Strings

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 (1 of 4)

- 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`*`]`

Accessing the Individual Characters in a String (2 of 4)

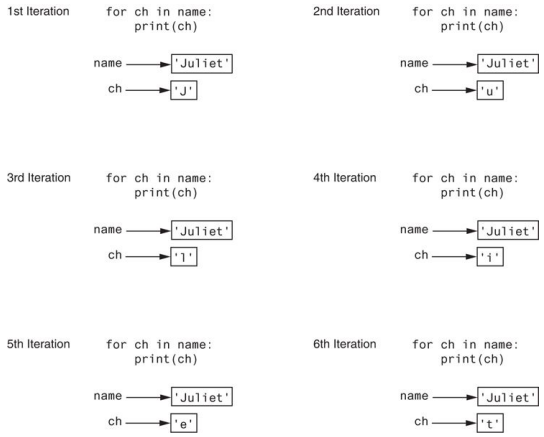

Figure 8-1 Iterating over the string 'Juliet'

Accessing the Individual Characters in a String (3 of 4)

Figure 8-2 String indexes

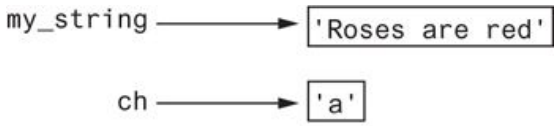

Figure 8-3 Getting a copy of a character from a string

Accessing the Individual Characters in a String (4 of 4)

- `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 (1 of 2)

- 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 (2 of 2)

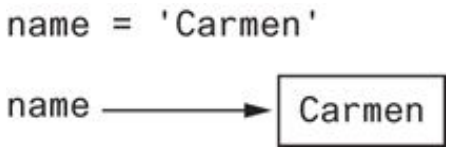

Figure 8-4 The string 'Carmen' assigned to name

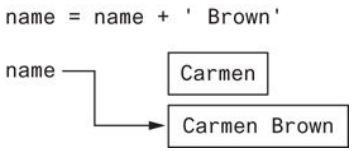

Figure 8-5 The string 'Carmen Brown' assigned to name

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

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

String Methods (1 of 7)

- 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 (2 of 7)

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 Methods (3 of 7)

- 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

String Methods (4 of 7)

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 (5 of 7)

- 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 (6 of 7)

- 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*

String Methods (7 of 7)

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

- Repetition operator: 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 (1 of 2)

- `split` method: 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

Splitting a String (2 of 2)

- Examples:

```
>>> my_string = 'One two three
four'
>>> word_list = my_string.split()
>>> word_list
['One', 'two', 'three', 'four']
>>>
```

```
>>> my_string = '1/2/3/4/5'
>>> number_list = my_string.split('/')
>>> number_list
['1', '2', '3', '4', '5']
>>>
```


String Tokens (1 of 4)

- Sometimes a string contains substrings that are separated by a special character
  - Example:

```
'peach raspberry strawberry vanilla'
```

  - This string contains the substrings *peach*, *raspberry*, *strawberry*, and *vanilla*
  - The substrings are separated by the space character
  - The substrings are known as *tokens* and the separating character is known as the *delimiter*

String Tokens (2 of 4)

- Example:

```
'17;92;81;12;46;5'
```

  - This string contains the tokens 17, 92, 81, 12, 46, and 5
  - The delimiter is the ; character

String Tokens (3 of 4)

- *Tokenizing* is the process of breaking a string into tokens
- When you tokenize a string, you extract the tokens and store them as individual items
- In Python you can use the split method to tokenize a string

String Tokens (4 of 4)

- Examples:

```
>>> str = 'peach raspberry strawberry vanilla'
>>> tokens = str.split()
>>> tokens
['peach', 'raspberry', 'strawberry', 'vanilla']
>>>
```

```
>>> my_address = 'www.example.com'
>>> tokens = my_address.split('.')
>>> tokens
['www', 'example', 'com']
>>>
```


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

Dictionaries and Sets

Topics

- Dictionaries
- Sets
- Serializing Objects

Dictionaries

- Dictionary: 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

- To delete a key-value pair:

```
del dictionary[key]
```

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

Getting the Number of Elements and Mixing Data Types

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

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`*`:`

Some Dictionary Methods (1 of 5)

- `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 (2 of 5)

- `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 (3 of 5)

- `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 (4 of 5)

- `popitem` method: Returns, as a tuple, the key-value pair that was last added to the dictionary. The method also removes the 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 (5 of 5)

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, as a tuple, the key-value pair that was last added to the dictionary. The method also removes the key-value pair from the dictionary.
`values`	Returns all the values in the dictionary as a sequence of tuples.

Dictionary Comprehensions (1 of 6)

- Dictionary comprehension: an expression that reads a sequence of input elements and uses those input elements to produce a dictionary

Dictionary Comprehensions (2 of 6)

- Example: create a dictionary in which the keys are the integers 1 through 4 and the values are the squares of the keys

Using a `for` loop

```
>>> numbers = [1, 2, 3, 4]
>>> squares = {}
>>> for item in numbers:
...     squares[item] = item**2
...
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16}
>>>
```

Using a dictionary comprehension

```
>>> squares = {item:item**2 for item in
numbers}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16}
>>>
```


Dictionary Comprehensions (3 of 6)

```
squares = {item:item**2 for item in numbers}
```

Result Expression

Iteration Expression

- The iteration expression iterates over the elements of numbers
- Each time it iterates, the target variable item is assigned the value of an element
- At the end of each iteration, an element containing item as the key and item**2 as the value is added to the new dictionary

Dictionary Comprehensions (4 of 6)

- Example: You have an existing list of strings. Create a dictionary in which the keys are the stings in the list, and the values are the lengths of the strings

```
>>> names = ['Jeremy', 'Kate', 'Peg']
>>> str_lengths = {item:len(item) for item in names}
>>> str_lengths
{'Jeremy': 6, 'Kate': 4, 'Peg': 3}
>>>
```

Dictionary Comprehensions (5 of 6)

- Example: making a copy of a dictionary

```
>>> dict1 = {'A':1, 'B':2, 'C':3}
>>> dict2 = {k:v for k,v in dict1.items()}
>>> dict2
{'A': 1, 'B': 2, 'C': 3}
>>>
```


Dictionary Comprehensions (6 of 6)

- You can use an `if` clause in a dictionary comprehension to select only certain elements of the input sequence
  - Example: A dictionary contains cities and their populations as key-value pairs. Select only the cities with a population greater than 2 million

```
>>> populations = {'New York': 8398748, 'Los Angeles': 3990456,
...                'Chicago': 2705994, 'Houston': 2325502,
...                'Phoenix': 1660272, 'Philadelphia': 1584138}
>>> largest = {k:v for k,v in populations.items() if v > 2000000}
>>> largest
{'New York': 8398748, 'Los Angeles': 3990456, 'Chicago': 2705994,
'Houston': 2325502}
>>>
```


Sets

- Set: object that stores a collection of data in same way as mathematical set
  - All items must be unique
  - Set is unordered
  - Elements can be of different data types

Creating a Set

- `set` function: used to create a set
  - For empty set, call `set()`
  - For non-empty set, call `set(`*`argument`*`)` where *`argument`* is an object that contains iterable elements
    - e.g., *`argument`* can be a list, string, or tuple
    - If *`argument`* is a string, each character becomes a set element
      - For set of strings, pass them to the function as a list
    - If *`argument`* contains duplicates, only one of the duplicates will appear in the set

Getting the Number of and Adding Elements

- `len` function: returns the number of elements in the set
- Sets are mutable objects
- `add` method: adds an element to a set
- `update` method: adds a group of elements to a set
  - Argument must be a sequence containing iterable elements, and each of the elements is added to the set

Deleting Elements From a Set

- `remove` and `discard` methods: remove the specified item from the set
  - The item that should be removed is passed to both methods as an argument
  - Behave differently when the specified item is not found in the set
    - `remove` method raises a `KeyError` exception
    - `discard` method does not raise an exception
- `clear` method: clears all the elements of the set

Using the `for` Loop, `in`, and `not in` Operators With a Set

- A `for` loop can be used to iterate over elements in a set
  - General format: `for` *`item`* `in` *`set`*`:`
  - The loop iterates once for each element in the set
- The `in` operator can be used to test whether a value exists in a set
  - Similarly, the `not in` operator can be used to test whether a value does not exist in a set

Finding the Union of Sets

- Union of two sets: a set that contains all the elements of both sets
- To find the union of two sets:
  - Use the `union` method
    - Format: *`set1`*`.union(`*`set2`*`)`
  - Use the `|` operator
    - Format: *`set1`* `|` *`set2`*
  - Both techniques return a new set which contains the union of both sets

Finding the Intersection of Sets

- Intersection of two sets: a set that contains only the elements found in both sets
- To find the intersection of two sets:
  - Use the `intersection` method
    - Format: *`set1`*`.intersection(`*`set2`*`)`
  - Use the `&` operator
    - Format: *`set1 & set2`*
  - Both techniques return a new set which contains the intersection of both sets

Finding the Difference of Sets

- Difference of two sets: a set that contains the elements that appear in the first set but do not appear in the second set
- To find the difference of two sets:
  - Use the `difference` method
    - Format: *`set1`*`.difference(`*`set2`*`)`
  - Use the `-` operator
    - Format: *`set1 - set2`*

Finding the Symmetric Difference of Sets

- Symmetric difference of two sets: a set that contains the elements that are not shared by the two sets
- To find the symmetric difference of two sets:
  - Use the `symmetric_difference` method
    - Format: *`set1`*`.symmetric_difference(`*`set2`*`)`
  - Use the `^` operator
    - Format: *`set1 ^ set2`*

Finding Subsets and Supersets (1 of 2)

- Set A is subset of set B if all the elements in set A are included in set B
- To determine whether set A is subset of set B
  - Use the `issubset` method
    - Format: *`setA`*`.issubset(`*`setB`*`)`
  - Use the `<=` operator
    - Format: *`setA <= setB`*

Finding Subsets and Supersets (2 of 2)

- Set A is superset of set B if it contains all the elements of set B
- To determine whether set A is superset of set B
  - Use the `issuperset` method
    - Format: *`setA`*`.issuperset(`*`setB`*`)`
  - Use the >= operator
    - Format: *`setA >= setB`*

Set Comprehensions (1 of 4)

- Set comprehension: a concise expression that creates a new set by iterating over the elements of a sequence
- Set comprehensions are written just like list comprehensions, except that a set comprehension is enclosed in curly braces ({}) instead of brackets ([])

Set Comprehensions (2 of 4)

- Example: making a copy of a set

```
>>> set1 = set([1, 2, 3, 4, 5])
>>> set2 = {item for item in set1}
>>> set2
{1, 2, 3, 4, 5}
>>>
```


Set Comprehensions (3 of 4)

- Example: creating a set that contains the squares of the numbers stored in another set

```
>>> set1 = set([1, 2, 3, 4, 5])
>>> set2 = {item**2 for item in set1}
>>> set2
{1, 4, 9, 16, 25}
>>>
```

Set Comprehensions (4 of 4)

- Example: copying the numbers in a set that are less than 10

```
>>> set1 = set([1, 20, 2, 40, 3, 50])
>>> set2 = {item for item in set1 if item < 10}
>>> set2
{1, 2, 3}
>>>
```


Serializing Objects (1 of 3)

- 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 (2 of 3)

- 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 (3 of 3)

- Unpickling: 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

Summary (1 of 2)

- 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

Summary (2 of 2)

- This chapter covered (cont'd):
  - Sets:
    - Creating sets
    - Adding elements to and removing elements from sets
    - Finding set union, intersection, difference and symmetric difference
    - Finding subsets and supersets
  - Serializing objects
    - Pickling and unpickling objects