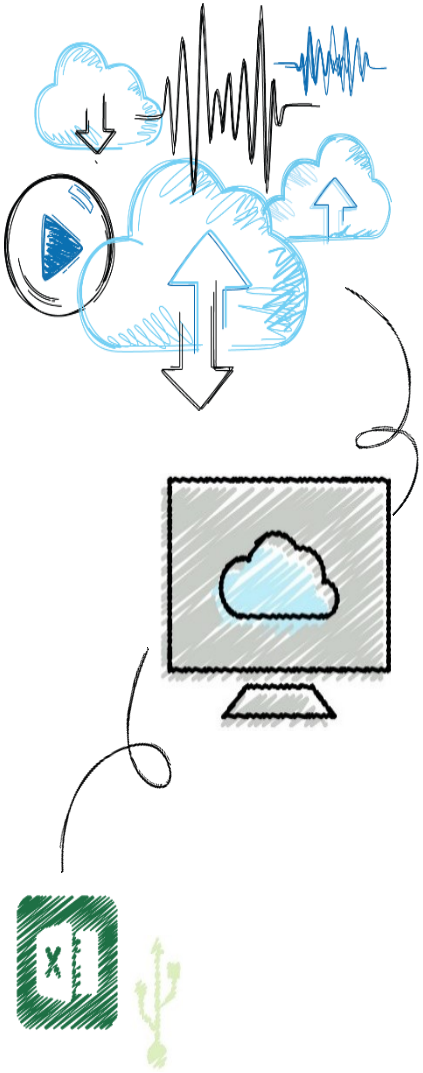
Fundamentals of Python: First Programs
Second Edition

Chapter 5

Lists and Dictionaries

CENGAGE



Objectives (1 of 2)

5.1 Construct lists and access items in those lists

5.2 Use methods to manipulate lists

5.3 Perform traversals of lists to process items in the lists

5.4 Define simple functions that expect parameters and return values

CENGAGE

Objectives (2 of 2)

5.5 Construct dictionaries and access entries in those dictionaries

5.6 Use methods to manipulate dictionaries

5.7 Determine whether a list or a dictionary is an appropriate data structure for a given application

Introduction

- A **list** allows the programmer to manipulate a sequence of data values of any types
- A **dictionary** organizes data values by association with other data values rather than by sequential position
- Lists and dictionaries provide powerful ways to organize data in useful and interesting applications

Lists

- List: Sequence of data values (**items** or **elements**)
- Some examples:
  - Shopping list for the grocery store
  - To-do list
  - Roster for an athletic team
  - Guest list for a wedding
  - Recipe, which is a list of instructions
  - Text document, which is a list of lines
  - Names in a phone book
- Each item in a list has a unique **index** that specifies its position (from 0 to length – 1)

List Literals and Basic Operators (1 of 4)

- Some examples:

```
['apples', 'oranges', 'cherries']
[[5, 9], [541, 78]]
```

- When an element is an expression, its value is included in the list:

```
>>> import math
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
>>> [x + 1]
[3]
```

List Literals and Basic Operators (2 of 4)

- Lists of integers can be built using **range**:

```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
```

- The list function can build a list from any iterable sequence of elements:

```
>>> third = list(“Hi there!”)
>>> third
[‘H’, ‘I’, ‘ ’ , ‘t’, ‘h’, ‘e’, ‘r’, ‘e’, ‘!’]
```

List Literals and Basic Operators (3 of 4)

- **len**, **[]**, **+**, and **==** work on lists as expected:

```
>>> len(first)
4
>>> first[0]
1
>>> first[2:4]
[3, 4]
```

- Concatenation (+) and equality (==) also work as expected for lists:

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
```

List Literals and Basic Operators (4 of 4)

- To print the contents of a list:

```
>>> print(“1234”)
1234
>>> print([1, 2, 3, 4])
[1, 2, 3, 4]
```

- **in** detects the presence of an element:

```
>>> 3 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
```

Replacing an Element in a List (1 of 2)

- A list is **mutable**
  - Elements can be inserted, removed, or replaced
  - The list itself maintains its identity, but its **state**—its length and its contents—can change
- Subscript operator is used to replace an element:

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

- Subscript is used to reference the **target** of the assignment, which is not the list but an element's position within it

Replacing an Element in a List (2 of 2)

- The first session shows how to replace each number in a list with its square:

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> for index in range(len(numbers)):
numbers[index] = numbers[index] ** 2
>>> numbers
[4, 9, 16, 25]
```

- Next session uses the string method **split** to extract a list of words:

```
>>> sentence = “This example has five words. ”
>>> words = sentence.split()
>>> words
[‘This’, ‘example’, ‘has’, ‘five’, ‘words.’]
>>> for index in range(len(words)):
words[index] = words[index].upper()
>>> words
[‘THIS’, ‘EXAMPLE’, ‘HAS’, ‘FIVE’, ‘WORDS.’]
```

CENGAGE

List Methods for Inserting and Removing Elements (1 of 4)

- The **list** type includes several methods for inserting and removing elements

List Method	**What It Does**
L..append(element)	Adds **element** to the end of **L**
L.extend(aList)	Adds the elements of **aList** to the end of **L**
L..insert(index, element)	Inserts **element** at **index** if **index** is less than the length of **L**. Otherwise, inserts **element** at the end of **L**.
L.pop()	Removes and returns the element at the end of **L**.
L.pop(index)	Removes and returns the element at **index**

List Methods for Inserting and Removing Elements (2 of 4)

- The method **insert** expects an integer index and the new element as arguments

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
```

List Methods for Inserting and Removing Elements (3 of 4)

- The method **append** expects just the new element as an argument and adds the new element to the end of the list
- The method extend performs a similar operation, but adds the elements of its list argument to the end of the list

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(3)
>>> example
[1, 2, 3]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 3, 11, 12, 13]
>>> example + [14, 15]
[1, 2, 3, 11, 12, 13, 14, 15]
>>> example
[1, 2, 3, 11, 12, 13]
```

List Methods for Inserting and Removing Elements (4 of 4)

- The method **pop** is used to remove an element at a given position

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop() # Remove the last element
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0) # Remove the first element
1
>>> example
[2, 10, 11, 12]
```

Searching a List

- **in** determines an element’s presence or absence, but does not return position of element (if found)
- Use method **index** to locate an element’s position in a list
  - Raises an error when the target element is not found

```
aList = [34, 45, 67]
target = 45
if target in aList:
        print(aList.index(target))
else:
        print(–l)
```

Sorting a List

- A list’s elements are always ordered by position, but you can impose a **natural ordering** on them
  - For example, in alphabetical order
- When the elements can be related by comparing them <, >, and ==, they can be sorted
  - The method **sort** mutates a list by arranging its elements in ascending order

```
>>>example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

Mutator Methods and the Value None

- All of the functions and methods examined in previous chapters return a value that the caller can then use to complete its work
- **Mutator** methods (e.g., **insert**, **append,extend,pop,and sort**) usually return no value of interest to caller
  - Python automatically returns the special value **None**

```
>>> aList = aList.sort()
>>> print(aList)
None
```

Aliasing and Side Effects (1 of 2)

- Mutable property of lists leads to interesting phenomena:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```

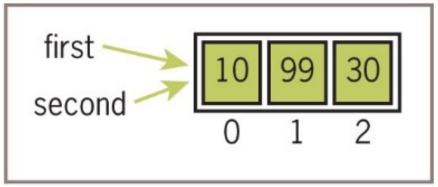


Figure 5-1 Two variables refer to the same list object

- First and second are **aliases**
  - They refer to the exact same list object

Aliasing and Side Effects (2 of 2)

- To prevent aliasing, create a new object and copy contents of original:

```
>>> third = []
>>> for element in first:
third.append(element)
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[l] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```

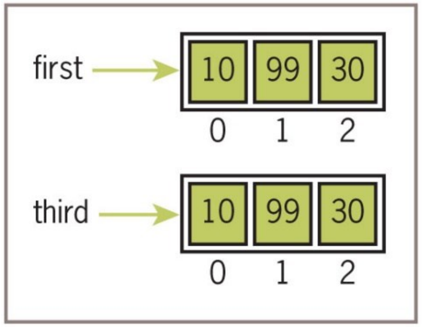


Figure 5-2 Two variables refer to different list objects

Equality: Object Identity and Structural Equivalence (1 of 2)

- Programmers might need to see whether two variables refer to the exact same object or to different objects
- Example, you might want to determine whether one variable is an alias for another
  - The == operator returns True if the variables are aliases for the same object.
  - Unfortunately, == also returns True if the contents of two different objects are the same
- The first relation is called **object identity**
  - The second relation is called **structural equivalence**.
- The == operator has no way of distinguishing between these two types of relations.

CENGAGE

Equality: Object Identity and Structural Equivalence (2 of 2)

- Python's **is** operator can be used to test for object identity

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first) # Or first[:]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```

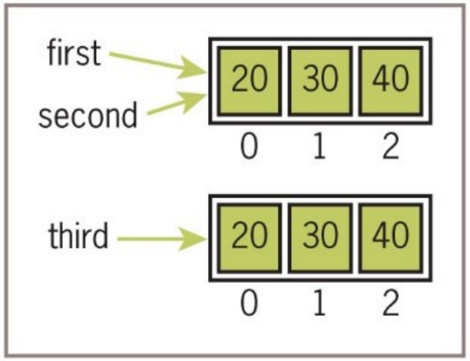


Figure 5-3 Three variables and two distinct list objects

Example: Using a List to Find the Median of a Set of Numbers

- To find the **median** of a set of numbers:

```
“““
File: median.py
Prints the median of a set of numbers in a file.
”””
fileName = input(“Enter the filename: ”)
f = open(fileName, ‘r’)
# Input the text, convert it to numbers, and
# add the numbers to a list
numbers = []
for line in f:
        words = line.split()
        for word in words:
         numbers.append(float(word))
# Sort the list and print the number at its midpoint
numbers.sort()
midpoint = len(numbers) // 2
print(“The median is”, end = “ ”)
if len(numbers) % 2 == 1:
        print(numbers[midpoint])
else:
        print((numbers[midpoint] + numbers[midpoint – 1]) / 2)
```

Tuples

- A **tuple** resembles a list, but is immutable
  - Indicate by enclosing its elements in ()

```
>>> fruits = (“apple”, “banana”)
>>> fruits
(‘apple’, ‘banana’)
>>> meats = (“fish”, “poultry”)
>>> meats
(‘fish’, ‘poultry’)
>>> food = meats + fruits
>>> food
(‘fish’, ‘poultry’, ‘apple’, ‘banana’)
>>> veggies = [“celery”, “beans”]
>>> tuple(veggies)
(‘celery’, ‘beans’)
```

Defining Simple Functions

- Defining our own functions allows us to organize our code in existing scripts more effectively
- This section provides a brief overview of how to do this

The Syntax of Simple Function Definitions

- Definition of a function consists of header and body

```
def square(x):
    “““Returns the square of x.”””
    return x * x
```

- Docstring contains information about what the function does; to display, enter **help(square)**
- A function can be defined in a Python shell, but it is more convenient to define it in an IDLE window
- Syntax of a function definition:

```
def <function name>(<parameter−1>, ..., <parameter−n>):
<body>
```

Parameters and Arguments

- A parameter is the name used in the function definition for an argument that is passed to the function when it is called
- For now, the number and positions of arguments of a function call should match the number and positions of the parameters in the definition
- Some functions expect no arguments
  - They are defined with no parameters

The Return Statement

- Place a **return** statement at each exit point of a function when function should explicitly return a value
- Syntax:

return <expression>

- If a function contains no **return** statement, Python transfers control to the caller after the last statement in the function's body is executed
  - The special value **None** is automatically returned

Boolean Functions

- A **Boolean function** usually tests its argument for the presence or absence of some property
  - Returns **True** if property is present; **False** otherwise
- Example:

```
>>> odd(5)
True
>>> odd(6)
False
def odd(x):
“““ Returns True if x is odd or False otherwise.”””
if x % 2 == 1:
return True
else:
return False
```

Defining a Main Function (1 of 2)

- **main** serves as the entry point for a script
  - Usually expects no arguments and returns no value
- Definition of **main** and other functions can appear in no particular order in the script
  - As long as **main** is called at the end of the script
- Script can be run from IDLE, imported into the shell, or run from a terminal command prompt

Defining a Main Function (2 of 2)

- Example:

```
“““
File: computesquare.py
Illustrates the definition of a main function.
”””
def main():
“““ The main function for this script.”””
number = float(input(“Enter a number: ”))
result = square(number)
print(“The square of”, number, “is”, result)
def square(x):
““““Returns the square of x.”””
return x * x
# The entry point for program execution
if __name__ == “__main:”__
main()
```

Dictionaries

- A dictionary organizes information by **association**, not position
  - Example: When you use a dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn to the words beginning with "M"
- Data structures organized by association are also called **tables** or **association lists**
- In Python, a **dictionary** associates a set of **keys** with data values

Dictionary Literals

- A Python dictionary is written as a sequence of key/value pairs separated by commas
  - Pairs are sometimes called **entries**
  - Enclosed in curly braces (**{** and **}**)
  - A colon (**:**) separates a key and its value
- Examples:

A phone book: **{‘Savannah’:‘476-3321’, ‘Nathaniel’:‘351-7743’}**

Personal information: **{‘Name’:‘Molly’, ‘Age’:18}**

An empty dictionary: **{}**

- Keys in a dictionary can be data of any immutable types, including other data structures
  - They are normally strings or integers

Adding Keys and Replacing Values

- Add a new key/value pair to a dictionary using **[]**:

  <a dictionary>[<a key>] = <a value>

- Example:

```
>>> info = {}
>>> info[“name”] = “Sandy”
>>> info[“occupation”] = “hacker”
>>> info
{‘name’:‘Sandy’, ‘occupation’:‘hacker’}
```

- Use **[]** also to replace a value at an existing key:

```
>>> info[“occupation”] = “manager”
>>> info
{‘name’:‘Sandy’, ‘occupation’: ‘manager’}
```

Accessing Values (1 of 2)

- Use **[]** to obtain the value associated with a key
- If key is not present in dictionary, an error is raised

```
>>> info[“name”]
‘Sandy’
>>> info[“job”]
Traceback (most recent call last):
  File “<pyshell#1>”, line 1, in <module>
          info[“job”]
KeyError: ‘job’
```

Accessing Values (2 of 2)

- If the existence of a key is uncertain, test for it using the dictionary method **has_key**
  - Easier strategy is to use the method **get**

```
>>> if “job” in info:
        print(info.[“job”])
```

Removing Keys

- To delete an entry from a dictionary, remove its key using the method **pop**
    - **pop** expects a key and an optional default value as arguments

```
>>> print(info.pop(“job”, None))
None
>>> print(info.pop(“occupation”))
manager
>>> info
{‘name’:‘Sandy’}
```

Traversing a Dictionary (1 of 3)

- To print all of the keys and their values:

```
for key in info:
    print(key, info[key])
```

- Alternative: Use the dictionary method **items()**

```
>>> grades = {90:‘A’, 80:‘B’, 70:‘C’}
>>> list(grades.items())
[(80, ‘B’), (90,‘A’), (70,‘C’)]
```

- Entries are represented as tuples within the list

```
for (key, value) in grades.items():
print(key, value)
```

Traversing a Dictionary (2 of 3)

- You can sort the list first then traverse it to print the entries of the dictionary in alphabetical order:

```
theKeys = list(info.keys())
theKeys.sort()
for key in theKeys:
    print(key, info[key])
```

Traversing a Dictionary (3 of 3)

Dictionary Operation	What It Does
len(d)	Returns the number of entries in **d**
d[key]	Used for inserting a new key, replacing a value, or obtaining a value at an existing key
d.get(key [, default])	Returns the value if the key exists or returns the default if the key does not exist
d.pop(key [, default])	Removes the key and returns the value if the key exists or returns the default if the key does not exist
list(d.keys())	Returns a list of the keys
list(d.values())	Returns a list of the values
list(d.items())	Returns a list of tuples containing the keys and values for each entry
d.clear()	Removes all the keys
for key in d:	**key** is bound to each key in d in an unspecified order

Example: The Hexadecimal System Revisited (1 of 2)

- You can keep a hex-to-binary **lookup table** to aid in the conversion process

```
hexToBinaryTable = {‘0’:‘0000’, ‘1’:‘0001’, ‘2’:‘0010’,
‘3’:‘0011’, ‘4’:‘0100’, ‘5’:‘0101’,
‘6’:‘0110’, ‘7’:‘0111’, ‘8’:‘1000’,
‘9’:‘1001’, ‘A’:‘1010’, ‘B’:‘1011’,
‘C’:‘1100’, ‘D’:‘1101’, ‘E’:‘1110’,
‘F’:‘1111’}
```

Example: The Hexadecimal System Revisited (2 of 2)

- The function convert expects two parameters: a string representing the number to be converted and a table of associations of digits
- Example:

```
def convert(number, table):
        """"Builds and returns the base two representation of
        number.""""
        binary = " "
        for digit in number:
            binary = table[digit] + binary
        return binary

>>> convert("35A", hexToBinaryTable)
'001101011010'
```

Example: Finding the Mode of a List of Values (1 of 2)

- The **mode** of a list of values is the value that occurs most frequently
- The following script inputs a list of words from a text file and prints their mode

```
fileName = input(“Enter the filename:”)
f = open(fileName, ‘r’)

# Input the text, convert its words to uppercase, and
# add the words to a list
words = []
for line in f:
          for word in line.split():
          words.append(word.upper())
```

Example: Finding the Mode of a List of Values (2 of 2)

```
# Obtain the set of unique words and their
# frequencies, saving these associations in
# a dictionary
theDictionary = {}
for word in words:
        number = theDictionary.get(word, None)
        if number == None:
          # word entered for the first time
          theDictionary[word] = 1
else:
        # word already seen, increment its number
        theDictionary[word] = number + 1

# Find the mode by obtaining the maximum value
# in the dictionary and determining its key
theMaximum = max(theDictionary.values())
for key in theDictionary:
        if theDictionary[key] == theMaximum:
          print(“The mode is”, key)
          break
```

Chapter Summary (1 of 2)

- A list is a sequence of zero or more elements
  - Can be manipulated with the subscript, concatenation, comparison, and **in** operators
  - Mutable data structure
  - **index** returns position of target element in a list
  - Elements can be arranged in order using **sort**
- Mutator methods are called to change the state of an object; usually return the value **None**
- Assignment of a variable to another one causes both to refer to the same data object (aliasing)

Chapter Summary (2 of 2)

- A tuple is similar to a list, but is immutable
- A function definition consists of header and body
  - **return** returns a value from a function definition
- The number and positions of arguments in a function call must match the number and positions of required parameters specified in the function's definition
- A dictionary associates a set of keys with values
  - **[]** is used to add a new key/value pair to a dictionary or to replace a value associated with an existing key
  - **dict** type includes methods to access and remove data in a dictionary
- Testing can be bottom-up, top-down, or you can use a mix of both