UNIT- 3

LISTS , DICTIONARIES, FUNCTIONS AND MODULES

List and Dictionaries: Lists, Defining Simple Functions, Dictionaries.
Design with Function: Functions as Abstraction Mechanisms, Problem Solving with Top Down Design, Design with Recursive Functions, Case Study Gathering Information from a File System, Managing a Program's Namespace, Higher Order Function.
Modules: Modules, Standard Modules, Packages

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
- Each of the items in a list is ordered by position.
- Like a character in a string, each item in a list has a unique index that specifies its position.
- The index of the first item is 0, and the index of the last item is the length of the list minus 1

List Literals and Basic Operators :

- In Python, a list literal is written as a sequence of data values separated by commas.
- The entire sequence is enclosed in square brackets ([and]).
- Here are some example list literals:
  - [1951, 1969, 1984]   # A list of integers
  - ["apples", "oranges", "cherries"]   # A list of strings
  - []   # An empty list
- You can also use other lists as elements in a list, thereby creating a list of lists. Here is one example of such a list:
  - [[5, 9], [541, 78]]
- The Python interpreter evaluates a list literal, and each of the elements are also evaluated if required

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

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

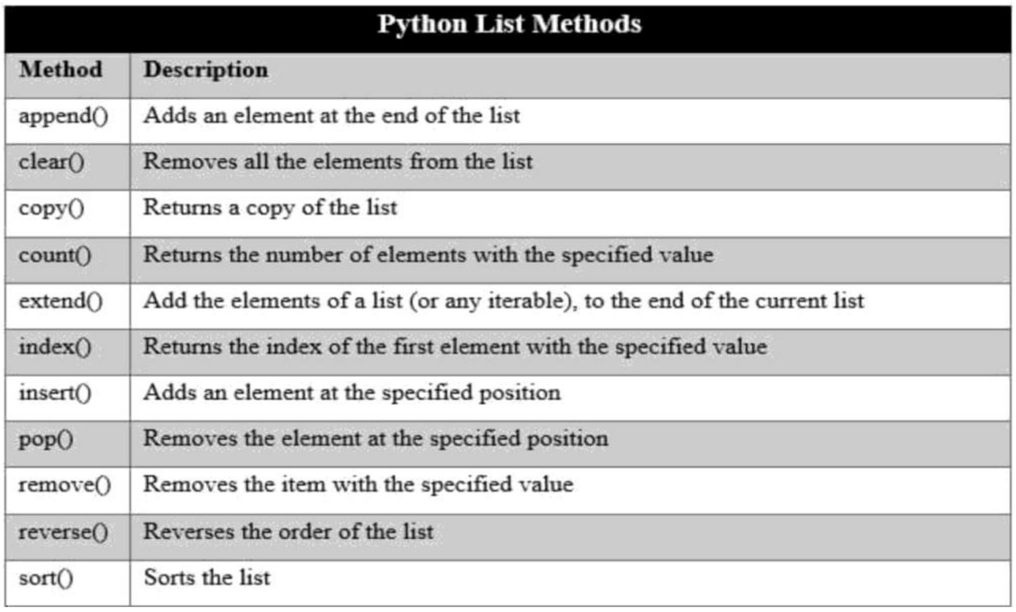
- You can also build lists of integers using the range and list functions

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

- The list function can build a list from any iterable sequence of elements, such as a string:

```
>>> third = list("Hi there!")
>>> third
['H', 'i', ' ' , 't', 'h', 'e', 'r', 'e', '!']
```

List Methods :



Python List Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

- **append() and extend()** :
  - 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]
```

- **pop() :**
  - The method pop is used to remove an element at a given position. If the position is not specified, pop removes and returns the last element.
  - In that case, the elements that followed the removed element are shifted one position to the left

```
>>> 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**
  - first use the **in** operator to test for presence and then the **index** method if this test returns True.
  - The next code segment shows how this is done for an example list and target element:

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

- **Sorting a List** :
  - When the elements can be related by comparing them for less than and greater than as well as equality, they can be sorted.
  - The list 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]
```

NOTE:

- **Mutator Methods and the Value None :**
  - Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called **mutators. Examples** are the list methods **insert, append, extend, pop**, and **sort**

Dictionaries:

- A dictionary organizes information by **association, not position.**
- **For example, when you** use an english dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn directly to the words beginning with "M."
  - Phone books, address books, encyclopedias, and other reference sources also organize information by association.
- In computer science, data structures organized by association are also called **tables or association lists.**
- **In** Python, a **dictionary associates a set of keys with values.**

Dictionary Literals:

- A Python dictionary is written as a sequence of key/value pairs separated by commas.
- These pairs are sometimes called **entries. The entire sequence of entries is enclosed in curly braces** ({ and }).
- A colon (:) separates a key and its value. Here are some example dictionaries:
  - phonebook= {"Savannah":"476-3321", "Nathaniel":"351-7743"}
  - Info={"Name":"Molly", "Age":18}
- You can even create an empty dictionary—that is, a dictionary that contains no entries.
  - {}

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'}
```

- The same operation is used for two different purposes: insertion of a new entry and modification of an existing entry.

Accessing Values :

- You can also use the subscript to obtain the value associated with a key. However, if the key is not present in the dictionary, Python raises an exception.

```
>>>info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
```

```
info["job"]
KeyError: 'job'
```

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

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:

```
>>>info ={“name”:”Surya”,”phone”:9876543211}
>>>for key in info:
                print(key, info[key])
phone 9876543211
name Surya
```

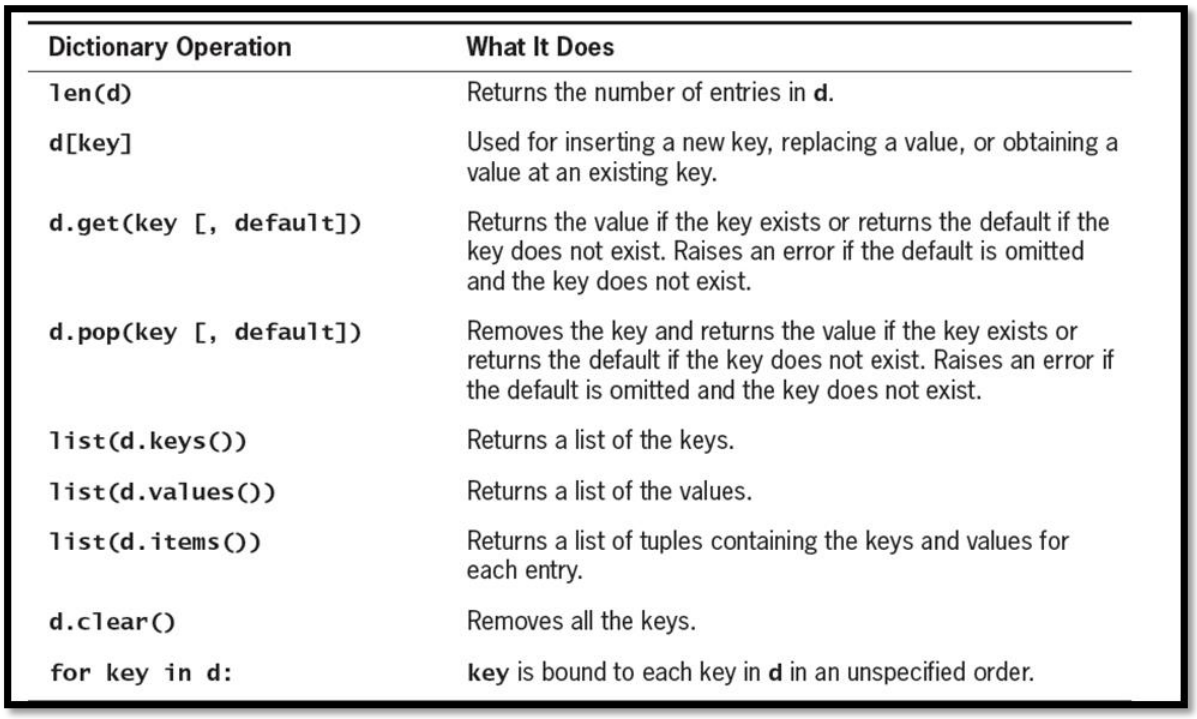
- The entries are represented as tuples within the list. A tuple of variables can then access the key and value of each entry in this list within a for loop:

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

Gives same output as the previous one

- On each pass through the loop, the variables key and value within the tuple are assigned the key and value of the current entry in the list. The use of a structure containing variables to access data within another structure is called **pattern matching**.

Dictionary Operations :



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.

```
d={1:1,2:2**3,3:3**3,4:4**3,5:5**3,6:6**3}
>>> d
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216}
>>> len(d)
6
>>> d[5]
125
>>> d.get(4)
64
>>>d.pop(4)
64
>>> d
{1: 1, 2: 8, 3: 27, 5: 125, 6: 216}
>>> d.keys()
dict_keys([1, 2, 3, 5, 6])
>>> d.values()
dict_values([1, 8, 27, 125, 216])
>>> d.items()
dict_items([(1, 1), (2, 8), (3, 27), (5, 125), (6, 216)])
```

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

Conversion of hexadecimal to Binary:

- The algorithm visits each digit in the hexadecimal number, selects the corresponding four bits that represent that digit in binary, and adds these bits to a result string.
- If you maintain the set of associations between hexadecimal digits and binary digits in a dictionary, then you can just look up each hexadecimal digit's binary equivalent with a subscript operation. Such a dictionary is sometimes called a lookup table. Here is the definition of the lookup table required for hex-to-binary conversions:
- 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):
binary = ""
for digit in number:
        binary = binary + table[digit]
return binary

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

FUNCTIONS:

1. Design with Function

- A function packages an algorithm in a chunk of code that you can call by name
- A function can be called from anywhere in a program's code, including code within other functions
- A function can receive data from its caller via **arguments**
- When a function is called, any expressions supplied as arguments are first evaluated.
- Their values are copied to temporary storage locations named by the parameters in the function's definition
- A function may have one or more **return** statements, whose purpose is to terminate the execution of the function and return control to its caller. A return statement may be followed by an expression.

1.1 Functions as Abstraction Mechanisms

- Human brain can wrap itself around just a few things at once , People cope with complexity by developing a mechanism to simplify or hide it. This mechanism is called an **abstraction.**
- An abstraction hides detail and thus allows a person to view many things as just one thing
- **"doing my laundry" :** This expression is simple, but it refers to a complex process that involves
  - fetching dirty clothes from the hamper,
  - separating them into whites and colors,
  - loading them into the washer,
  - Transferring them to the dryer, and
  - folding them and
  - putting them into the dresser
- Without abstractions, most of our everyday activities would be impossible to discuss, plan, or carry out. Likewise, effective designers must invent useful abstractions to control complexity.

1.2 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.

```
def summation(lower, 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
```

- Code redundancy is bad for several reasons. For one thing, it requires the programmer to laboriously enter or copy the same code over and over, and to get it correct every time.
- Then, if the programmer decides to improve the algorithm by adding a new feature or making it more efficient, he or she must revise each instance of the redundant code throughout the entire program leading to many maintainance problems
- By relying on a single function definition, instead of multiple instances of redundant code, the programmers free themselves to write only a single algorithm in just one place—say, in a library module.
- Any other module or program can then import the function for its use. Once imported, the function can be called as many times as necessary.
- When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition. There is no need to edit the parts of the program that call the function

1.3 Functions Hide Complexity

- Functions serve as abstraction mechanisms is by hiding complicated details.
- A function call expresses the idea of a process to the programmer, without forcing him or her to wade through the complex code that realizes that idea
  - In summation function, although the idea of summing a range of numbers is simple, the code for computing a summation is not.

  - There are three variables to manipulate, as well as count-controlled loop logic to construct.

1.4 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 the summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed.
- The summation function contains both the code for the summation algorithm and the means of supplying problem instances to this algorithm. The problem instances are the data sent as arguments to the function.

1.5 Functions Support the Division of Labor

- In a computer program, functions can enforce a division of labor.
- Ideally, each function performs a single coherent task, such as computing a summation or formatting a table of data for output.
- Each function is responsible for using certain data, computing certain results, and returning these to the parts of the program that requested them.
- Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions.

2 . Problem Solving with Top-Down Design

- The top down 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 subproblem is isolated, its solution is** assigned to a function. Problem decomposition may continue down to lower levels, because a subproblem might in turn contain two or more lower-level problems to solve.
- As functions are developed to solve each subproblem, the solution to the overall problem is gradually filled out in detail. This process is also called **stepwise refinement.**

2.1 The Design of the Text-Analysis Program

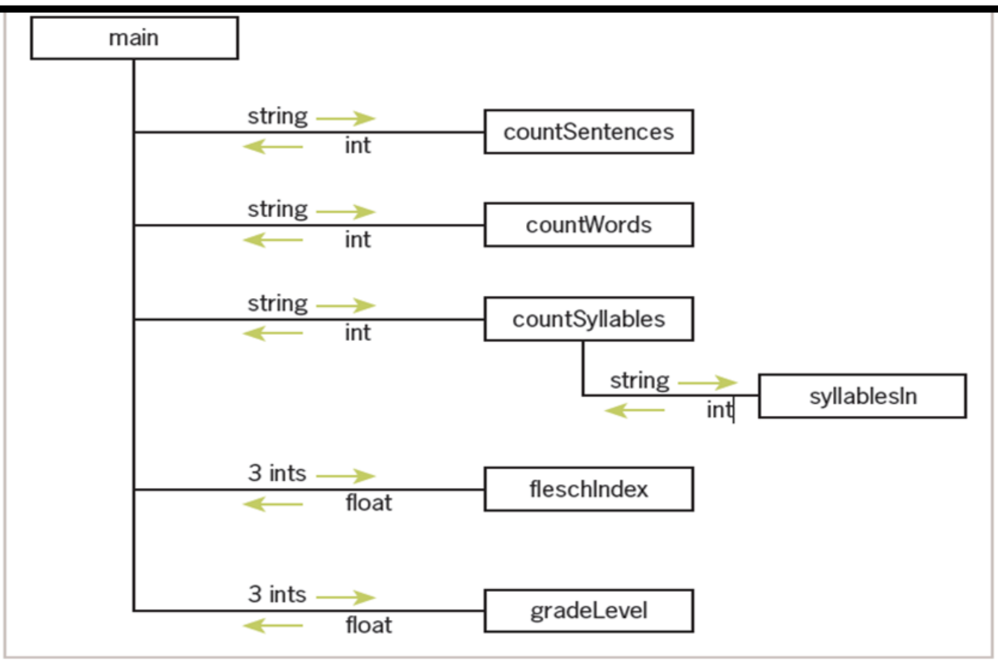


Figure 6-1 A structure chart for the text-analysis program

- The program requires simple input and output components, so these can be expressed as statements within a main function.
- The processing of the input is complex enough to decompose into smaller subprocesses, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores.
- We develop a new function for each of these computational tasks. The relationships among the functions in this design are expressed in the structure chart

Structure chart

- A **structure chart is a** diagram that shows the relationships among a program's functions and the passage of data between them.
- Each box in the structure chart is labeled with a function name. The main function at the top is where the design begins, and decomposition leads us to the lower-level functions on which main depends.
- The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them. For example, the function countSentences takes a string as an argument and returns the number of sentences in that string.
- Note that all functions except one are just one level below main

3 . Design with Recursive Functions

- In some cases of top down design , 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.**

#Python **recursive** function for summation

```
def  summation(lower, upper):
        """Returns the sum of the numbers from lower through upper."""
        if   lower > upper:
                return 0
        else:
                return lower + summation (lower + 1, upper)
```

The recursive call of summation adds up the numbers from lower + 1 through upper .The function then adds lower to this result and returns it.

Using Recursive Definitions to Construct Recursive Functions

- 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)
```

Infinite Recursion:

- Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.
- In fact, the Python virtual machine eventually runs out of memory resources to manage the process, so it halts execution with a message indicating a **stack overflow error.**
- **The next session defines a** function that leads to this result:

```
def runForever(n):
        if n > 0:
                runForever(n)
        else:
                runForever(n - 1)
>>> runForever(1)
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
runForever(1)
File "<pyshell#5>", line 3, in runForever
runForever(n)
File "<pyshell#5>", line 3, in runForever
runForever(n)
File "<pyshell#5>", line 3, in runForever
runForever(n)
[Previous line repeated 989 more times]
File "<pyshell#5>", line 2, in runForever
if n > 0:
```

```
RecursionError: maximum recursion depth exceeded in comparison
```

The PVM keeps calling runForever(1) until there is no memory left to support another recursive call. Unlike an infinite loop, an infinite recursion eventually halts execution with an error message.

The Costs and Benefits of Recursion :

- The run-time system on a real computer, such as the PVM(Python Virtual Machine), must devote some overhead to recursive function calls.
- At program startup, the PVM reserves an area of memory named a **call stack. For each call of a function,** recursive or otherwise, the PVM must allocate on the call stack a small chunk of memory called a **stack frame.**
- In this type of storage, the system places the values of the arguments and the return address for each function call. Space for the function call's return value is also reserved in its stack frame.
- When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code, and the memory for the stack frame is deallocated.
- When, because of a design error, the recursion is infinite, the stack frames are added until the PVM runs out of memory, which halts the program with an error message.

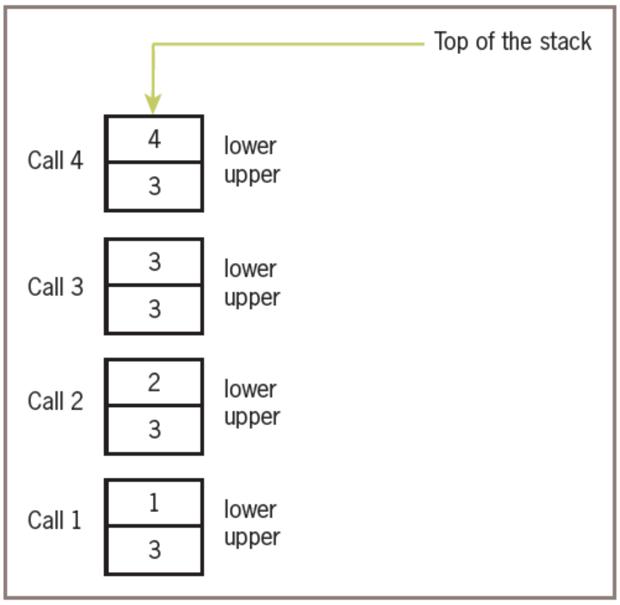


Figure 6-4 The stack frames for `displayRange(1, 3)`

4. Case Study Gathering Information from a File System

- Modern file systems come with a graphical browser, allowing 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 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:

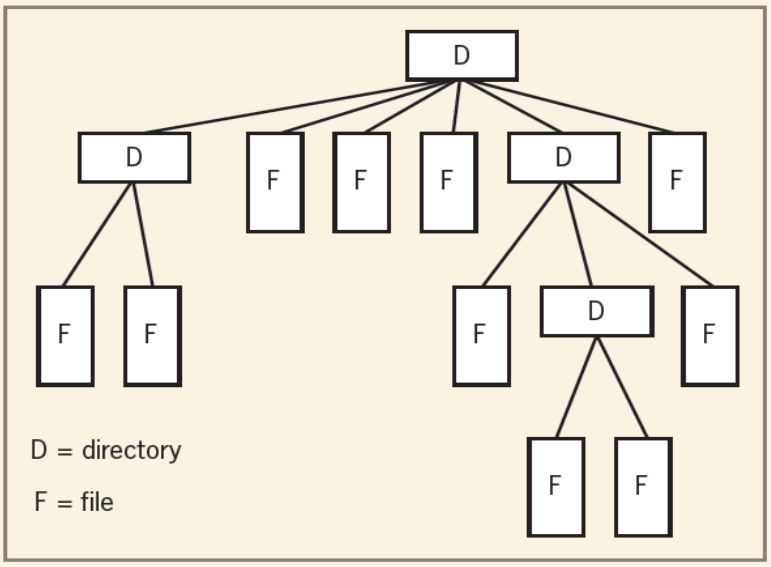
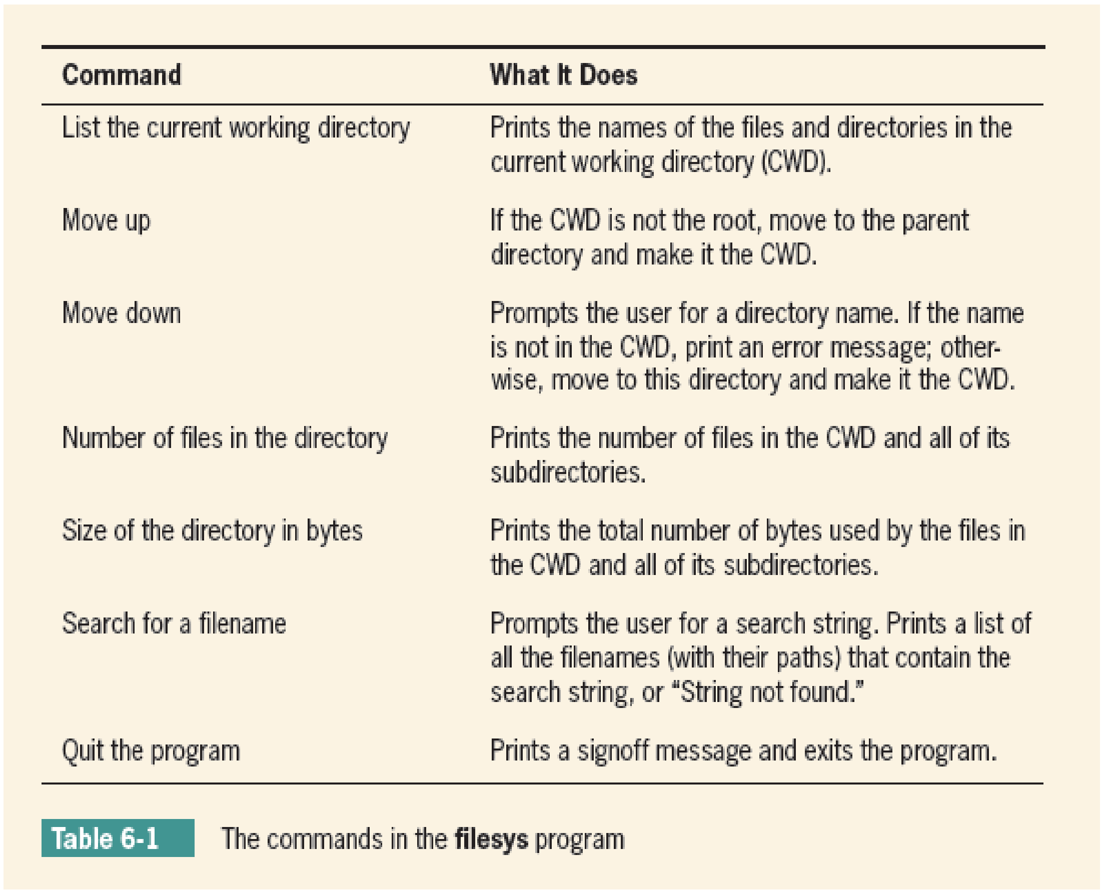


Figure 6-5 The structure of a file system



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

```
import os, os.path
QUIT = '7'
COMMANDS = ('1', '2', '3', '4', '5', '6', '7')

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:
```

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

```
        print(os.getcwd())
        print(MENU)
        command = acceptCommand() #takes choice
        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:
            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)
            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()
```

5. Managing a Program’s Namespace

Namespaces in Python

- A namespace is a collection of currently defined symbolic names along with information about the object that each name references.
- You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves.
  - Each key-value pair maps a name to its corresponding object
- In a Python program, there are four types of namespaces:
  - Built-In
  - Global
  - Enclosing
  - Local

i) The Built-In Namespace

- The **built-in namespace** contains the names of all of Python’s built-in objects. These are available at all times when Python is running.
- You can list the objects in the built-in namespace with the following command:

```
>>> dir(__builtins__)
```

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

ii) The Global Namespace

- The **global namespace** contains any names defined at the level of the main program.
- Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.
- The interpreter also creates a global namespace for any **module** that your program loads with the import statement.

iii) The Local and Enclosing Namespaces

The interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates

```
def f():
    print('Start f()')
    def g():
        print('Start g()')
        print('End g()')
        return
    g()
    print('End f()')
    return
```

Output :

```
>>> f()
Start f()
Start g()
End g()
End f()
```

- When the main program calls f(), Python creates a new namespace for f(). Similarly, when f() calls g(), g() gets its own separate namespace.
- The namespace created for g() is the **local namespace**, and the namespace created for f() is the **enclosing namespace**.
- Each of these namespaces remains in existence until its respective function terminates.

Scope:

- In Python, a name's scope is the area of program text in which the name refers to a given value
- In general, the meanings of temporary variables are restricted to the body of the functions in which they are introduced, and they are invisible elsewhere in a module.
- The restricted visibility of temporary variables befits their role as temporary working storage for a function.
- Although a Python function can reference a module variable for its value, it cannot under normal circumstances assign a new value to a module variable.
- When such an attempt is made, the PVM creates a new, temporary variable of the same name within the function.
- The following script shows how this works:

```
x = 5
def f():
        x = 10   # Attempt to reset x
f()                    # Does the top-level x change?
print(x)               # No, this displays 5
```

- When the function f is called, it does not assign 10 to the module variable x; instead, it assigns 10 to a temporary variable x.
- In fact, once the temporary variable is introduced, the module variable is no longer visible within function f. In any case, the module variable's value remains unchanged by the call

Lifetime:

- A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it.
- When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.
- The concept of lifetime explains the existence of two variables called x in our last example session.
  - The module variable x comes into existence before the temporary variable x and survives the call of function f.
  - During the call of f, storage exists for both variables, so their values remain distinct.
- **Using Keywords for Default and Optional Arguments:**
- The programmer can also specify **optional arguments with default values in any function** definition.
- Here is the syntax:

  def <function name>(<required arguments>, <key-1> = <val-1>, ... <key-*n*> = <*val-n*>)
- The required arguments are listed first in the function header. These are the ones that are "essential" for the use of the function by any caller.
- Following the required arguments are one or more **default arguments or keyword arguments.** These are assignments of values to the argument names. When the function is called without these arguments, their default values are automatically assigned to them. When the function is called with these arguments, the default values are overridden by the caller's values.
- When using functions that have default arguments, you must provide the required

arguments and place them in the same positions as they are in the function definition's header.

- The default arguments that follow can be supplied in two ways:

1. **By position**. In this case, the values are supplied in the order in which the arguments occur in the function header. Defaults are used for any arguments that are omitted.
2. **By keyword**. In this case, one or more values can be supplied in any order, using the syntax **<key> = <value>** in the function call.

- Here is an example of a function with one required argument and two default arguments and a session that shows these options:

```
>>> def example(required, option1 = 2, option2 = 3):
                        print(required, option1, option2)
>>> example(1)                  # Use all the defaults
1 2 3
>>> example(1, 10)    # Override the first default
1 10 3
>>> example(1, 10, 20)        # Override all the defaults
1 10 20
>>> example(1, option2 = 20) # Override the second default
1 2 20
>>> example(1, option2 = 20, option1 = 10) # In any order
1    10 20
```

6. Anonomyous Function or Lambda function:

- An anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.

EXAMPLE 1:

```
>>> d = lambda x: x * 2
>>> print(d(5))
10
```

Surya Lakshmi Kantham Vinti
CSE Dept
ACET

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions are used along with built-in functions like filter(), map() etc.

EXAMPLE 2 :Lambda with filter():

- The filter() function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list = list(filter(lambda x: (x%2 == 0) , my_list))
>>> print(new_list)
[4, 6, 8, 12]
```

EXAMPLE 3: Lambda with map():

- The map() function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

```
>>> my_list = [1, 5, 4, 6, 8, 11, 3, 12]
>>> new_list=list(map(lambda x:x**2 , my_list))
>>> new_list
[1, 25, 16, 36, 64, 121, 9, 144]
```

7. Higher Order Functions

- A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as Higher order Functions
- The 3 mostly used higher order functions are:
  - map()
  - filter()
  - reduce()

map() :

- **map()** function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)
- **Syntax :**

  map(fun, iter)
- **Parameters :**
  - **fun :** It is a function to which map passes each element of give iterable.
  - **iter :** It is a iterable which is to be mapped.

```
# Python program to demonstrate working  of map.
# Return double of n
def addition(n):
   return n + n
# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output:

```
[2, 4, 6, 8]
```

- **filter()**
  - The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
  - **Syntax:**

    filter(function, sequence)
  - **Parameters:**
    - function: function that tests if each element of a sequence true or not.
    - sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.
  - **Returns:** returns an iterator that is already filtered.

```
# function that filters vowels
def fun(variable):
   letters = ['a', 'e', 'i', 'o', 'u']
```

```
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
# using filter function
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
    print(s)
```

OUTPUT :

```
The filtered letters are: e e
```

- **reduce() :**
  - The Python functools module includes a reduce function that expects a function of two arguments and a list of values. The reduce function returns the result of applying the function as just described.
  - The following example shows reduce used twice—once to produce a sum and once to produce a product:

```
>>> from functools import reduce
>>> def   add(x, y):
                return  x + y
>>> def   multiply(x, y):
                return  x * y
>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
```

8. Modules in Python:

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for example: example.py, is called a module, and its module name would be example
- Modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

User defined module :

Let us create a module. Type the following and save it as example.py.

```
# Python Module example
def add(a, b):
   """This program adds two  numbers and return the result"""
      result = a + b
       return result
```

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

```
>>> import example
```

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)
9.5
```

- Modules are imported by using import statement

Syntax:

i) import module_name

Example:

```
>>>import math
>>>print(math.sqrt(25))
5.0
```

ii) from….import statement:

A module may contain definition of many functions and variables.

When you import a module, you can use any variable or any function defined in that module but if we want to use only selective variables and functions then we will use the “from……import statement”

Syntax:

from module_name import function_name/variable_name

e.g.1

```
>>>from time import asctime
print(asctime())
Thu Aug 26 15:08:52 2021
```

e.g.2

```
>>>from math import pi
>>>print("pi= ", pi)
```

To import more than one item from the module, we use a comma separated list like below

```
from math import sqrt, pow
print(sqrt(25), pow(10,2))
```

iii) "as keyword":

To avoid the confusion in function names we use as keyword to give a alias name

e.g.

```
>>>from math import sqrt as square_root
>>>print(square_root(25))
```

Creating a module: num.py

```
def square(x):
        return(x*x)
def cube(x):
        return(x*x*x)
def power(x, y):
    return(x**y)
```

Example program:

```
import  num
print("Square of 10",num.square(10))
print("Cube of 10",num.cube(10))
print("Power of 10, 2 is ",num.power(10,5))
```

9. Packages in Python:

Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files

Creating a Package:

Let's create a package named mypackage, using the following steps:

- Create a new folder named C:\MyApp.
- Inside MyApp, create a subfolder with the name 'mypackage'.
- Create an empty **__init__**.py file in the mypackage folder.
- Using a Python-aware editor like IDLE, create modules greet.py and functions.py with the following code:

  greet.py

  ```
  def  SayHello(name):
      print("Hello ", name)
  ```

  functions.py

  ```
   def sum(x,y):
          return x+y
  def average(x,y):
  ```

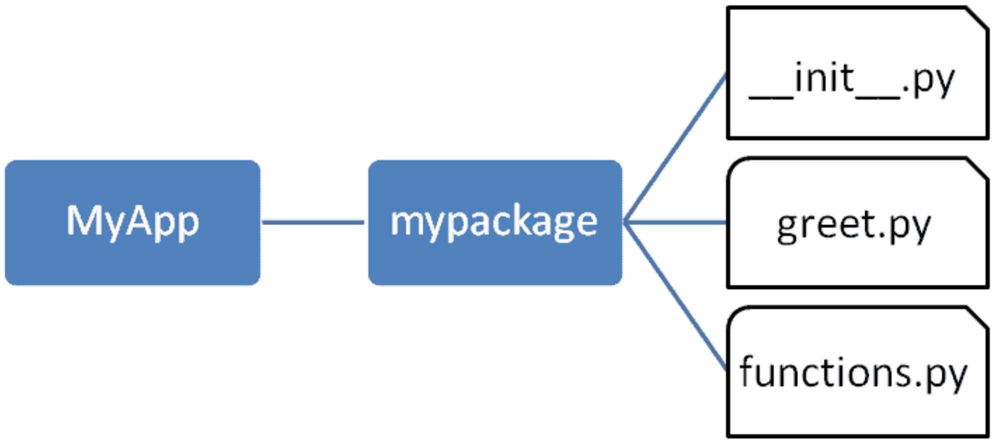
```
        return (x+y)/2
def power(x,y):
        return x**y
```

__init__.py :

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

- The Python interpreter recognizes a folder as the package if it contains __init__.py file.
- __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from the above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package.



- Import the functions module from the mypackage package and call its power() function.

```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

- It is also possible to import specific functions from a module in the package.

```
>>> from mypackage.functions import sum
>>> sum(10,20)
```

```
30
>>> average(10,12)
Traceback (most recent call last):
File "<pyshell#13>", line 1, in <module>
NameError: name 'average' is not defined
```