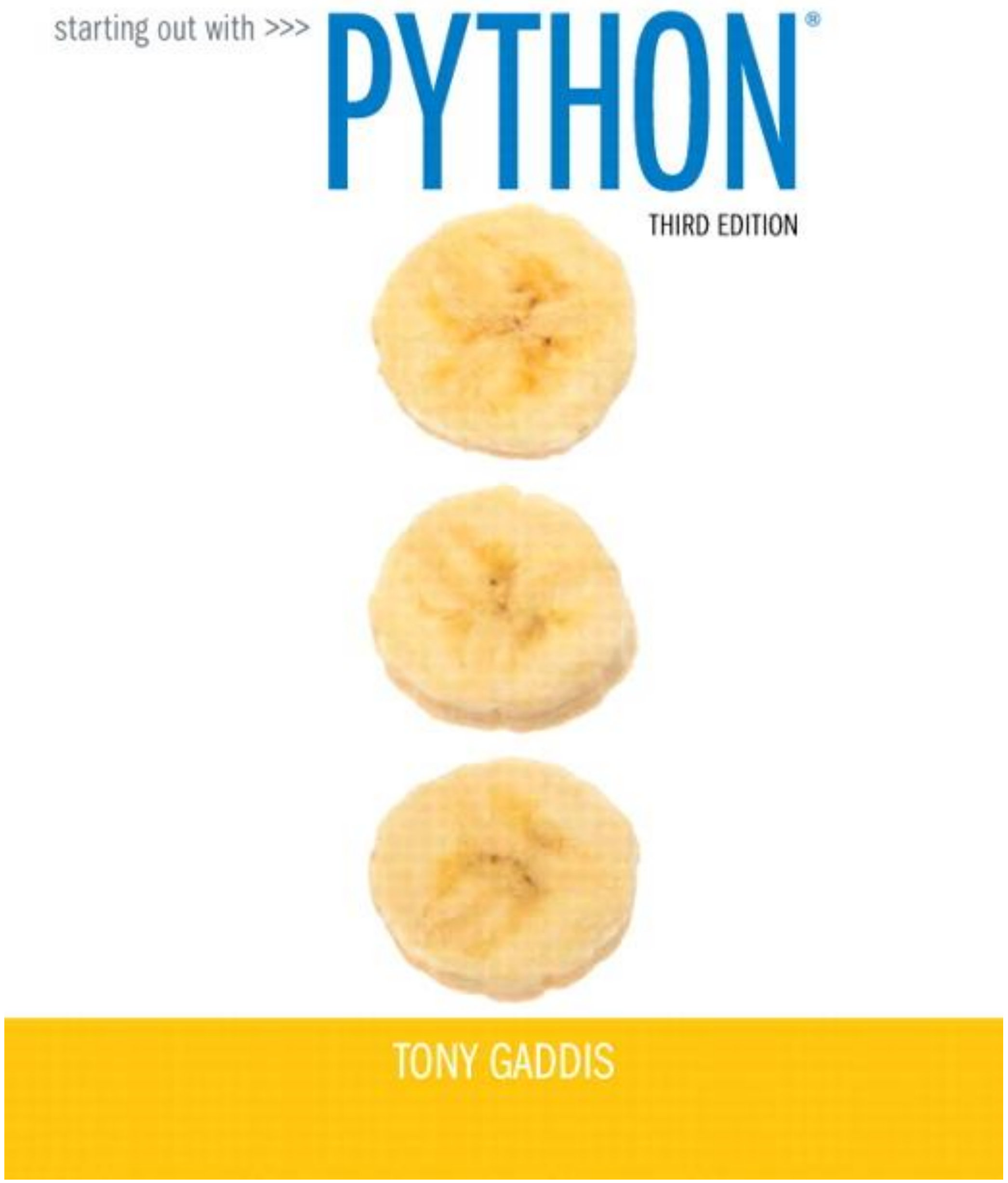
CHAPTER 8

More About Strings



Addison-Wesley
is an imprint of
PEARSON

Topics

- Basic String Operations
- String Slicing
- Testing, Searching, and Manipulating Strings

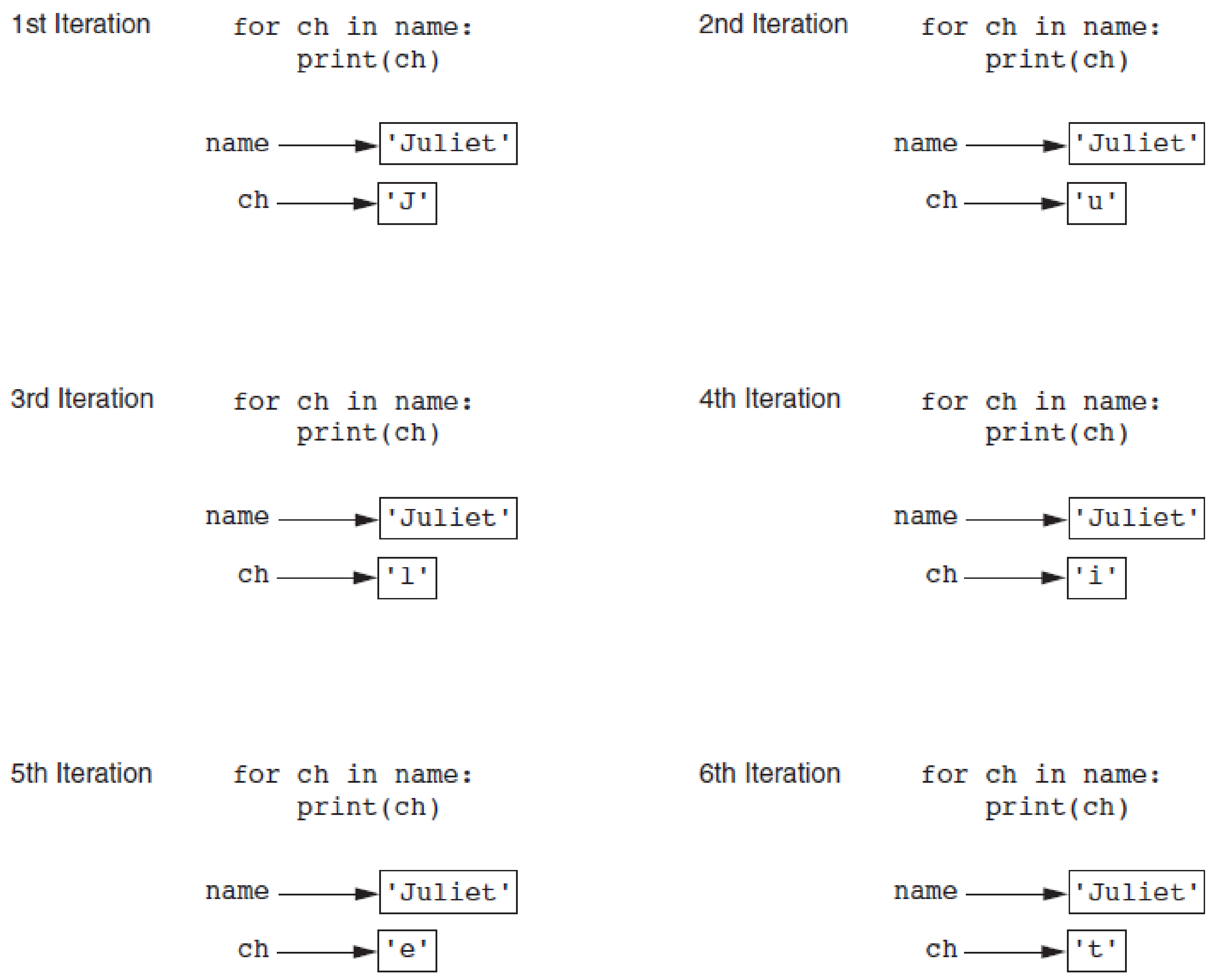
Basic String Operations

- **Many types of programs perform operations on strings**
- **In Python, many tools for examining and manipulating strings**
  - Strings are sequences, so many of the tools that work with sequences work with strings

Accessing the Individual Characters in a String

- **To access an individual character in a string:**
  - Use a `for` loop
    - Format: `for character in string:`
    - Useful when need to iterate over the whole string, such as to count the occurrences of a specific character
  - Use indexing
    - Each character has an index specifying its position in the string, starting at 0
    - Format: `character = my_string[i]`

Figure 8-1 Iterating over the string `'Juliet'`



```
# 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()
```

Accessing the Individual Characters in a String (cont'd.)

Figure 8-2 String indexes

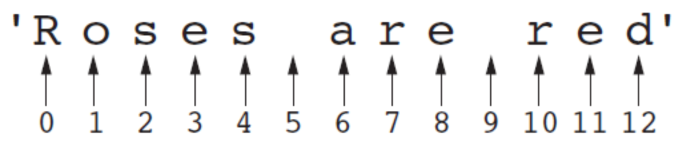
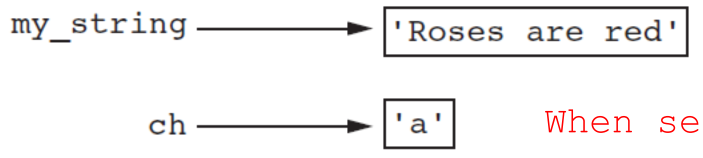


Figure 8-3 Getting a copy of a character from a string



When setting ch=my_string[6]

Accessing the Individual Characters in a String (cont'd.)

- **`IndexError` exception will occur if:**
  - You try to use an index that is out of range for the string
    - Likely to happen when loop iterates beyond the end of the string
- **`len(`*`string`*`)` function can be used to obtain the length of a string**
  - Useful to prevent loops from iterating beyond the end of a string

String Concatenation

- **Concatenation: appending one string to the end of another string**
  - Use the + operator to produce a string that is a combination of its operands
  - The augmented assignment operator += can also be used to concatenate strings
    - The operand on the left side of the += operator must be an existing variable; otherwise, an exception is raised

Strings Are Immutable

- **Strings are immutable**
  - Once they are created, they cannot be changed
    - Concatenation doesn't actually change the existing string, but rather creates a new string and assigns the new string to the previously used variable
  - Cannot use an expression of the form

```
string[index] = new_character
```

  - Statement of this type will raise an exception

Strings Are Immutable (cont'd.)

```
def main():
    name = 'Carmen'
    print('The name is', name)
    name = name + ' Brown'
    print('Now the name is', name)

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

Figure 8-4 The string 'Carmen' assigned to `name`

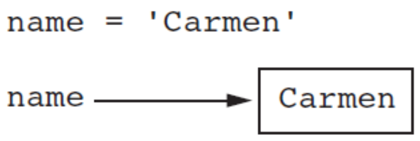
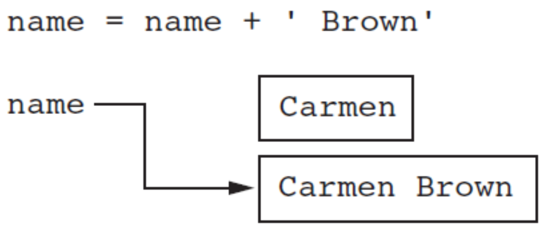


Figure 8-5 The string 'Carmen Brown' assigned to `name`



String Slicing

- **Slice: span of items taken from a sequence, known as *substring***
  - Slicing format: `string[start : end]`
    - Expression will return a string containing a copy of the characters from `start` up to, but not including, `end`
    - If `start` not specified, `0` is used for start index
    - If `end` not specified, `len(string)` is used for end index
  - Slicing expressions can include a step value and negative indexes relative to end of string

```
>>> full_name = 'Patty Lynn Smith'
>>> middle_name = full_name[6:10]
>>> print (middle_name)
Lynn
>>> first_name = full_name[:5]
>>> print (first_name)
Patty
>>> last_name = full_name[11:]
>>> print (last_name)
Smith
>>> mystring = full_name[0:len(full_name)]
>>> print (my_string)
>>> print (mystring)
Patty Lynn Smith
>>> print (full_name[0:len(full_name):2])
PtyLn mt
>>>
```

Testing, Searching, and Manipulating Strings

- **You can use the `in` operator to determine whether one string is contained in another string**
  - General format: *`string1 in string2`*
    - *`string1`* and *`string2`* can be string literals or variables referencing strings
- **Similarly you can use the `not in` operator to determine whether one string is not contained in another string**

```
>>> text = 'Four score and seven years ago'
>>> 'seven' in text
True
>>> 'ok' in text
False
>>> 'seve' in text
True
>>> 'e ' in text
True
>>> 'Seven' in text
False
```

String Methods

- **Strings in Python have many types of methods, divided into different types of operations**
  - General format:

    `*mystring*.*method*(*arguments*)`
- **Some methods test a string for specific characteristics**
  - Generally Boolean methods, that return `True` if a condition exists, and `False` otherwise

String Methods (cont'd.)

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.

```
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 main function.
main();
```

String Methods (cont'd.)

- **Some methods return a copy of the string, to which modifications have been made**
  - Simulate strings as mutable objects
- **String comparisons are case-sensitive**
  - Uppercase characters are distinguished from lowercase characters
  - `lower` and `upper` methods can be used for making case-insensitive string comparisons

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

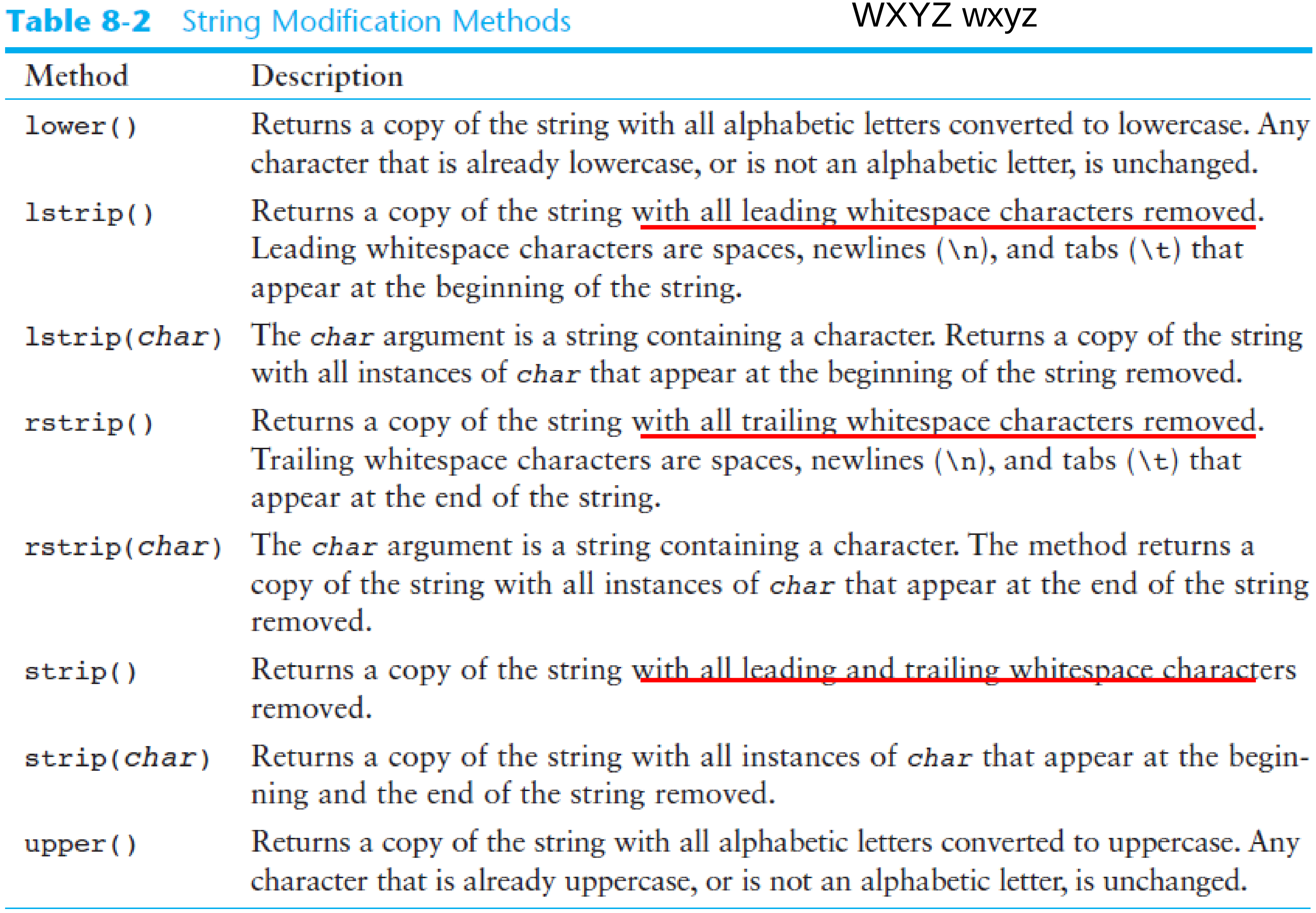


Table 8-2 String Modification Methods

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.

String Methods (cont'd.)

- **Programs commonly need to search for substrings**
- **Several methods to accomplish this:**
  - `endswith(`*`substring`*`)`: checks if the string ends with *`substring`*
    - Returns `True` or `False`
  - `startswith(`*`substring`*`)`: checks if the string starts with *`substring`*
    - Returns `True` or `False`

String Methods (cont'd.)

- **Several methods to accomplish this (cont'd):**
  - `find(`*`substring`*`)`: searches for *`substring`* within the string
    - Returns lowest index of the substring, or if the substring is not contained in the string, returns -1
  - `replace(`*`substring, new_string`*`)`:
    - Returns a copy of the string where every occurrence of *`substring`* is replaced with *`new_string`*

```
>>> filename = input('enter file name')
enter file name AA.py
>>> if filename.endswith('.py'):
...     print('This is a python file')
...
This is a python file
>>>
```

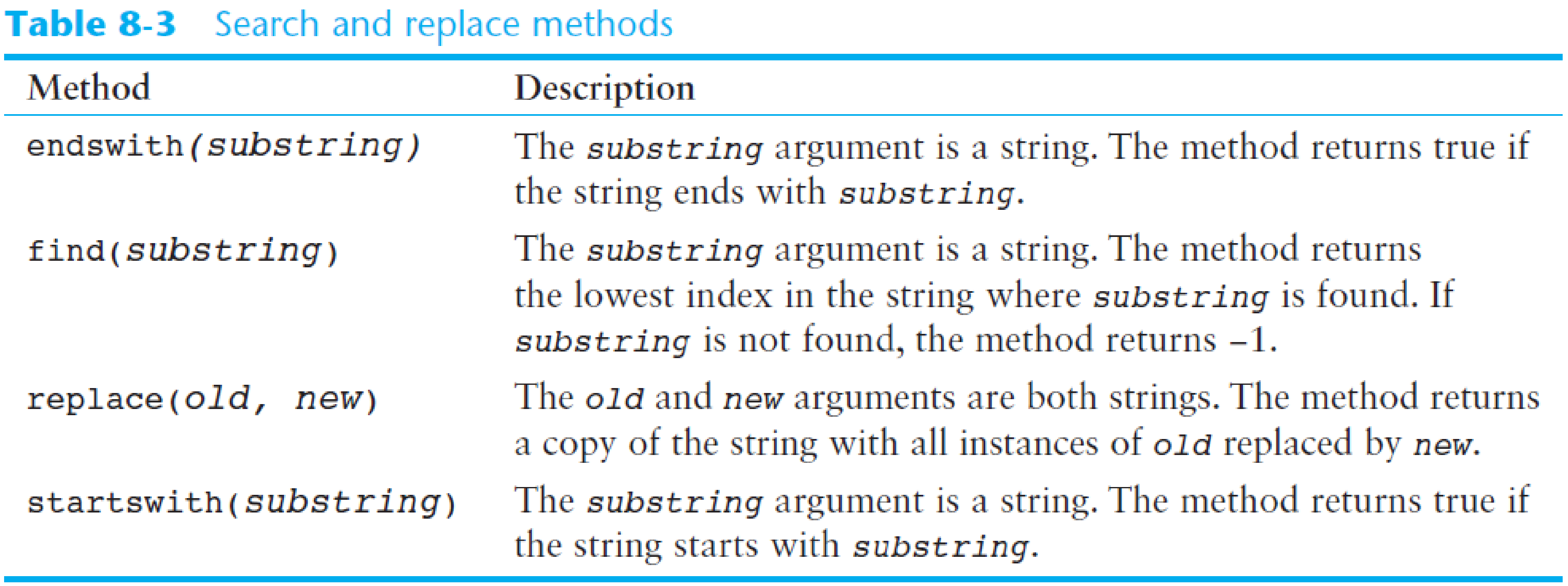


Table 8-3 Search and replace methods

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`*.

The Repetition Operator

- **<u>Repetition operator</u>: makes multiple copies of a string and joins them together**
  - The * symbol is a repetition operator when applied to a string and an integer
    - String is left operand; number is right
  - General format: *`string_to_copy * n`*
  - Variable references a new string which contains multiple copies of the original string

Splitting a String

- **<u>`split` method</u>: returns a list containing the words in the string**
  - By default, uses space as separator
  - Can specify a different separator by passing it as an argument to the `split` method

```
# 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()
```

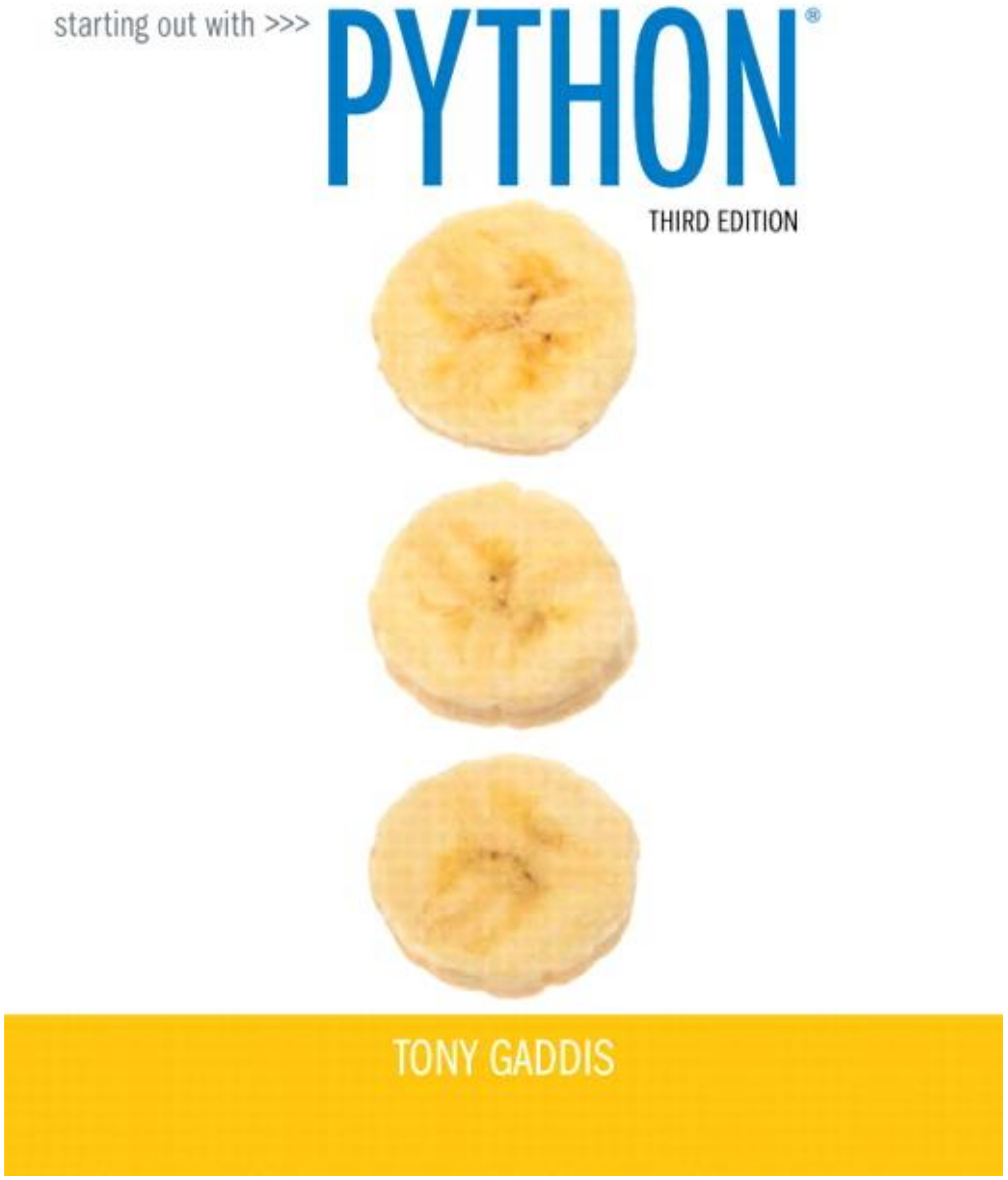
```python
# This program calls the split method, using the
# '/' character as a separator.

def main():
    # Create a string with a date.
    date_string = '11/26/2012'

    # 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()
```

Summary

- **This chapter covered:**
  - String operations, including:
    - Methods for iterating over strings
    - Repetition and concatenation operators
    - Strings as immutable objects
    - Slicing strings and testing strings
    - String methods
    - Splitting a string

CHAPTER 9

Dictionaries and Sets



Topics

- Dictionaries
- Serializing Objects

Dictionaries

- **<u>Dictionary</u>: object that stores a collection of data**
  - Each element consists of a *key* and a *value*
    - Often referred to as *mapping* of key to value
    - Key must be an immutable object
  - To retrieve a specific value, use the key associated with it
  - Format for creating a dictionary

```
dictionary =
        {key1:val1, key2:val2}
```

Retrieving a Value from a Dictionary

- **Elements in dictionary are unsorted**
- **General format for retrieving value from dictionary: `dictionary[key]`**
  - If `key` in the dictionary, associated value is returned, otherwise, `KeyError` exception is raised
- **Test whether a key is in a dictionary using the `in` and `not in` operators**
  - Helps prevent `KeyError` exceptions

Adding Elements to an Existing Dictionary

- **Dictionaries are mutable objects**
- **To add a new key-value pair:**

```
dictionary[key] = value
```

  - If key exists in the dictionary, the value associated with it will be changed

Deleting Elements From an Existing Dictionary and len function

- **To delete a key-value pair:**

  ```
  del dictionary[key]
  ```

  - If key is not in the dictionary, `KeyError` exception is raised

- **`len` function: used to obtain number of elements in a dictionary**

```
>>> phonebook={'chris':'111', 'katie':'222'}
>>> phonebook
{'chris': '111', 'katie': '222'}
>>> phonebook['chirs']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'chirs'
>>> phonebook['chris']
'111'
>>> if 'chris' in phonebook:
...     print(phonebook['chris'])
...
111
>>> phonebook['joe']= '333'
>>> phonebook
{'chris': '111', 'joe': '333', 'katie': '222'}
>>> del phonebook['chris']
>>> phonebook
{'joe': '333', 'katie': '222'}
>>> len(phonebook)
2
```

Getting the Number of Elements and Mixing Data Types

- **Keys must be immutable objects, but associated values can be any type of object**
  - One dictionary can include keys of several different immutable types
- **Values stored in a single dictionary can be of different types**

```
>>> test = {[1,2]:[1,2,3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Creating an Empty Dictionary and Using `for` Loop to Iterate Over a Dictionary

- **To create an empty dictionary:**
  - Use `{}`
  - Use built-in function `dict()`
  - Elements can be added to the dictionary as program executes
- **Use a `for` loop to iterate over a dictionary**
  - General format: `for` *`key`* `in` *`dictionary`*`:`

```
>>> mixed_up = {'abc':1, 999:'yaya', (1,2):[23,33]}
>>> mixed_up
{(1, 2): [23, 33], 'abc': 1, 999: 'yaya'}
>>> phonebook = {}
>>> phonebook
{}
>>> phonebook['chris'] = '999'
>>> phonebook['bill'] = '888'
>>> phonebook
{'chris': '999', 'bill': '888'}
>>> for key in phonebook:
...     print(key, phonebook[key])
...
chris 999
bill 888
```

Some Dictionary Methods

- **`clear` method: deletes all the elements in a dictionary, leaving it empty**
  - Format: `dictionary.clear()`
- **`get` method: gets a value associated with specified key from the dictionary**
  - Format: `dictionary.get(key, default)`
    - `default` is returned if `key` is not found
  - Alternative to `[]` operator
    - Cannot raise `KeyError` exception

Some Dictionary Methods (cont'd.)

- **`items` method: returns all the dictionaries keys and associated values**
  - Format: `dictionary.items()`
  - Returned as a *dictionary view*
    - Each element in dictionary view is a tuple which contains a key and its associated value
    - Use a `for` loop to iterate over the tuples in the sequence
      - Can use a variable which receives a tuple, or can use two variables which receive key and value

Some Dictionary Methods (cont'd.)

- **`keys` method: returns all the dictionaries keys as a sequence**
  - Format: *`dictionary`*`.keys()`
- **`pop` method: returns value associated with specified key and removes that key-value pair from the dictionary**
  - Format: *`dictionary`*`.pop(`*`key, default`*`)`
    - *`default`* is returned if *`key`* is not found

Some Dictionary Methods (cont'd.)

- **`popitem` method: returns a randomly selected key-value pair and removes that key-value pair from the dictionary**
  - Format: *`dictionary`*`.popitem()`
  - Key-value pair returned as a tuple
- **`values` method: returns all the dictionaries values as a sequence**
  - Format: *`dictionary`*`.values()`
  - Use a `for` loop to iterate over the values

Some Dictionary Methods (cont'd.)

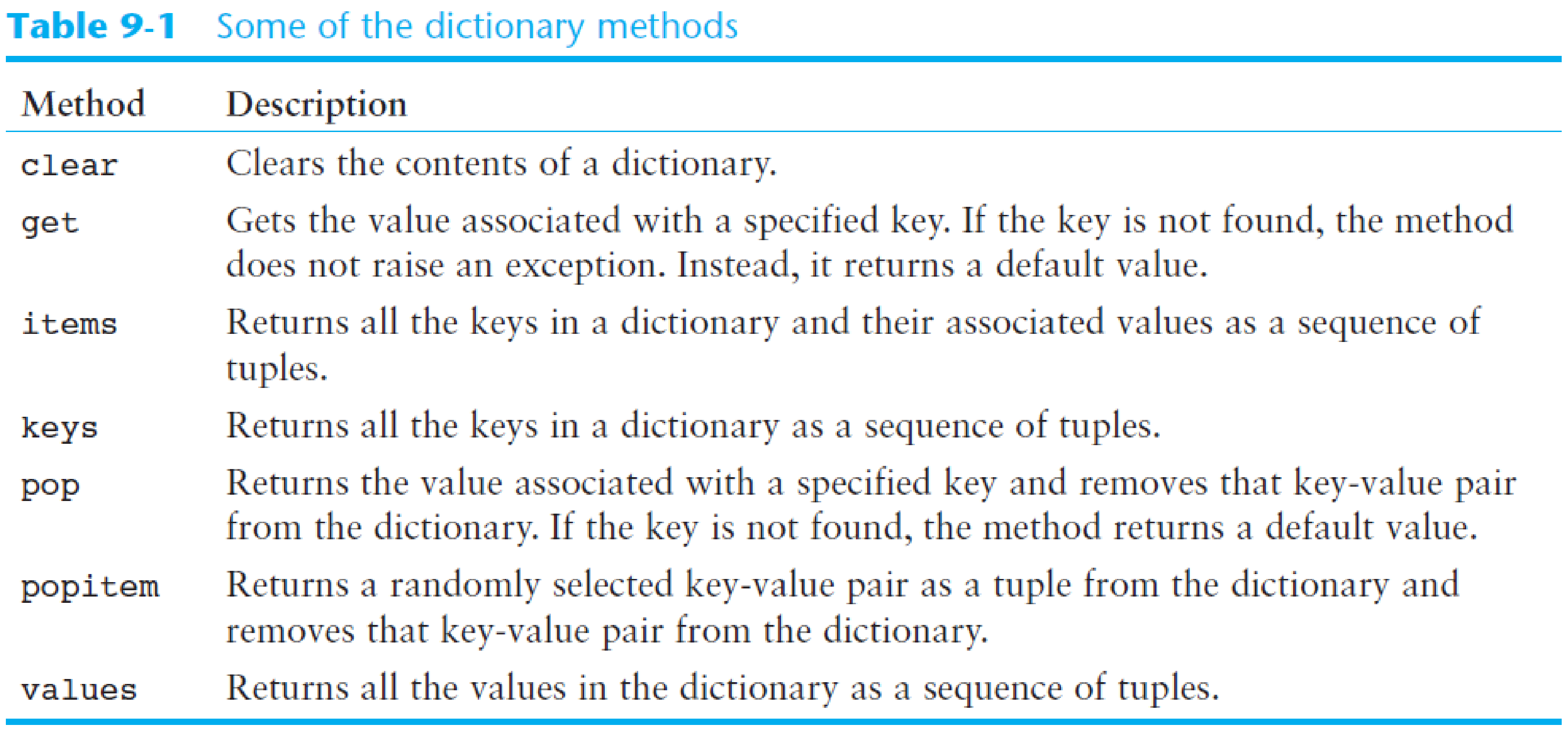


Table 9-1 Some of the dictionary methods

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.

```
>>> value = phonebook.get('bill', 'not found')
>>> print(value)
888
>>> value = phonebook.get('andy', 'not found')
>>> print(value)
not found

>>> phonebook.items()
dict_items([('chris', '999'), ('bill', '888')])

>>> for name, phone in phonebook.items():
...     print(name, phone)
...
chris 999
bill 888

>>> phonebook.keys()
dict_keys(['chris', 'bill'])

>>> phonebook.pop('chris', 'not found')
'999'
>>> phonebook
{'bill': '888'}
```

```
>>> phonebook['andy'] = '777'
>>> phonebook
{'andy': '777', 'bill': '888'}
>>> key, value = phonebook.popitem()
>>> print (key, value)
andy 777
>>> phonebook
{'bill': '888'}

>>> phonebook['andy'] = '777'
>>> phonebook['david'] = '444'
>>> phonebook.values()
dict_values(['777', '444', '888'])
>>> phonebook
{'andy': '777', 'david': '444', 'bill': '888'}

>>> phonebook.clear()
>>> phonebook
{}
```

Serializing Objects

- **Serialize an object: convert the object to a stream of bytes that can easily be stored in a file**
- **Pickling: serializing an object**

Serializing Objects (cont'd.)

- **To pickle an object:**
  - Import the `pickle` module
  - Open a file for binary writing
  - Call the `pickle.dump` function
    - Format: `pickle.dump(`*`object, file`*`)`
  - Close the file
- **You can pickle multiple objects to one file prior to closing the file**

Serializing Objects (cont'd.)

- **<u>Unpickling</u>: retrieving pickled object**
- **To unpickle an object:**
  - Import the `pickle` module
  - Open a file for binary writing
  - Call the `pickle.load` function
    - Format: `pickle.load(`*`file`*`)`
  - Close the file
- **You can unpickle multiple objects from the file**

```
>>> import pickle
>>> phonebook = [{'chris':999}, {'david':888}, {'andy':111}]
>>> with open('phonebook.dat', 'wb’) as output_file:
>>> ....pickle.dump(phonebook, output_file)
```

```
mspan@stu-000000005:~$ ls
java  phonebook.dat  public_html  python
mspan@stu-000000005:~$ more phonebook.dat
▒}q
```

```
>>> import pickle
>>> with open('phonebook.dat', 'rb’) as input_file:
>>> ....pb = pickle.load(input_file)
>>>
>>> pb
[{'chris':999}, {'david':888}, {'andy':111}]
```

Summary

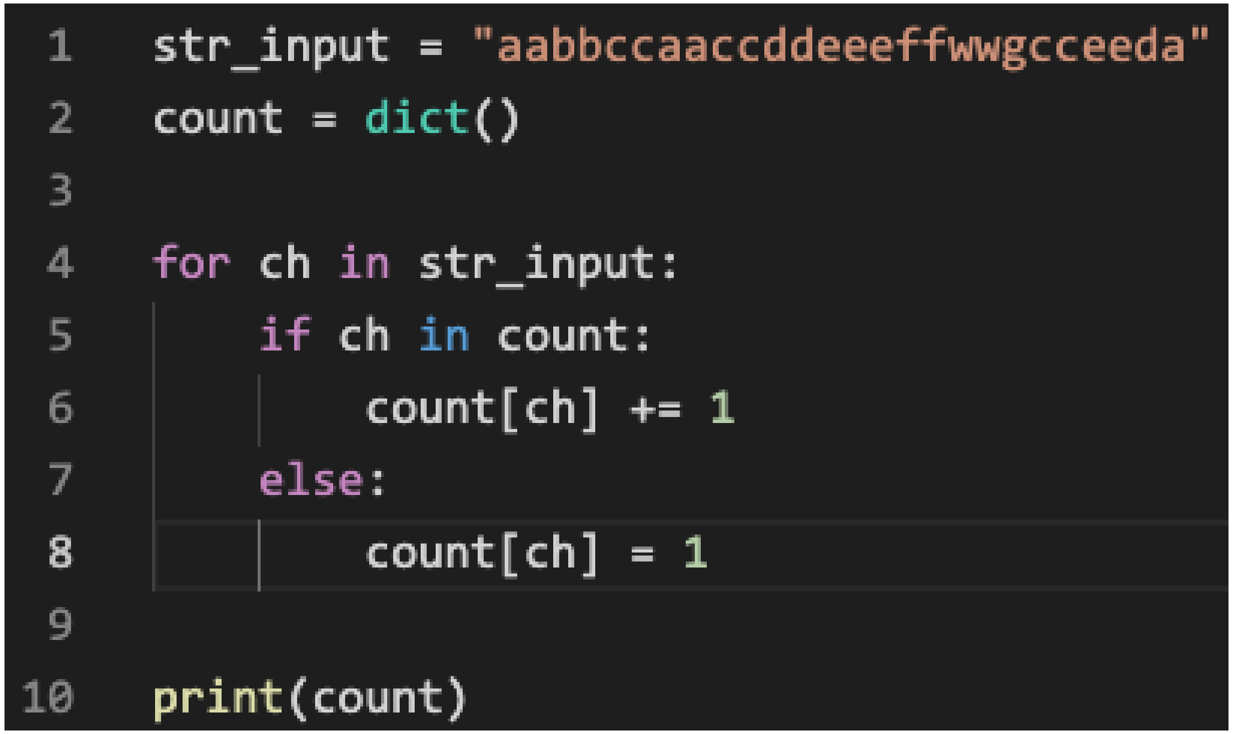
- **This chapter covered:**
  - Dictionaries, including:
    - Creating dictionaries
    - Inserting, retrieving, adding, and deleting key-value pairs
    - `for` loops and `in` and `not in` operators
    - Dictionary methods
  - Serializing objects
    - Pickling and unpickling objects

Notice

- You should take the dictionary as a data structure
- The data structure in python is dynamic
- Use .get to access the dictionary in a more safe way
- Use .items() to traverse all items in a dictionary

Quiz now

- Given any string str_input, please count the number of individual character. For example, if the str_input is “aabbccaaccddeeeffwwgcceeda”, the result will be a: 5, b: 2, …..



```python
str_input = "aabbccaaccddeeeffwwgcceeda"
count = dict()

for ch in str_input:
    if ch in count:
        count[ch] += 1
    else:
        count[ch] = 1

print(count)
```

Assignment

- Maintain student’s data structure by a dictionary
- Plan your data structure by yourself
- 要擋掉相同姓名