**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

**Dictionaries**

* + Dictionaries are used to store data values in key:value pairs.
  + A dictionary is a collection which is unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
  + It is changeable, means that we can change, add or remove items after the dictionary has been created.
  + It does not allow duplicates means cannot have two items with the same key.
  + Dictionaries are written with curly brackets, and have keys and values
  + It can be referred by using the key name.
  + The values in dictionary items can be of any data type.

Example-1: Create and print a dictionary.

dict1 = { "brand": "Ford",

"model": "Mustang", "year": 1964

}

print(dict1)

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Example-2: Print the "brand" value of the dictionary

dict1 = { "brand": "Ford",

"model": "Mustang", "year": 1964

Output: Ford

}

print(dict1["brand"])

Example -3: Duplicate values will overwrite existing values

dict1 = { "brand": "Ford",

"model": "Mustang", "year": 1964,

"year": 2020

}

print(dict1)

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}

Example – 4: String, int, boolean, and list data types

dict1 = { "brand": "Ford", "electric": False, "year": 1964,

"colors": ["red", "white", "blue"]

}

print(dict1)

Output: {'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}

**Python Dictionary Methods**: Python has a set of built-in methods that you can use on dictionaries.

# Method Description

clear() Removes all the elements from the dictionary

copy() Returns a copy of the dictionary

fromkeys() Returns a dictionary with the specified keys and value get() Returns the value of the specified key

items() Returns a list containing a tuple for each key value pair keys() Returns a list containing the dictionary's keys

pop() Removes the element with the specified key popitem() Removes the last inserted key-value pair

setdefault() Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

update() Updates the dictionary with the specified key-value pairs values() Returns a list of all the values in the dictionary

# Dictionary clear():

* + The clear() method removes all the elements from a dictionary.

Syntax: dictionary.clear()

Example: **Remove** all elements from the car list. car = {

"brand": "Ford",

"model": "Mustang",

"year": 1964

}

car.clear() print(car)

Output: {}

# Dictionary copy():

* + You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.
  + The copy() method returns a copy of the specified dictionary.

Syntax: dictionary.copy() Example: Copy the car dictionary.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.copy() print(x)

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

* + Another way to make a copy is to use the built-in function dict().

Example: Make a copy of a dictionary with the dict() function.

thisdict = { "brand": "Ford",

"model": "Mustang", "year": 1964

}

mydict = dict(thisdict) print(mydict)

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

# Dictionary fromkeys():

* + The fromkeys() method returns a dictionary with the specified keys and the specified value.

Syntax: dict.fromkeys(keys, value) Here,

keys Required. An iterable specifying the keys of the new dictionary

value Optional. The value for all keys. Default value is None Example: 1 Create a dictionary with 3 keys, all with the value 0.

x = ('key1', 'key2', 'key3') y = 0

thisdict = dict.fromkeys(x, y) print(thisdict)

Output: ['key1': 0, 'key2': 0, 'key3': 0]

Example: 2 Same example as above, but without specifying the value. x = ('key1', 'key2', 'key3')

thisdict = dict.fromkeys(x) print(thisdict)

Output: ['key1': None, 'key2': None, 'key3': None]

# Dictionary get():

* + The get() method returns the value of the item with the specified key.

Syntax: dictionary.get(keyname, value) Here,

keyname Required. The keyname of the item you want to return the value from

value Optional. A value to return if the specified key does not exist. Default value None.

Example-1: Get the value of the "model" item. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.get("model") print(x)

Output: Mustang

Example-2: Try to return the value of an item that do not exist. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.get("price", 15000) print(x)

Output: 15000

# Dictionary items():

* + The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.
  + The view object will reflect any changes done to the dictionary.

Syntax: dictionary.items()

Example-1: Return the dictionary's key-value pairs:

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.items() print(x)

Output: dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

Example-2: When an item in the dictionary changes value, the view object also gets updated.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.items() car["year"] = 2018 print(x)

Output: dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)])

# Dictionary keys():

* + The keys() method returns a view object. The view object contains the keys of the dictionary, as a list.
  + The view object will reflect any changes done to the dictionary.

Syntax: dictionary.keys()

Example – 1: Return the keys.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.keys()

print(x)

Output: dict\_keys(['brand', 'model', 'year'])

Example – 2: When an item is added in the dictionary, the view object also gets updated. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.keys() car["color"] = "white" print(x)

dict\_keys(['brand', 'model', 'year', 'color'])

Output:

# Dictionary pop():

* + The pop() method removes the specified item from the dictionary.
  + The value of the removed item is the return value of the pop() method.

Syntax: dictionary.pop(keyname, defaultvalue) Here,

keyname Required. The keyname of the item you want to remove

defaultvalue Optional. A value to return if the specified key do not exist. If this parameter is not specified, and the no item with the specified key is found, an error is raised.

Example: 1 **Remove** "model" from the dictionary.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

car.pop("model") print(car)

Output:

{'brand': 'Ford', 'year': 1964}

Example: 2 The value of the removed item is the return value of the pop() method. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.pop("model") print(x)

Output: Mustang

# Dictionary popitem():

* + The popitem() method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem() method removes a random item.
  + The removed item is the return value of the popitem() method, as a tuple.

Syntax: dictionary.popitem()

Example: 1 **Remove** the last item from the dictionary. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

car.popitem() print(car)

Output: {'brand': 'Ford', 'model': 'Mustang'}

Example: 2 The removed item is the return value of the pop() method. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.popitem() print(x)

Output: ('year', 1964)

# Dictionary setdefault():

* + The setdefault() method returns the value of the item with the specified key.
  + If the key does not exist, insert the key, with the specified value.

Syntax: dictionary.setdefault(keyname, value) Here,

keyname Required. The keyname of the item you want to return the value from

value Optional. If the key exist, this parameter has no effect.

If the key does not exist, this value becomes the key's value Default value None.

Example: 1 Get the value of the "model" item. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.setdefault("model", "Bronco") print(x)

Output: Mustang

Example: 2 Get the value of the "color" item, if the "color" item does not exist, insert "color" with the value "white"

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.setdefault("color", "white") print(x)

Output: White

# Dictionary update():

* + The update() method inserts the specified items to the dictionary.
  + The specified items can be a dictionary, or an iterable object with key value pairs.

Syntax: dictionary.update(iterable) Here,

iterable A dictionary or an iterable object with key value pairs, that will be inserted to the dictionary.

Example: Insert an item to the dictionary. car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

car.update({"color": "White"})

print(car)

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}

# Dictionary values():

* + The values() method returns a view object. The view object contains the values of the dictionary, as a list.
  + The view object will reflect any changes done to the dictionary, see example below.

Syntax: dictionary.values()

Example: 1 Return the values.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.values() print(x)

Output: dict\_values(['Ford', 'Mustang', 1964])

Example: 2 when a value is changed in the dictionary, the view object also gets updated.

car = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

x = car.values() car["year"] = 2018 print(x)

Output: dict\_values(['Ford', 'Mustang', 2018])

# del

* + The del keyword removes the item with the specified key name.

Example: 1

thisdict = { "brand": "Ford",

"model": "Mustang", "year": 1964

}

del thisdict["model"] print(thisdict)

Output: {'brand': 'Ford', 'year': 1964}

Example: 2 The del keyword can also delete the dictionary completely.

thisdict = { "brand": "Ford",

"model": "Mustang", "year": 1964

Output:

}

del thisdict

print(thisdict) #this will cause an error because "thisdict" no longer exists.

Traceback (most recent call last):

File "demo\_dictionary\_del3.py", line 7, in <module>

print(thisdict) #this will cause an error because "thisdict" no longer exists.

NameError: name 'thisdict' is not defined

# Check if Key Exists:

* + To determine if a specified key is present in a dictionary use the **in** keyword.

Example:1 Check if "model" is present in the dictionary.

thisdict = { "brand": "Ford",

"model": "Mustang", "year": 1964

}

if "model" in thisdict:

print("Yes, 'model' is one of the keys in the thisdict dictionary") Output: Yes, 'model' is one of the keys in the thisdict dictionary

# Dictionary Length:

* + To determine how many items a dictionary has, use the len() function: Example: Print the number of items in the dictionary.

thisdict = { "brand": "Ford",

"model": "Mustang", "year": 1964,

"year": 2020

}

print(len(thisdict))

Output: 3

# Loop Through a Dictionary:

* + You can loop through a dictionary by using a for loop.
  + When looping through a dictionary, the return values are the keys of the dictionary, but there are methods to return the values as well.

Example: 1 Print all key names in the dictionary, one by one. thisdict = {

"brand": "Ford",

"model": "Mustang", "year": 1964

}

Output: brand model year

for x in thisdict: print(x)

Example: 2 Print all values in the dictionary, one by one.

for x in thisdict: print(thisdict[x])

Output:

Ford Mustang 1964

Example: 3 You can also use the values() method to return values of a dictionary.

Output: Ford Mustang 1964

for x in thisdict.values(): print(x)

Example: 4 You can use the keys() method to return the keys of a dictionary.

Output: brand model year

for x in thisdict.keys(): print(x)

Example: 5 Loop through both keys and values, by using the items() method.

Output: brand Ford

for x, y in thisdict.items(): print(x, y)

model Mustang year 1964

# Nested Dictionaries:

* + A dictionary can contain dictionaries, this is called nested dictionaries.

Example: 1 Create a dictionary that contain three dictionaries.

myfamily = { "child1" : {

"name" : "Emil", "year" : 2004

},

"child2" : {

"name" : "Tobias" , "year" : 2007

},

"child3" : { "name" : "Linus", "year" : 2011

}

}

Output:

{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias',

'year': 2007},'child3': {'name': 'Linus', 'year': 2011}}

Example: 2 Create three dictionaries, then create one dictionary that will contain the other three dictionaries.

child1 = { "name" : "Emil", "year" : 2004

}

child2 = {

"name" : "Tobias", "year" : 2007

}

child3 = {

"name" : "Linus", "year" : 2011

}

myfamily = { "child1" : child1, "child2" : child2, "child3" : child3

}

Output:

{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias',

'year': 2007},'child3': {'name': 'Linus', 'year': 2011}}

### 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.

### 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.

### 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 maintenance 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

### 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.

### 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.

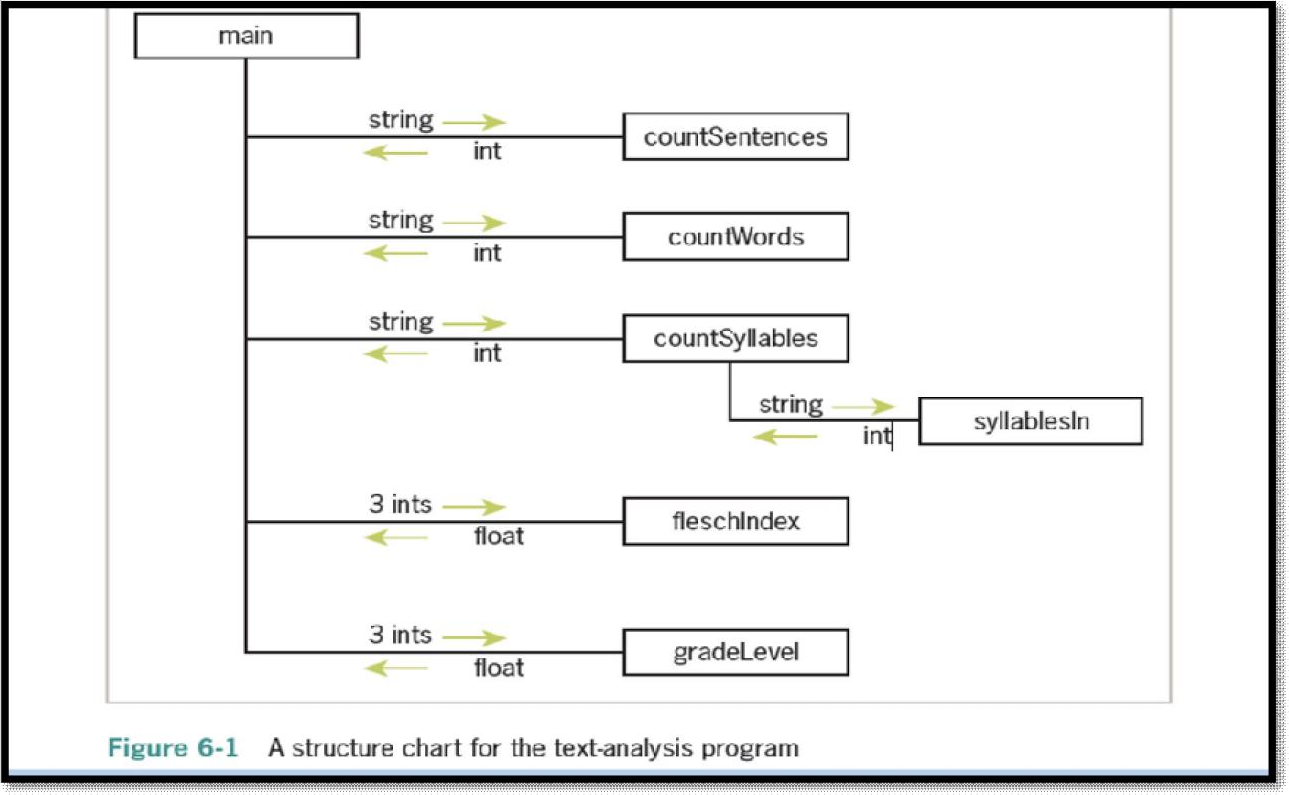
### Functions Support the Division of Labor

* + 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.

### . 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.**

### 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.
  + 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

### . 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)

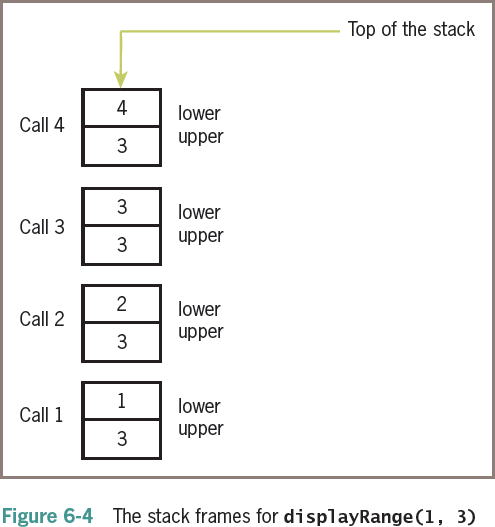
[Previous line repeated 989 more times] File "<pyshell#5>", line 2, in runForever if n > 0:

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.



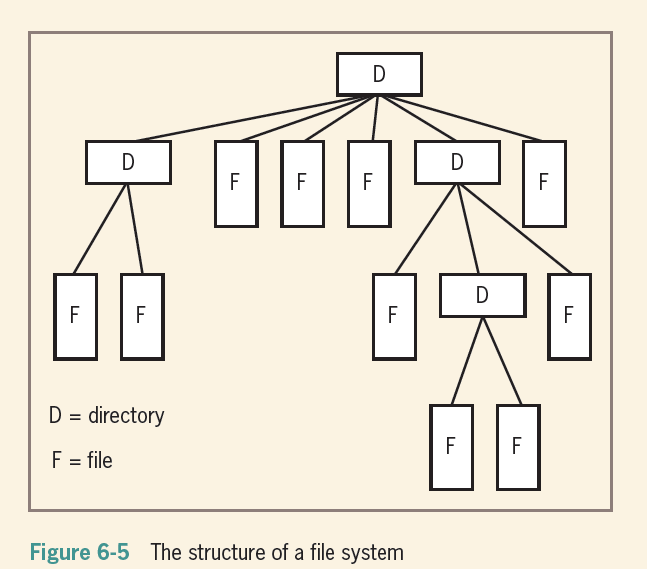
### Case Study Gathering Information from a File System

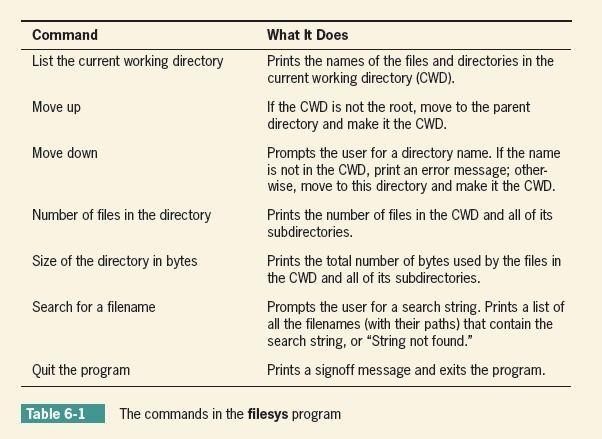
* + Modern file systems come with a graphical browser, allowing the user to navigate to files or folders by selecting icons of folders, opening these by double-clicking, and selecting commands from a drop-down menu. Information on a folder or a file, such as the size and contents, is also easily obtained 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:





import os, os.path QUIT = '7'

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

MENU = """1 List the current directory 2 Move up

1. Move down
2. 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() #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())

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.""" lyst = os.listdir(path)

# print(lyst)

if target in lyst: print('File found')

else:

print('File not found')

if name == " main ": main()

### 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

### 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.

### 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.

### 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.

–

### 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

* + 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]

### 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

### 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:

1. **import module\_name**

Example:

>>>import math

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

### 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))

1. "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))

### 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 :

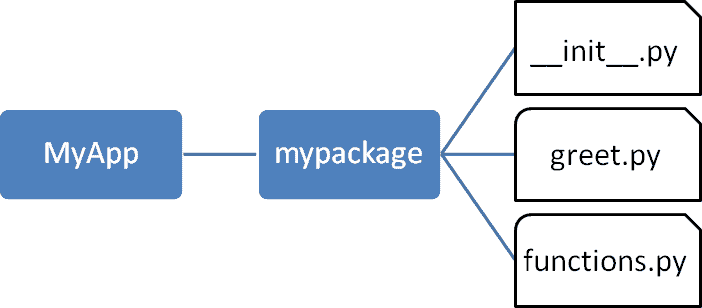
The package folder contains a special file called init .py, which stores the package's content. It serves two purposes:

* + 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