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
  - Guest list for a wedding
  - Recipe, which is a list of instructions
  - Text document, which is a list of lines
  - Words in a dictionary
- Each item in a list has a unique **index** that specifies its position (from 0 to length – 1)

List Literals and Basic Operators

- Some examples:

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

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

```
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
```

- 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]
>>>
```

List Literals and Basic Operators (continued)

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

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

- 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:

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


List Literals & Basic Operators (cont.)

OPERATOR OR FUNCTION	WHAT IT DOES
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L + L`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range `0` through *`upper`* `- 1`.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns `True` if all the results are true, or `False` otherwise.
`for <variable> in L:` `  <statement>`	Iterates through the list, binding the variable to each element.
`<any value> in L`	Returns `True` if the value is in the list or `False` otherwise.

[TABLE 5.1] Some operators and functions used with lists

Replacing an Element in a List

- 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 (continued)

- Examples:

```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> index = 0
>>> while index < len(words):
        words[index] = words[index].upper()
        index += 1

>>> words
['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
```

```
>>> numbers = range(6)
>>> numbers
[0, 1, 2, 3, 4, 5]
>>> numbers[0:3] = [11, 12, 13]
>>> numbers
[11, 12, 13, 3, 4, 5]
```

List Methods for Inserting and Removing Elements

- 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 `L` to the end of `aList`.
`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`.

[TABLE 5.2] List methods for inserting and removing elements

List Methods for Inserting and Removing Elements (continued)

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

```
>>> example = [1, 2]
>>> example.append(10)
>>> example
[1, 2, 10]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)
1
```

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(-1)
```


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`**) usually return no value of interest to caller
  - Python automatically returns the special value **`None`**

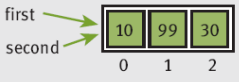
```
>>> aList = aList.sort()
>>> print(aList)
None
```


Aliasing and Side Effects

- 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]
```

`first` and **`second`** are **aliases** (refer to the exact same list object)



[FIGURE 5.1] Two variables refer to the same list object

Aliasing and Side Effects (continued)

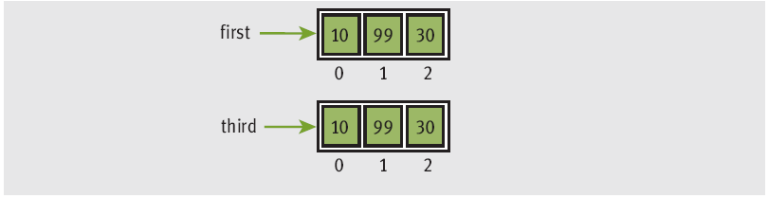
- To prevent aliasing, copy contents of object:

```
>>> third = []
>>> for element in first:
        third.append(element)

>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
```

Alternative:

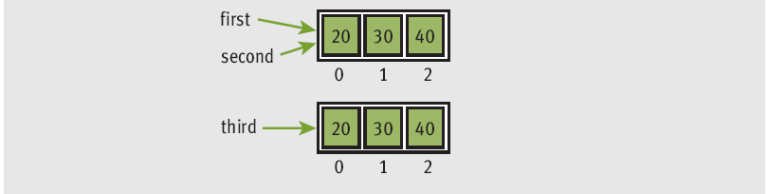
```
>>> third = first[:]
```



[FIGURE 5.2] Two variables refer to different list objects

Equality: Object Identity and Structural Equivalence

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = [20, 30, 40]
>>> 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:

```
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')
```

- Most of the operators and functions used with lists can be used in a similar fashion with tuples

Defining Simple Functions

- Defining our own functions allows us to organize our code in existing scripts more effectively

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
```

```
>>> square(2)
4
```

  - 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

- **`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 (continued)

```
"""
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
main()
```


Case Study: Generating Sentences

- Request: write a program that generates sentences
- Analysis: program will generate sentences from a simplified subset of English

PHRASE	ITS CONSTITUENTS
Sentence	Noun phrase + Verb phrase
Noun phrase	Article + Noun
Verb phrase	Verb + Noun phrase + Prepositional phrase
Prepositional phrase	Preposition + Noun phrase

[TABLE 5.3] The grammar rules for the sentence generator

```
> python generator.py
Enter the number of sentences: 3
THE BOY HIT THE BAT WITH A BOY
THE BOY HIT THE BALL BY A BAT
THE BOY SAW THE GIRL WITH THE GIRL
```

Case Study: Generating Sentences (continued)

- Design:
  - Assign task of generating each phrase to a separate function

```
def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase() + "."

def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)

def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
    for count in range(number):
        print(sentence())
```


Case Study: Generating Sentences (continued)

- Implementation (coding):
  - The variables for the data are initialized just below the **`import`** statement

```
"""
Program: generator.py
Author: Ken
Generates and displays sentences using simple grammar
and vocabulary.  Words are chosen at random.
"""

import random

articles = ("A", "THE")

nouns = ("BOY", "GIRL", "BAT", "BALL",)

verbs = ("HIT", "SAW", "LIKED")

prepositions = ("WITH", "BY")
```

Case Study: Generating Sentences (continued)

```
def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase()

def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)

def verbPhrase():
    """Builds and returns a verb phrase."""
    return random.choice(verbs) + " " + nounPhrase() + " " + \
           prepositionalPhrase()

def prepositionalPhrase():
    """Builds and returns a prepositional phrase."""
    return random.choice(prepositions) + " " + nounPhrase()

def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
    for count in range(number):
        print(sentence())

main()
```


Case Study: Generating Sentences (continued)

- Testing:
  - Two approaches:
    - Bottom-up
    - Top-down
  - Wise programmer can mix bottom-up and top-down testing as needed

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:

```
{'Sarah':'476-3321', 'Nathan':'351-7743'}     A Phone book
{'Name':'Molly', 'Age':18}                    Personal information
{}                                            An empty dictionary
```

- Keys can be data of any immutable types, including other data structures

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

- 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 "<stdin>", line 1, in <module>
KeyError: 'job'
>>>
```

- 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`**

```
>>> print(info.get("job", None))
None
>>>
```

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

- 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"}
>>> 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)
```

- You can sort the list first:

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

Traversing a Dictionary (continued)

DICTIONARY OPERATION	WHAT IT DOES
`len(d)`	Returns the number of entries in `d`.
`aDict[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. Raises an error if the default is omitted and 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. Raises an error if the default is omitted and 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.has_key(key)`	Returns `True` if the key exists or `False` otherwise.
`d.clear()`	Removes all the keys.
`for key in d:`	`key` is bound to each key in `d` in an unspecified order.

[TABLE 5.4] Some commonly used dictionary operations

Example: The Hexadecimal System Revisited

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

>>> convert("35A", hexToBinaryTable)
'001101011111'
```

Example: Finding the Mode of a List of Values

- 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:
    wordsInLine = line.split()
    for word in wordsInLine:
        words.append(word.upper())
```


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

```
# 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
```

Case Study: Nondirective Psychotherapy (Request)

- Doctor in this kind of therapy responds to patient's statements by rephrasing them or indirectly asking for more information
- Request:
  - Write a program that emulates a nondirective psychotherapist

Case Study: Nondirective Psychotherapy (Analysis)

```
Good morning, I hope you are well today.
What can I do for you?

>> My mother and I don't get along
Why do you say that your mother and you don't get along

>> she always favors my sister
You seem to think that she always favors your sister

>> my dad and I get along fine
Can you explain why your dad and you get along fine

>> he helps me with my homework
Please tell me more

>> quit
Have a nice day!
```

[FIGURE 5.4] A session with the doctor program

Case Study: Nondirective Psychotherapy (Analysis) (continued)

- When user enters a statement, program responds in one of two ways:
  - With a randomly chosen hedge, such as “Please tell me more”
  - By changing some key words in user’s input string and appending string to a randomly chosen qualifier
    - Thus, to “My teacher always plays favorites,” program might reply, “Why do you say that your teacher always plays favorites?”

Case Study: Nondirective Psychotherapy (Design)

- Program consists of a set of collaborating functions that share a common data pool
- Pseudocode:

```
output a greeting to the patient
while True
    prompt for and input a string from the patient
    if the string equals “Quit”
        output a sign-off message to the patient
        break
    call another function to obtain a reply to this string
    output the reply to the patient
```

Case Study: Nondirective Psychotherapy (Implementation)

```
"""
Program: doctor.py
Author: Ken
Conducts an interactive session of nondirective psychotherapy.
"""
import random

hedges = ("Please tell me more.",
          "Many of my patients tell me the same thing.",
          "Please continue.")

qualifiers = ("Why do you say that ",
              "You seem to think that ",
              "Can you explain why ")

replacements = {"I":"you", "me":"you", "my":"your",
                "we":"you", "us":"you", "mine":"yours"}
```


Case Study: (Implementation, cont.)

```
def reply(sentence):
    """Builds and returns a reply to the sentence."""
    probability = random.randint(1, 4)
    if probability == 1:
        return random.choice(hedges)
    else:
        return random.choice(qualifiers) + changePerson(sentence)

def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)

def main():
    """Handles the interaction between patient and doctor."""
    print("Good morning, I hope you are well today.")
    print("What can I do for you?")
    while True:
        sentence = input("\n>> ")
        if sentence.upper() == "QUIT":
            print("Have a nice day!")
            break
        print(reply(sentence))

main()
```

Case Study: Nondirective Psychotherapy (Testing)

- Functions in this program can be tested in a bottom-up or a top-down manner
- Program's replies break down when:
  - User addresses the therapist in the second person
  - User uses contractions (for example, I'm and I'll)
- With a little work, you can make the replies more realistic

Summary

- 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)

Summary (continued)

- 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
- 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