After this statement executes, the myset variable will reference a set containing the elements 'a', 'b', and 'c'.

Creating a Set

- What if you want to create a set in which each element is a string containing more than one character? For example, how would you create a set containing the elements 'one', 'two', and 'three'?
- The following code does not accomplish the task, because you can pass no more than one argument to the set function:

  This is an ERROR!

  myset = set('one', 'two', 'three')
- The following does not accomplish the task either:

  This does not do what we intend.

  myset = set('one two three')
- After this statement executes, the myset variable will reference a set containing the elements 'o', 'n', 'e', ' ', 't', 'w', 'h', and 'r'.

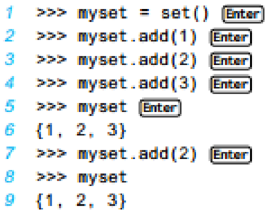
Introduction

- To create the set that we want, we have to pass a list containing the strings 'one', 'two', and 'three' as an argument to the set function.
- Here is an example:

  OK, this works.

  myset = set(['one', 'two', 'three'])
- After this statement executes, the mysetvariable will reference a set containing the elements 'one', 'two', and 'three'.

Adding and Removing Elements



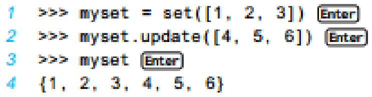
```
1  >>> myset = set() Enter
2  >>> myset.add(1) Enter
3  >>> myset.add(2) Enter
4  >>> myset.add(3) Enter
5  >>> myset Enter
6  {1, 2, 3}
7  >>> myset.add(2) Enter
8  >>> myset
9  {1, 2, 3}
```

- Sets are mutable objects, so you can add items to them and remove items from them. You use the add method to add an element to a set.

- The statement in line 1 creates an empty set and assigns it to the myset variable.
- The statements in lines 2 through 4 add the values 1, 2, and 3 to the set.
- Line 5 displays the contents of the set, which is shown in line 6.
- The statement in line 7 attempts to add the value 2 to the set. The value 2 is already in the set, however.
- If you try to add a duplicate item to a set with the add method, the method does not raise an exception. It simply does not add the item

Adding and Removing Elements

- You can add a group of elements to a set all at one time with the update method.
- When you call the update method as an argument, you pass an object that contains iterable elements, such as a list, a tuple, string, or another set.
- The individual elements of the object that you pass as an argument become elements of the set.



```
>>> myset = set([1, 2, 3]) Enter
>>> myset.update([4, 5, 6]) Enter
>>> myset Enter
{1, 2, 3, 4, 5, 6}
```

Adding and Removing Elements

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> myset.update('abc') Enter
3  >>> myset Enter
4  {'a', 1, 2, 3, 'c', 'b'}
```

- The statement in line 1 creates a set containing the values 1, 2, and 3.
- Line 2 calls the myset.update method, passing the string 'abc' as an argument.
- This causes the each character of the string to be added as an element to myset.

Adding and Removing Elements

```
>>> myset = set([1, 2, 3, 4, 5]) Enter
>>> myset Enter
{1, 2, 3, 4, 5}
>>> myset.remove(1) Enter
>>> myset Enter
{2, 3, 4, 5}
>>> myset.discard(5) Enter
>>> myset Enter
{2, 3, 4}
>>> myset.discard(99) Enter
>>> myset.remove(99) Enter
Traceback (most recent call last):
   File "<pyshell#12>", line 1, in <module>
      myset.remove(99)
KeyError: 99
```

- You can remove an item from a set with either the remove method or the discard method.
- You pass the item that you want to remove as an argument to either method, and that item is removed from the set.
- The only difference between the two methods is how they behave when the specified item is not found in the set.
- The remove method raises a KeyError exception, but the discard method does not raise an exception.

Adding and Removing Elements

```
 1  >>> myset = set([1, 2, 3, 4, 5]) Enter
 2  >>> myset Enter
 3  {1, 2, 3, 4, 5}
 4  >>> myset.remove(1) Enter
 5  >>> myset Enter
 6  {2, 3, 4, 5}
 7  >>> myset.discard(5) Enter
 8  >>> myset Enter
 9  {2, 3, 4}
10  >>> myset.discard(99) Enter
11  >>> myset.remove(99) Enter
12  Traceback (most recent call last):
13    File "<pyshell#12>", line 1, in <module>
14      myset.remove(99)
15  KeyError: 99
```

- Line 1 creates a set with the elements 1, 2, 3, 4, and 5.
- Line 2 displays the contents of the set, which is shown in line 3.
- Line 4 calls the remove method to remove the value 1 from the set.
- You can see in the output shown in line 6 that the value 1 is no longer in the set.
- Line 7 calls the discard method to remove the value 5 from the set.

Adding and Removing Elements

```
1  >>> myset = set([1, 2, 3, 4, 5]) Enter
2  >>> myset Enter
3  {1, 2, 3, 4, 5}
4  >>> myset.remove(1) Enter
5  >>> myset Enter
6  {2, 3, 4, 5}
7  >>> myset.discard(5) Enter
8  >>> myset Enter
9  {2, 3, 4}
10 >>> myset.discard(99) Enter
11 >>> myset.remove(99) Enter
12 Traceback (most recent call last):
13   File "<pyshell#12>", line 1, in <module>
14     myset.remove(99)
15 KeyError: 99
```

- You can see in the output in line 9 that the value 5 is no longer in the set.
- Line 10 calls the discard method to remove the value 99 from the set. The value is not found in the set, but the discard method does not raise an exception.
- Line 11 calls the remove method to remove the value 99 from the set. Because the value is not in the set, a KeyError exception is raised, as shown in lines 12 through 15.

Using the for Loop to Iterate over a Set

- You can use the for loop in the following general format to iterate over all the elements in a set:

```
for var in set:
    statement
    statement
    etc.
```

- In the general format, var is the name of a variable and set is the name of a set. This loop iterates once for each element in the set. Each time the loop iterates, var is assigned an element.
- Lines 2 through 3 contain a for loop that iterates once for each element of the myset set.
- Each time the loop iterates, an element of the set is assigned to the val variable. Line 3 prints the value of the val variable.

```
1  >>> myset = set(['a', 'b', 'c']) Enter
2  >>> for val in myset: Enter
3          print(val) Enter Enter
```

Using the in and not in Operators to Test for a Value in a Set

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 1 in myset: Enter
3          print('The value 1 is in the set.') Enter Enter
4
5  The value 1 is in the set.
```

- You can use the in operator to determine whether a value exists in a set.
- The if statement in line 2 determines whether the value 1 is in the myset set. If it is, the statement in line 3 displays a message.
- You can also use the not inoperator to determine if a value does not exist in a set.

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 99 not in myset: Enter
3          print('The value 99 is not in the set.') Enter Enter
```

Finding the Union of Sets

- The union of two sets is a set that contains all the elements of both sets.
- In Python, you can call the union method to get the union of two sets.
- Here is the general format:
  set1.union(set2)
- In the general format, set1 and set2 are sets. The method returns a set that contains the elements of both set1 and set2.
- The statement in line 3 calls the set1 object's union method, passing set2 as an argument.
- The method returns a set that contains all the elements of set1 and set2 (without duplicates, of course). The resulting set is assigned to the set3 variable.

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1.union(set2) Enter
4  >>> set3 Enter
5  {1, 2, 3, 4, 5, 6}
```

Finding the Union of Sets

- You can also use the | operator to find the union of two sets.
- Here is the general format of an expression using the | operator with two sets:

  set1 | set2
- In the general format, set1 and set2 are sets.
- The expression returns a set that contains the elements of both set1 and set2.

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 | set2 Enter
>>> set3 Enter
{1, 2, 3, 4, 5, 6}
```

Finding the Intersection of Sets

- The intersection of two sets is a set that contains only the elements that are found in both sets.
- In Python, you can call the intersection method to get the intersection of two sets.
- Here is the general format: set1.intersection(set2)
- In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in both set1 and set2.
- The statement in line 3 calls the set1 object's intersection method, passing set2 as an argument.
- The method returns a set that contains the elements that are found in both set1 and set2. The resulting set is assigned to the set3 variable.

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1.intersection(set2) Enter
4  >>> set3 Enter
5  {3, 4}
```

Finding the Intersection of Sets

- You can also use the & operator to find the intersection of two sets.
- Here is the general format of an expression using the & operator with two sets:

  set1 & set2
- In the general format, set1 and set2 are sets.
- The expression returns a set that contains the elements that are found in both set1 and set2.

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 & set2 Enter
>>> set3 Enter
{3, 4}
```

Finding the Difference of Sets

- The difference of set1 and set2 is the elements that appear in set1 but do not appear in set2.
- In Python, you can call the difference method to get the difference of two sets.
- Here is the general format:
  set1.difference(set2)
- In the general format, set1 and set2 are sets. The method returns a set that contains the elements that are found in set1 but not in set2.

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1.difference(set2) Enter
>>> set3 Enter
{1, 2}
```

Finding the Difference of Sets

- You can also use the − operator to find the difference of two sets.
- Here is the general format of an expression using the − operator with two sets:

  set1 − set2
- In the general format, set1 and set2 are sets.
- The expression returns a set that contains the elements that are found in set1 but not in set2.

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 - set2 Enter
>>> set3 Enter
{1, 2}
```

Finding the Symmetric Difference of Sets

- The symmetric difference of two sets is a set that contains the elements that are not shared by the sets.
- In other words, it is the elements that are in one set but not in both.
- In Python, you can call the symmetric_difference method to get the symmetric difference of two sets.
- Here is the general format:

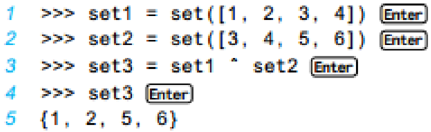
  set1.symmetric_difference(set2)
- In the general format, set1 and set2 are sets.
- The method returns a set that contains the elements that are found in either set1 or set2 but not both sets.

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1.symmetric_difference(set2) Enter
>>> set3 Enter
{1, 2, 5, 6}
```

Finding the Symmetric Difference of Sets

- You can also use the ˆ operator to find the symmetric difference of two sets.
- Here is the general format of an expression using the ˆ operator with two sets:

  set1 ˆ set2
- In the general format, set1 and set2 are sets.
- The expression returns a set that contains the elements that are found in either set1 or set2, but not both sets.



```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 ^ set2 Enter
>>> set3 Enter
{1, 2, 5, 6}
```

Finding Subsets and Supersets

- Suppose you have two sets, and one of those sets contains all of the elements of the other set.
- Here is an example:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

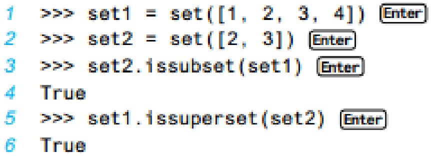
- In this example, set1 contains all the elements of set2, which means that set2 is a subset of set1.
- It also means that set1 is a superset of set2.
- In Python, you can call the issubset method to determine whether one set is a subset of another.
- Here is the general format:

```
set2.issubset(set1)
```

- In the general format, set1 and set2are sets. The method returns True if set2 is a subset of set1. Otherwise, it returns False.

Finding Subsets and Supersets

- You can call the issuperset method to determine whether one set is a superset of another.
- Here is the general format:
  set1.issuperset(set2)
- In the general format, set1 and set2 are sets. The method returns True if set1 is a superset of set2. Otherwise, it returns False.



```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([2, 3]) Enter
>>> set2.issubset(set1) Enter
True
>>> set1.issuperset(set2) Enter
True
```

Finding Subsets and Supersets

- You can also use the <= operator to determine whether one set is a subset of another and the >= operator to determine whether one set is a superset of another.
- Here is the general format of an expression using the <= operator with two sets:

  set2 <= set1
- Here is the general format of an expression using the >= operator with two sets: set1 >= set2

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([2, 3]) Enter
>>> set2 <= set1 Enter
True
>>> set1 >= set2 Enter
True
>>> set1 <= set2 Enter
False
```