UNIT-III

List and Dictionaries

Topic 1: Lists

A list is a sequence of data values called items or elements. An item can be of any type.

Here are some real-world examples of lists:

• A shopping list for the grocery store

• A to-do list

• A roster for an athletic team

• A guest list for a wedding

• A recipe, which is a list of instructions

• A text document, which is a list of lines

• The names in a phone book

The logical structure of a list resembles the structure of a string. Each of the items in a list is ordered by position.

List Rules:

- insertion order preserved.
- duplicate objects are allowed
- heterogeneous objects are allowed.
- List is dynamic because based on our requirement we can increase the size and decrease the size.
- In List the elements will be placed within square brackets and with comma separator.

We can differentiate duplicate elements by using index and we can preserve Insertion order by using index. Hence index will play very important role. Python supports both positive and negative indexes. +ve index means from left to right where as negative index means right to left.

- List objects are mutable. i.e we can change the content

A list can be define as below

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]

1. Creation of List Objects:

1. We can create empty list object as follows...

```
1)  list=[]
2)  print(list)
3)  print(type(list))
4)
5)  []
6)  <class 'list'>
```

2.If we know elements already then we can create list as follows

```
list=[10,20,30,40]
```

3. With dynamic input:

```
1)  list=eval(input("Enter List:"))
2)  print(list)
3)  print(type(list))
4)
5)   D:\Python_classes>py test.py
6)  Enter List:[10,20,30,40]
7)  [10, 20, 30, 40]
8)  <class 'list'>
```

4.List vs mutability:

Once we creates a List object, we can modify its content. Hence List objects are Mutable.

```
1)  n=[10,20,30,40]
2)  print(n)
3)  n[1]=777
4)  print(n)
5)
6)  D:\Python_classes>py test.py
7)  [10, 20, 30, 40]
8)  [10, 777, 30, 40]
```

2. **Accessing elements of List:**

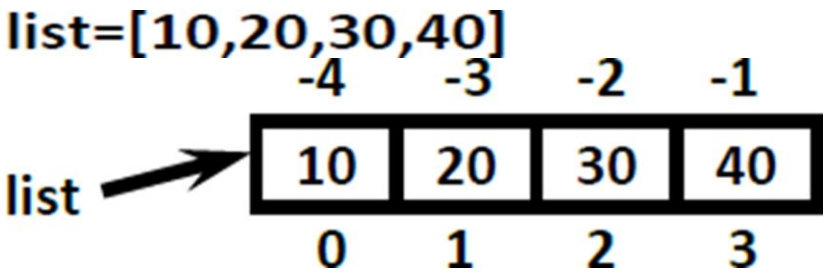
We can access elements of the list either by using index or by using slice operator(:)

1. By using index:

List follows zero based index. ie index of first element is zero. List supports both +ve and -ve indexes.

+ve index meant for **Left to Right**

-ve index meant for **Right to Left**



print(list[0]) ==>10

print(list[-1]) ==>40

print(list[10]) ==>IndexError: list index out of range

2. Accessing elements of List:

We can access elements of the list either by using index or by using slice operator(:)

2. By using slice operator:

Syntax:

list2= list1[start:stop:step]

start ==>it indicates the index where slice has to start default value is 0

stop ===>It indicates the index where slice has to end

default value is max allowed index of list ie length of the list step ==>increment value

default value is 1.

```
1)  n=[1,2,3,4,5,6,7,8,9,10]
2)  print(n[2:7:2])
3)  print(n[4::2])
4)  print(n[3:7])
5)  print(n[8:2:-2])
6)  print(n[4:100])
```

```
8)   Output
9)   D:\Python_classes>py test.py
10) [3, 5, 7]
11) [5, 7, 9]
12) [4, 5, 6, 7]
13) [9, 7, 5]
14)  [5, 6, 7, 8, 9, 10]
```

Important functions of List:

1. **To get information about list** len(), count(), index()
2. **Manipulating elements of List**
   append(), insert(), extend(), remove(), pop()
3. **Ordering elements of List** reverse(), sort()

1.len():

returns the number of elements present in the list

Eg: n=[10,20,30,40]

print(len(n))

= = > 4

2. **count():**

It returns the number of occurrences of specified item in the list

```
1)  n=[1,2,2,2,2,3,3]
2)  print(n.count(1))
3)  print(n.count(2))
4)  print(n.count(3))
5)  print(n.count(4))
```

```
7)  Output
8)  D:\Python_classes>py test.py
9)  1
10) 4
11) 2
12) 0
```

3.index():

returns the index of first occurrence of the specified item.

```
1)  n=[1,2,2,2,2,3,3]
2)  print(n.index(1))    ==>0
3)  print(n.index(2))    ==>1
4)  print(n.index(3))    ==>5
5)  print(n.index(4))    ==>ValueError: 4 is not in list
```

4.append():

We can use append() function to add item at the end of the list.

```
1)  list=[]
2)  list.append("A")
3)  list.append("B")
4)  list.append("C")
5)  print(list)
6)
7)  D:\Python_classes>py test.py
8)  ['A', 'B', 'C']
```

```
1)  list=[]
2)  for i in range(101):
3)      if i%10==0:
4)          list.append(i)
5)  print(list)
6)
7)
8)  D:\Python_classes>py test.py
9)  [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

5.insert():

To insert item at specified index position .

```
1)  n=[1,2,3,4,5]
2)  n.insert(1,888)
3)  print(n)
4)
5)  D:\Python_classes>py test.py
6)  [1, 888, 2, 3, 4, 5]
```

```
1)  n=[1,2,3,4,5]
2)  n.insert(10,777)
3)  n.insert(-10,999)
4)  print(n)
6)  D:\Python_classes>py test.py
7)  [999, 1, 2, 3, 4, 5, 777]
```

Differences between append() and insert()

append()	insert()
In List when we add any element it will come in last i.e. it will be last element.	In List we can insert any element in particular index number

6.extend():

To add all items of one list to another list

l1.extend(l2) - all items present in l2 will be added to l1

```
1) order1=["Chicken","Mutton","Fish"]
2) order2=["RC","KF","FO"]
3) order1.extend(order2)
4) print(order1)
5)
6) D:\Python_classes>py test.py
7) ['Chicken', 'Mutton', 'Fish', 'RC', 'KF', 'FO']
```

```
1) order=["Chicken","Mutton","Fish"]
2) order.extend("Mushroom")
3) print(order)
4)
5) D:\Python_classes>py test.py
6) ['Chicken', 'Mutton', 'Fish', 'M', 'u', 's', 'h', 'r', 'o', 'o', 'm']
```

7.remove():

We can use this function to remove specified item from the list. If the item present multiple times then only first occurrence will be removed.

```
1) n=[10,20,10,30]
2) n.remove(10)
3) print(n)
4)
5) D:\Python_classes>py test.py
6) [20, 10, 30]
```

8. **pop():**

- It removes and returns the last element of the list.
- This is only function which manipulates list and returns some element.

```
1) n=[10,20,30,40]
2) print(n.pop())
3) print(n.pop())
4) print(n)
```

```
6) D:\Python_classes>py test.py
7) 40
8) 30
9) [10, 20]
```

Differences between remove() and pop()

remove()	pop()
1) We can use to remove special element from the List.	1) We can use to remove last element from the List.
2) It can't return any value.	2) It returned removed element.
3) If special element not available then we get VALUE ERROR.	3) If List is empty then we get Error.

9.reverse():

We can use to reverse() order of elements of list.

```
1) n=[10,20,30,40]
2) n.reverse()
3) print(n)
4)
5) D:\Python_classes>py test.py
6) [40, 30, 20, 10]
```

10.sort():

In list by default insertion order is preserved. If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

- For numbers ==>default natural sorting order is Ascending Order
- For Strings ==> default natural sorting order is Alphabetical Order

```
1) n=[20,5,15,10,0]
2) n.sort()
3) print(n)      #[0,5,10,15,20]
4)
5) s=["Dog","Banana","Cat","Apple"]
6) s.sort()
7) print(s)  #['Apple','Banana','Cat','Dog']
```

11.clear():

We can use clear() function to remove all elements of List.

```
1. n=[10,20,30,40]
2. print(n)
3. n.clear()
4. print(n)
5.
6. Output
7. D:\Python_classes>py test.py
8. [10, 20, 30, 40]
9. []
```

Topic 2: Defining Simple Functions of List

1.The Syntax of Simple Function Definitions:

Most of the functions used thus far expect one or more arguments and return a value. Let's define a function that expects a number as an argument and returns the square of that number. First, we consider how the function will be used. Its name is square, so you can call it like this:

>>> square(2)

4

```
>>> square(6)
36
>>> square(2.5)
6.25
```

The definition of this function consists of a header and a body. Here is the code:

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

2.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. Some functions expect no arguments, so they are defined with no parameters.

3.The return Statement

The programmer places a return statement at each exit point of a function when that function should explicitly return a value. The syntax of the return statement for these cases is the following:

return <expression>

4.Boolean Functions

A Boolean function usually tests its argument for the presence or absence of some property. The function returns True if the property is present, or False otherwise. The next example shows the use and definition of the Boolean function odd, which tests a number to see whether it is odd.

```
>>> 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
```

Note that this function has two possible exit points, in either of the alternatives within the if/else statement.

5.Defining a main Function

In scripts that include the definitions of several cooperating functions, it is often useful to define a special function named main that serves as the entry point for the script. This func

```
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()
```

Topic:3 Dictionaries

We can use List, Tuple and Set to represent a group of individual objects as a single entity.

• If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

Eg:

rollno----name

phone number--address

ipaddress---domain name

Duplicate keys are not allowed but values can be duplicated.

• Heterogeneous objects are allowed for both key and values.

• insertion order is not preserved

• Dictionaries are mutable

• Dictionaries are dynamic

• indexing and slicing concepts are not applicable

Note: In C++ and Java Dictionaries are known as "Map" where as in Perl and

Ruby it is known as "Hash"

How to create Dictionary?

d={} or d=dict()

we are creating empty dictionary. We can add entries as follows

d[100]="srinu"

d[200]="ravi"

d[300]="shiva"

print(d) #{100: 'srinu', 200: 'ravi', 300: 'shiva'}

How to access data from the dictionary?

We can access data by using keys.

```
d={100:'srinu' ,200:'ravi', 300:'shiva'}
print(d[100]) #srinu
print(d[300]) #shiva
```

If the specified key is not available then we will get KeyError

```
print(d[400]) # KeyError: 400
```

How to update dictionaries?

d[key]=value

If the key is not available then a new entry will be added to the dictionary with the specified key-value pair

If the key is already available then old value will be replaced with new value.

Adding Keys and Replacing Values

You add a new key/value pair to a dictionary by using the subscript operator []. The form of this operation is the following:

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

The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
```

{'name':'Sandy', 'occupation':'hacker'}

The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

Removing Keys

To delete an entry from a dictionary, one removes its key using the method pop. This method expects a key and an optional default value as arguments. If the key is in the dictionary, it is removed, and its associated value is returned. Otherwise, the default value is returned.

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name':'Sandy'}
```

Traversing a Dictionary

When a for loop is used with a dictionary, the loop's variable is bound to each key in an unspecified order. The next code segment prints all of the keys and their values in our info dictionary:

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

Alternatively, you could use the dictionary method items() to access the dictionary's entries. The next session shows a run of this method with a dictionary of grades:

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

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. 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.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

Design with Function

Design is important in many fields. The architect who designs a building, the engineer who designs a bridge or a new automobile, and the politician, advertising executive, or army general who designs the next campaign must organize the structure of a system and coordinate the actors within it to achieve its purpose. Design is equally important in constructing software systems, some of which are the most complex artifacts ever built by human beings. In this chapter, we explore the use of functions to design software systems.

Topic 4: Functions as Abstraction Mechanisms

The problem is that the human brain can wrap itself around just a few things at once (psychologists say three things comfortably, and at most seven). People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an abstraction. Put most plainly, an abstraction hides detail and thus allows a person to view many things as just one thing.

The problem is that the human brain can wrap itself around just a few things at once (psychologists say three things comfortably, and at most seven). People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an abstraction. Put most plainly, an abstraction hides detail and thus allows a person to view many things as just one thing. We use abstractions to refer to the most common tasks in everyday life.

1.Functions Eliminate Redundancy

The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code. To explore the concept of redundancy, let's look at a function named summation, which returns the sum of the numbers within a given range of numbers. Here is the definition of summation, followed by a session showing its use:

```
def summation(lower, upper):
    """Arguments: A lower bound and an upper bound
    Returns: the sum of the numbers from lower through
    upper
    """
     result = 0
```

```
  while lower <= upper:
      result += lower
      lower += 1
  return result
>>> summation(1,4) # The summation of the numbers 1..4
10
>>> summation(50,100) # The summation of the numbers 50..100
3825
```

2.Functions Hide Complexity

Another way that functions serve as abstraction mechanisms is by hiding complicated details. To understand why this is true, let's return to the summation function. Although the idea of summing a range of numbers is simple, the code for computing a summation is not. We're not just talking about the amount or length of the code, but also about the number of interacting components. There are three variables to manipulate, as well as countcontrolled loop logic to construct.

3.Functions Support General Methods with Systematic Variations

An algorithm is a general method for solving a class of problems. The individual problems that make up a class of problems are known as problem instances. The problem instances for our summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed.

4.Functions Support the Division of Labor

In a well-organized system, whether it is a living thing or something created by humans, each part does its own job or plays its own role in collaborating to achieve a common goal. Specialized tasks get divided up and assigned to specialized agents. Some agents might assume the role of managing the tasks of others or coordinating them in some way. But, regardless of the task, good agents mind their own business and do not try to do the jobs of others.

Topic 5: Problem Solving with Top-Down Design

One popular design strategy for programs of any significant size and complexity is called top-down design. This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable sub problems—a process known as problem decomposition. As each sub problem is isolated, its solution is assigned to a function. Problem decomposition may continue down to lower levels, because a sub problem might in turn contain two or more lower-level problems to solve. As functions are developed to solve each sub problem, the solution to the overall problem is gradually filled out in detail. This process is also called stepwise refinement.

The Design of the Text-Analysis Program:

The program requires simple input and output components, so these can be expressed as statements within a main function. However, the processing of the input is complex enough to decompose into smaller sub processes, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores.

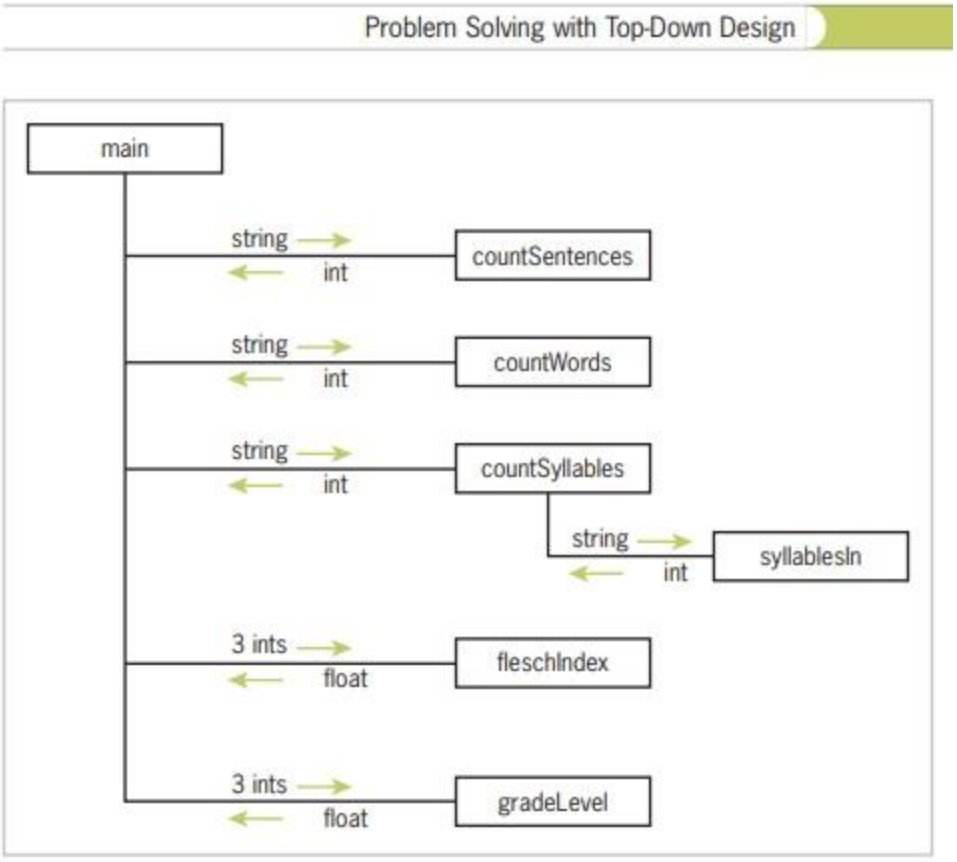


Figure 6-1 A structure chart for the text-analysis program

The Design of the Doctor Program

The doctor program processes the input by responding to it as an agent would in a conversation. Thus, the responsibility for responding is delegated to the reply function. Note that the two functions main and reply have distinct responsibilities. The job of main is to handle user interaction with the program, whereas reply is responsible for implementing the "doctor logic" of generating an appropriate reply. The assignment of roles and responsibilities to different actors in a program is also called responsibility-driven design.

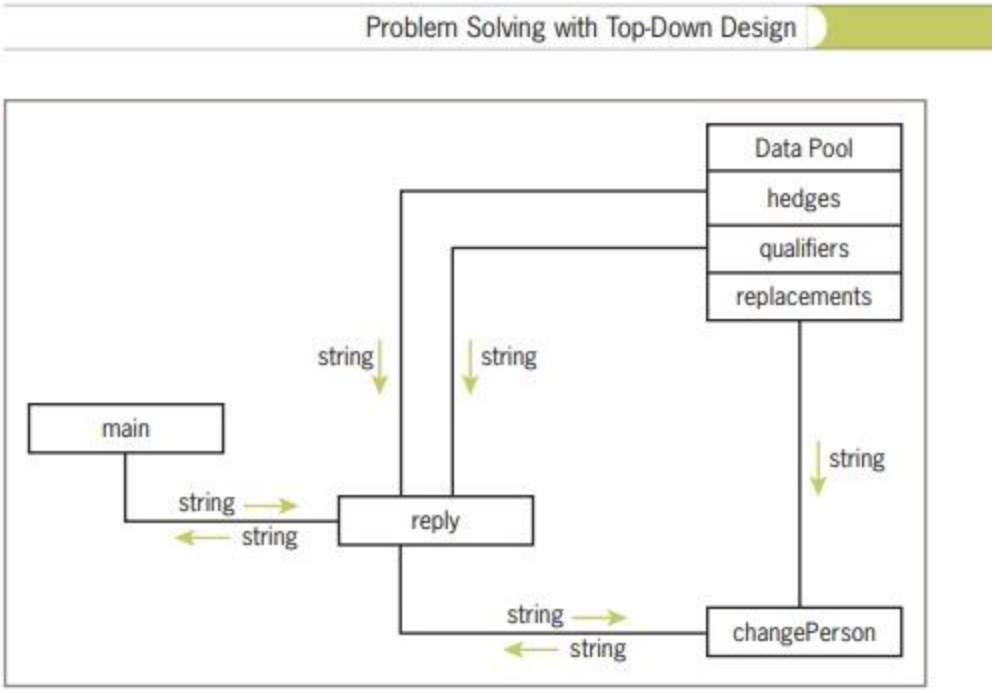


Figure 6-3 A structure chart for the doctor program

Topic 6: Design with Recursive Functions

In top-down design, you decompose a complex problem into a set of simpler problems and solve these with different functions. In some cases, you can decompose a complex problem into smaller problems of the same form. In these cases, the subproblems can all be solved by using

the same function. This design strategy is called recursive design, and the resulting functions are called recursive functions.

Defining a Recursive Function

A recursive function is a function that calls itself. To prevent a function from repeating itself indefinitely, it must contain at least one selection statement. This statement examines a condition called a base case to determine whether to stop or to continue with another recursive step.

Let's examine how to convert an iterative algorithm to a recursive function. Here is a definition of a function displayRange that prints the numbers from a lower bound to an upper bound:

```
def displayRange(lower, upper):
  """Outputs the numbers from lower through upper."""
   while lower <= upper:
      print(lower)
      lower = lower + 1
```

How would we go about converting this function to a recursive one? First, you should note two important facts:

1. The loop's body continues execution while lower <= upper.
2. When the function executes, lower is incremented by 1, but upper never changes.

The equivalent recursive function performs similar primitive operations, but the loop is replaced with a selection statement, and the assignment statement is replaced with a recursive call of the function. Here is the code with these changes:

```
def displayRange(lower, upper):
  """Outputs the numbers from lower through upper."""
```

```
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

Using Recursive Definitions to Construct Recursive Functions

Recursive functions are frequently used to design algorithms for computing values that have a recursive definition. A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases. For example, the Fibonacci sequence is a series of values with a recursive definition.

The first and second numbers in the Fibonacci sequence are 1. Thereafter, each number in the sequence is the sum of its two predecessors, as follows:

1 1 2 3 5 8 13 . . .

More formally, a recursive definition of the nth Fibonacci number is the following:

Fib(n) = 1, when n = 1 or n = 2

Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2

Given this definition, you can construct a recursive function that computes and returns the nth Fibonacci number. Here it is:

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
```

```
    else:
        return fib(n - 1) + fib(n - 2)
```

Note that the base case as well as the two recursive steps return values to the caller.

Infinite Recursion: Recursive functions tend to be simpler than the corresponding loops, but they still require thorough testing. One design error that might trip up a programmer occurs when the function can (theoretically) continue executing forever, a situation known as infinite recursion.

Topic 7: Managing a Program's Namespace

namespace—that is, the set of its variables and their values—is structured and how you can control it via good design principles.

Module Variables, Parameters, and Temporary Variables

The below program includes many variable names; for the purposes of this example, we will focus on the code for the variable replacements and the function changePerson

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

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)
```

This code appears in the file doctor.py, so its module name is doctor. The names in this code fall into four categories, depending on where they are introduced:

This code appears in the file doctor.py, so its module name is doctor. The names in this code fall into four categories, depending on where they are introduced:

1. Module variables. The names replacements and changePerson are introduced at the level of the module. Although replacements names a dictionary and changePersonnames a function, they are both considered variables. You can see the module variables of the doctor module by importing it and entering dir(doctor) at a shell prompt. When module variables are introduced in a program, they are immediately given a value.

2. Parameters. The name sentence is a parameter of the function changePerson. A parameter name behaves like a variable and is introduced in a function or method header. The parameter does not receive a value until the function is called.

3. Temporary variables. The names words, replyWords, and word are introduced in the body of the function changePerson. Like module variables, temporary variables receive their values as soon as they are introduced.

4. Method names. The names split and join are introduced or defined in the strtype. As mentioned earlier, a method reference always uses an object, in this case, a string, followed by a dot and the method name.

Topic 8: Higher-Order Functions

Like any skill, a designer's knack for spotting the need for a function is developed with practice. As you gain experience in writing programs, you will learn to spot common and redundant patterns in the code. One pattern that occurs again and again is the application of a function to a set of values to produce some results. Here are some examples:

• The numbers in a text file must be converted to integers or floats after they are input.

• The first-person pronouns in a list of words must be changed to the corresponding

second-person pronouns in the doctor program.

• Only scores above the average are kept in a list of grades.

• The sum of the squares of a list of numbers is computed.

In this section, we learn how to capture these patterns in a new abstraction called a higher-order function. For these patterns, a higher-order function expects a function and a set of data values as arguments.

Topic:10 Modules

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Example: create a simple module

```
# A simple module, calc.py
  def add(x, y):
    return (x+y)
  def subtract(x, y):
    return (x-y)
```

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Topic:11 Python Packages

We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently.

For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this.

The same analogy is followed by the Python package.

Creating Package:

- Let's create a package named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps –
- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. __init__.py
- Then create two modules mod1 and mod2 in this folder.

Mod1.py

- def gfg():
- print("Welcome to GFG")

Mod2.py

- def sum(a, b):
- return a+b

Understanding __init__.py

- _init__.py helps the Python interpreter to recognise the folder as package.
- It also specifies the resources to be imported from the modules. If the __init__.py is empty

__init__.py

- from .mod1 import gfg
- from .mod2 import sum
- This __init__.py will only allow the gfg and sum functions from the mod1 and mod2 modules to be imported.

Import Modules from a Package

- We can import these modules using the from…import statement and the dot(.) operator.
- **Syntax:**

Import package_name.module_name

Example: Import Module from package

- We will import the modules from the above created package and will use the functions inside those modules.
- from mypckg import mod1
- from mypckg import mod2
- mod1.gfg()
- res = mod2.sum(1, 2)
- print(res)

Output:

- Welcome to GFG
- 3

Topic:12 Case Study: Gathering Information from a File System

CASE STUDY: Gathering Information from a File System

Modern file systems come with a graphical browser, such as Microsoft's Windows Explorer or Apple's Finder. These browsers allow the user to navigate to files or folders by selecting icons of folders, opening these by double-clicking, and selecting commands from a drop-down menu. Information on a folder or a file, such as the size and contents, is also easily obtained in several ways.

Users of terminal-based user interfaces (see Chapter 2) must rely on entering the appropriate commands at the terminal prompt to perform these functions. In this case study, we develop a simple terminal-based file system navigator that provides some information about the system. In the process, we will have an opportunity to exercise some skills in top-down design and recursive design.

Request

Write a program that allows the user to obtain information about the file system.

Analysis

File systems are tree-like structures, as shown in Figure 6-5.

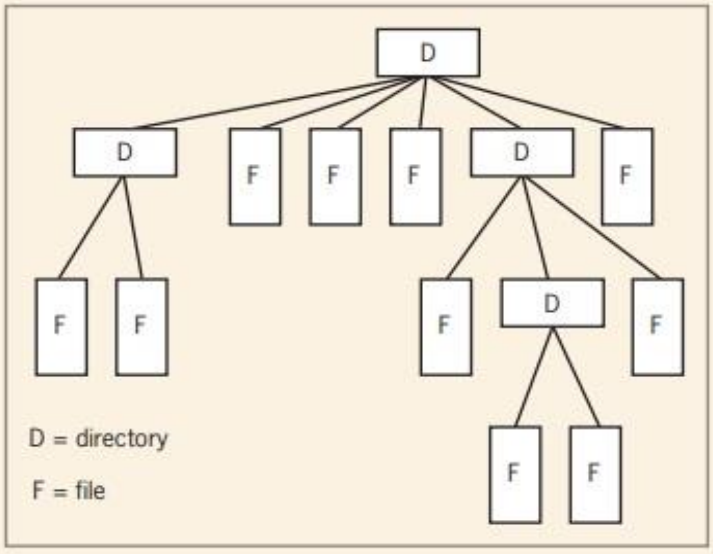


Figure 6-5 The structure of a file system

At the top of the tree is the **root directory** (the term "directory" is a synonym for "folder," among users of terminal-based systems). Under the root are files and subdirectories. Each directory in the system except the root lies within another directory called its **parent**. For example, in Figure 6-5, the root directory contains four files and two subdirectories. On a UNIX-based file system (the system that underlies macOS), the **path** to a given file or directory in the system is a string that starts with the / (forward slash) symbol (the root), followed by the names of the directories traversed to reach the file or directory. The / (forward slash) symbol also separates each name in the path. Thus, the path to the file for this chapter on Ken's laptop might be the following:

```
/Users/KenLaptop/Book/Chapter6/Chapter6.doc
```

On a Windows-based file system, the \ symbol is used instead of the / symbol.

The program we will design in this case study is named **filesys.py**. It provides some basic browsing capability as well as options that allow you to search for a given filename and find statistics on the number of files and their size in a directory. At program startup, the current working directory (CWD) is the directory containing the Python program file. The program should display the path of the CWD, a menu of command options, and a prompt for a command, as shown in Figure 6-6.

```
/Users/KenLaptop/Book/Chapter6
1   List the current directory
2   Move up
3   Move down
4   Number of files in the directory
5   Size of the directory in bytes
6   Search for a filename
7   Quit the program
Enter a number:
```

Figure 6-6 The command menu of the **filesys** program

When the user enters a command number, the program runs the command, which may display further information, and the program displays the CWD and command menu again. An unrecognized command produces an error message, and command number 7 quits the program. Table 6-1 summarizes what the commands do.

(continues)

(continued)

Command	What It Does
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the CWD, print an error message; otherwise, move to this directory and make it the CWD.
Number of files in the directory	Prints the number of files in the CWD and all of its subdirectories.
Size of the directory in bytes	Prints the total number of bytes used by the files in the CWD and all of its subdirectories.
Search for a filename	Prompts the user for a search string. Prints a list of all the filenames (with their paths) that contain the search string, or "String not found."
Quit the program	Prints a signoff message and exits the program.

Table 6-1 The commands in the **filesys** program

Design

You can structure the program according to two sets of tasks: those concerned with implementing a menu-driven command processor, and those concerned with executing the commands. The first group of operations includes the `main` function. In the following discussion, we work top-down and begin by examining the first group of operations.

As in many of the programs we have examined recently in this book, the `main` function contains a driver loop. This loop prints the CWD and the menu, calls other functions to input and run the commands, and breaks with a signoff message when the command is to quit. Here is the pseudocode:

```
function main()
    while True
        print(os.getcwd())
        print(MENU)
        command = acceptCommand()
        runCommand(command)
        if command == QUIT
            print("Have a nice day!")
            break
```

(continues)

(continued)

The function **os.getcwd** returns the path of the CWD. Note also that **MENU** and **QUIT** are module variables initialized to the appropriate strings before **main** is defined. The **acceptCommand** function loops until the user enters a number in the range of the valid commands. These commands are specified in a tuple named **COMMANDS** that is also initialized before the function is defined. The function thus always returns a valid command number.

The **runCommand** function expects a valid command number as an argument. The function uses a multi-way selection statement to select and run the operation corresponding to the command number. When the result of an operation is returned, it is printed with the appropriate labeling.

That's it for the menu-driven command processor in the **main** function. Although there are other possible approaches, this design makes it easy to add new commands to the program.

The operations required to list the contents of the CWD, move up, and move down are simple and need no real design work. They involve the use of functions in the **os** and **os.path** modules to list the directory, change it, and test a string to see if it is the name of a directory. The implementation shows the details.

The other three operations all involve traversals of the directory structure in the CWD. During these traversals, every file and every subdirectory are visited. Directory structure is in fact recursive: each directory can contain files (base cases) and other directories (recursive steps). Thus, we can develop a recursive design for each operation.

The **countFiles** function expects the path of a directory as an argument and returns the number of files in this directory and its subdirectories. If there are no subdirectories in the argument directory, the function just counts the files and returns this value. If there is a subdirectory, the function moves down to it, counts the files (recursively) in it, adds the result to its total, and then moves back up to the parent directory. Here is the pseudocode:

```
function countFiles(path)
    count = 0
    lyst = os.listdir(path)
    for element in lyst
        if os.path.isfile(element)
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("..")
    return count
```

(continues)

The **countBytes** function expects a path as an argument and returns the total number of bytes in that directory and its subdirectories. Its design resembles **countFiles**.

The **findFiles** function accumulates a list of the filenames, including their paths, that contain a given target string, and returns this list. Its structure resembles the other two recursive functions, but the **findFiles** function builds a list rather than a number. When the function encounters a target file, its name is appended to the path, and then the result string is appended to the list of files. We use the module variable **os.sep** to obtain the appropriate slash symbol (/ or \) on the current file system. When the function encounters a directory, it moves to that directory, calls itself with the new CWD, and extends the files list with the resulting list. Here is the pseudocode:

```
function findFiles(target, path)
    files = []
    lyst = os.listdir(path)
    for element in lyst
        if os.path.isfile(element):
            if target in element:
                files.append(path + os.sep + element)
            else:
                os.chdir(element)
        files.extend(findFiles(target, os.getcwd()))
        os.chdir("..")
    return files
```

The trick with recursive design is to spot elements in a structure that can be treated as base cases (such as files) and other elements that can be treated as recursive steps (such as directories). The recursive algorithms for processing these structures flow naturally from these insights.

Implementation (Coding)

Near the beginning of the program code, we find the important variables, with the functions listed in a top-down order.

```
"""
Program: filesys.py
Author: Ken
Provides a menu-driven tool for navigating a file system
and gathering information on files.
"""

import os, os.path

QUIT = '7'
COMMANDS = ('1', '2', '3', '4', '5', '6', '7')
```

(continues)

(continued)

```
MENU = """1 List the current directory
2 Move up
3 Move down
4 Number of files in the directory
5 Size of the directory in bytes
6 Search for a filename
7 Quit the program"""

def main():
    while True:
        print(os.getcwd())
        print(MENU)
        command = acceptCommand()
        runCommand(command)
        if command == QUIT:
            print("Have a nice day!")
            break

def acceptCommand():
    """Inputs and returns a legitimate command number."""
    command = input("Enter a number: ")
    if command in COMMANDS:
        return command
    else:
        print("Error: command not recognized")
        return acceptCommand()

def runCommand(command):
    """Selects and runs a command."""
    if command == '1':
        listCurrentDir(os.getcwd())
    elif command == '2':
        moveUp()
    elif command == '3':
        moveDown(os.getcwd())
    elif command == '4':
        print("The total number of files is", \
        countFiles(os.getcwd()))
    elif command == '5':
        print("The total number of bytes is", \
        countBytes(os.getcwd()))
    elif command == '6':
        target = input("Enter the search string: ")
        fileList = findFiles(target, os.getcwd())
        if not fileList:
            print("String not found")
        else:
```

(continues)

(continued)

```
            for f in fileList:
                print(f)

def listCurrentDir(dirName):
    """Prints a list of the cwd's contents."""
    lyst = os.listdir(dirName)
    for element in lyst: print(element)

def moveUp():
    """Moves up to the parent directory."""
    os.chdir("..")

def moveDown(currentDir):
    """Moves down to the named subdirectory if it exists."""
    newDir = input("Enter the directory name: ")
    if os.path.exists(currentDir + os.sep + newDir) and \
       os.path.isdir(newDir):
        os.chdir(newDir)
    else:
        print("ERROR: no such name")

def countFiles(path):
    """Returns the number of files in the cwd and
    all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("..")
    return count

def countBytes(path):
    """Returns the number of bytes in the cwd and
    all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += os.path.getsize(element)
        else:
            os.chdir(element)
```

(continues)

(continued)

```
            count += countBytes(os.getcwd())
            os.chdir("..")
    return count

def findFiles(target, path):
    """Returns a list of the filenames that contain
    the target string in the cwd and all its subdirectories."""
    files = []
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            if target in element:
                files.append(path + os.sep + element)
            else:
                os.chdir(element)
                files.extend(findFiles(target, os.getcwd()))
                os.chdir("..")
    return files

if __name__ == "__main__":
    main()
```