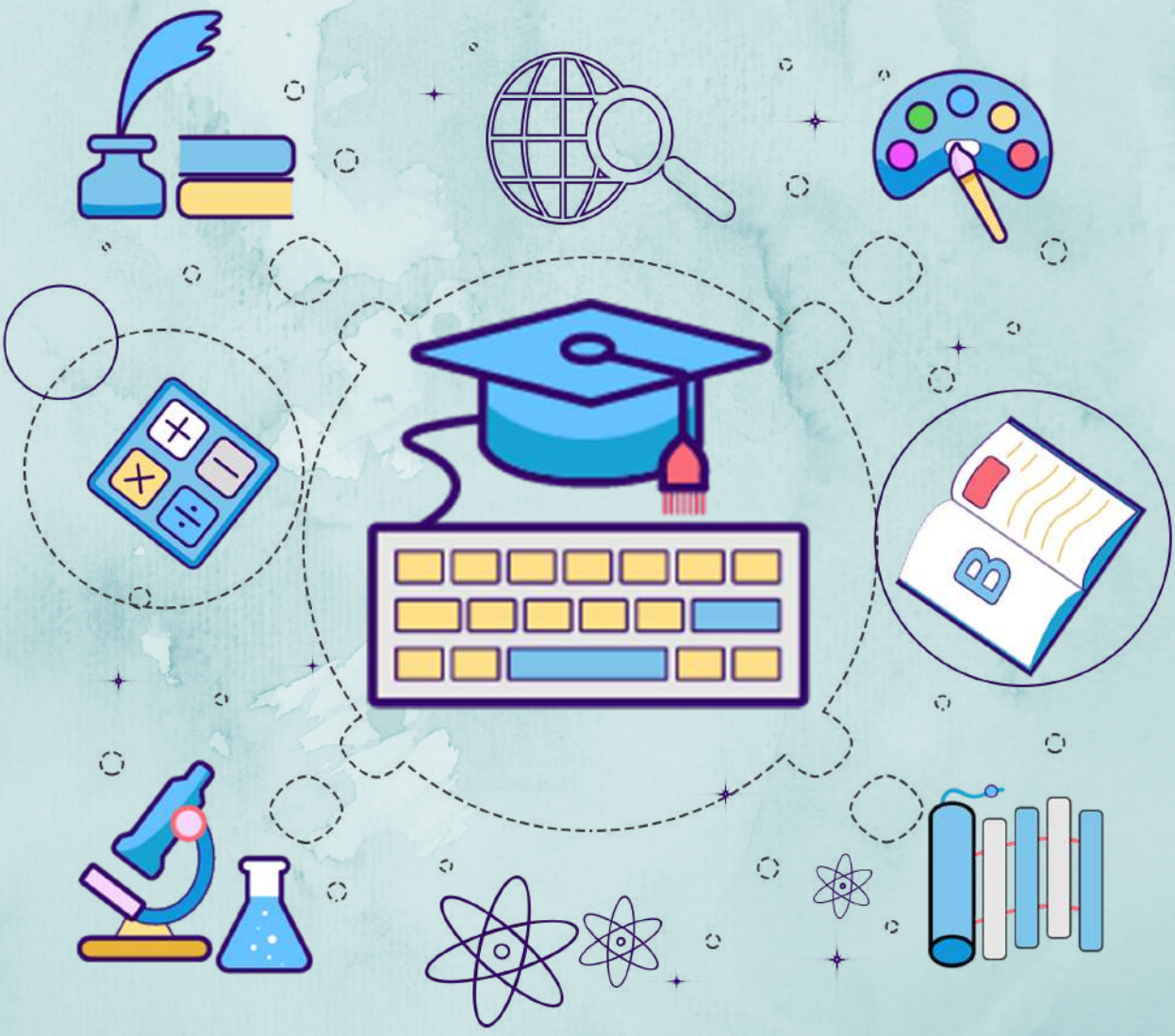
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

PROGRAMMING IN PYTHON

CST 362

Module 2

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE II

Strings and text files – Accessing characters, substrings, Data encryption, Strings and number system, String methods, Text files, A case study on text analysis. Design with Functions – Functions as Abstraction Mechanisms, Problem solving with top-down design, Design with recursive functions, Managing a program's namespace, Higher-Order Functions. Lists - Basic list Operations and functions, List of lists, Slicing, Searching and sorting list, List comprehension. Work with tuples. Sets. Work with dates and times, A case study with lists. Dictionaries - Dictionary functions, dictionary literals, adding and removing keys, accessing and replacing values, traversing dictionaries, reverse lookup. Case Study – Data Structure Selection.

Strings and text files

Accessing Characters and Substrings in Strings

The Structure of Strings

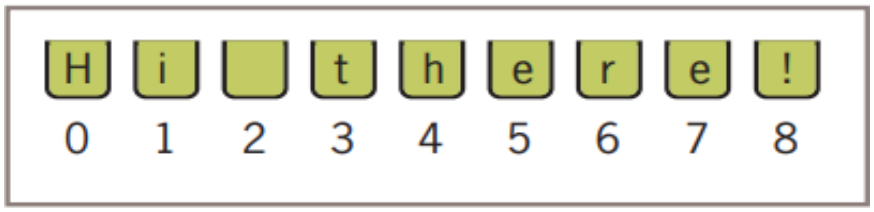
- Unlike an integer, which cannot be decomposed into more primitive parts, a string is a data structure. A data structure is a compound unit that consists of several other pieces of data.
- A string is a sequence of zero or more characters.
- We can mention a python string using either single quote marks or double quote marks. Here are some examples:

```
>>> "Hi there!"
'Hi there!'
>>> ""
''
 >>> 'R'
 'R'
```

- When working with strings, the programmer sometimes must be aware of a string's length and the positions of the individual characters within the string.
- A string's length is the number of characters it contains. Python's **len** function returns this value when it is passed a string.

```
>>> len("Hi there!")
 9
 >>> len("")
0
```

- The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right.
- The sequence of characters and their positions in the string "Hi there!"



- The string is an **immutable data structure**. This means that its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.

The Subscript Operator

- Although a simple for loop can access any of the characters in a string, sometimes you just want to inspect one character at a given position without visiting them all. The **subscript operator []** makes this possible.
- The simplest form of the subscript operation is the following:

<a string>[<an integer expression>]

- The first part of this operation is the string you want to inspect.
- The integer expression in brackets indicates the position of a particular character in that string. The integer expression is also called an **index.**
- In the following examples, the subscript operator is used to access characters in the string "Alan Turing":

```
>>> name = "Alan Turing"
>>> name[0]              # Examine the first character
'A'
>>> name[3]              # Examine the fourth character
'n'
>>> name[len(name)]      # Oops! An index error!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1]  # Examine the last character
'g'
>>> name[-1]             # Shorthand for the last character
'g'
>>> name[-2]             # Shorthand for next to last character
'n'
```

- The positions usually range from 0 to the length minus 1.

- Python allows negative subscript values to access characters at or near the end of a string. The programmer counts backward from –1 to access characters from the right end of the string.

Slicing for Substrings

- You can use Python's subscript operator to obtain a substring through a process called **slicing**.
- To extract a substring, the programmer places a **colon (:)** in the subscript. An integer value can appear on either side of the colon.
- Here are some examples that show how slicing is used:

```
>>> name = "myfile.txt"    # The entire string
>>> name[0:]
'myfile.txt'
>>> name[0:1]              # The first character
'm'
>>> name[0:2]              # The first two characters
'my'
>>> name[:len(name)]       # The entire string
'myfile.txt'
>>> name[-3:]              # The last three characters
'txt'
>>> name[2:6]              # Drill to extract 'file'
'file'
```

Data Encryption

- Data traveling on the Internet is vulnerable to spies and potential thieves. It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions.

- For this reason, most applications now use data encryption to protect information transmitted on networks.
- Encryption techniques are as old as the practice of sending and receiving messages. The sender **encrypts** a message by translating it to a secret code, called a **cipher text**.
- At the other end, the receiver **decrypts** the cipher text back to its original **plaintext** form.
- Both parties to this transaction must have at their disposal one or more keys that allow them to encrypt and decrypt messages.
- A very simple encryption method that has been in use for thousands of years is called a **Caesar cipher.**
- This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence. For positive distances, the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end.
- For example, if the distance value of a Caesar cipher equals three characters, the string **"invaders"** would be encrypted as **"lqydghuv"**.

A Caesar cipher with distance +3 for the lowercase alphabet

```
"""
File: encrypt.py
Encrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""

plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - \
                      (ord('z') - ordvalue + 1)
    code += chr(cipherValue)
print(code)


"""
File: decrypt.py
Decrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""

code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
    ordvalue = ord(ch)
    cipherValue = ordvalue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - \
                      (distance - (ord('a') - ordvalue - 1))
    plainText += chr(cipherValue)
print(plainText)
```

Strings and Number System

Number System

- Decimal number system: This system, also called the base ten number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits.

- Binary number system: the binary number system is used to represent all information in a digital computer. The two digits in this base two number system are 0 and 1.
- Octal number system: This system, also called the base eight number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, and 7 as digits.
- Hexadecimal number system: This system, also called the base sixteen number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F as digits.

To identify the system being used, you attach the base as a subscript to the number. The digits used in each system are counted from 0 to n - 1, where n is the system's base. Thus, the digits 8 and 9 do not appear in the octal system. To represent digits with values larger than 9_{10}, systems such as base 16 use letters. Thus, A_{16} represents the quantity 10_{10}, whereas 10_{16} represents the quantity 16_{10}.

For example, the following numbers represent the quantity 41510 in the binary, octal, decimal, and hexadecimal systems:

415 in binary notation	110011111_2
415 in octal notation	637_8
415 in decimal notation	415_{10}
415 in hexadecimal notation	$19F_{16}$

Handling numbers in various formats

1. **Converting Binary to Decimal**:

$1100111_2 =$

$1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 =$

$1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 =$

$64 + 32 + 4 + 2 + 1 = 103$

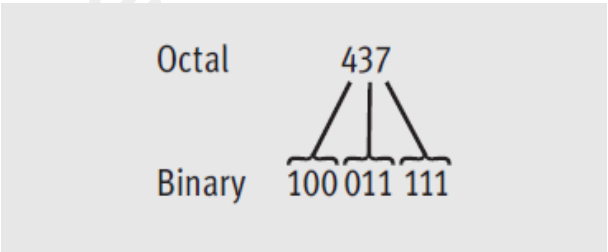
2. Converting Decimal to Binary:

Repeatedly divides the decimal number by 2. After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits. The quotient becomes the next dividend in the process. The string of bits is initially empty, and the process continues while the decimal number is greater than 0.

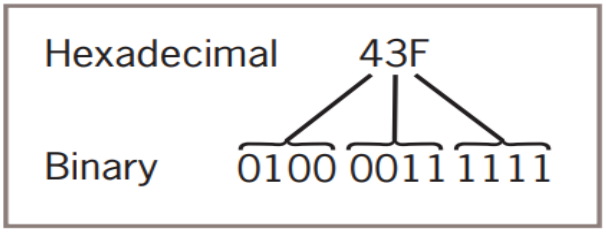
```
Enter a decimal integer: 34
Quotient Remainder Binary
   17        0           0
    8        1          10
    4        0         010
    2        0        0010
    1        0       00010
    0        1      100010
The binary representation is 100010
```

3. Octal and Hexadecimal Numbers:

a. **Conversion of octal to binary**:



b. Conversion of hexadecimal to binary:



String Methods/ String function

String Method	What it Does
`s.center(width)`	Returns a copy of **s** centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring **sub** in **s**. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.endswith(sub)`	Returns **True** if **s** ends with **sub** or **False** otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in **s** where substring **sub** is found. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.isalpha()`	Returns **True** if **s** contains only letters or **False** otherwise.
`s.isdigit()`	Returns **True** if **s** contains only digits or **False** otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is **s**.
`s.lower()`	Returns a copy of **s** converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of **s** with all occurrences of substring **old** replaced by **new**. If the optional argument **count** is given, only the first **count** occurrences are replaced.
`s.split([sep])`	Returns a list of the words in **s**, using **sep** as the delimiter string. If **sep** is not specified, any whitespace string is a separator.
`s. startswith(sub)`	Returns **True** if **s** starts with **sub** or **False** otherwise.
`s.strip([aString])`	Returns a copy of **s** with leading and trailing whitespace (tabs, spaces, newlines) removed. If **aString** is given, remove characters in **aString** instead.
`s.upper()`	Returns a copy of **s** converted to uppercase.

The following examples show some string methods in action:

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find("the")
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> " ".join(words)
'Hithere!'
>>> " ". join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
```

Text files

Programs that have taken input data from users at the keyboard as well as receive their input from text files. A **text file** is a software object that stores data on a permanent medium such as a disk, CD, or flash memory.

Writing Text to a File

- Data can be output to a text file using a file object. Python's open function, which expects a file name and a mode string as arguments, opens a connection to the file on disk and returns a file object.
- The mode string is '**r**' for input files and '**w**' for output files.
- Thus, the following code opens a file object on a file named myfile.txt for output:

  >>> f = open("myfile.txt", 'w')
- If the file does not exist, it is created with the given filename. If the file already exists, Python opens it. When an existing file is opened for output, any data already in it are erased.
- String data are written (or output) to a file using the method **write** with the file object. The write method expects a single string argument.

  >>> f.write("First line.\nSecond line.\n")
- When all of the outputs are finished, the file should be closed using the method close, as follows:

  >>> f.close()

Design with Functions

- A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.
- In Python, a function is a group of related statements that performs a specific task.

- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

```
def function_name(parameters):
    statement(s)
```

1. Keyword **def** that marks the start of the function header. How Function works in Python?
2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.
5. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
6. An optional return statement to return a value from the function

Calling a Function

- To call a function, use the function name followed by parenthesis:

Example:

```
def my_function():
    print("Hello from a function")
my_function() # calling a function
```

The return statement

- The return statement is used to exit a function and go back to the place from where it was called.

  Syntax of return:

  return [expression_list]

  This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.
- There are two types of functions in python
  - Built-in functions
  - User defined functions.
- There are many **built-in functions** provided by Python such as dir(), len(), abs(), etc.
- Users can also build their own functions, which are called **user-defined functions**.
- There are many advantages of using functions:
  - They reduce duplication of code in a program.
  - They break the large complex problems into small parts.
  - They help in improving the clarity of code.
  - A piece of code can be reused as many times as we want with the help of functions.

Functions as Abstraction Mechanism

- People deal complexity effectively 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.
- In this section, we examine the various ways in which functions serve as abstraction mechanisms in a program.

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):
    """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
```

- If the summation function didn't exist, the programmer would have to write the entire algorithm every time a summation is computed.
- In a program that must calculate multiple summations, the same code would appear multiple times. In other words, redundant code would be included in the program.
- By relying on a single function definition, instead of multiple instances of redundant code, the programmer frees herself to write only a single algorithm in just one place—say, in a library module.
- When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition.
- Another way that functions serve as abstraction mechanisms is by **hiding complicated details**.

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 subproblems-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**.

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):

    while lower<=upper:

        print(lower)

        lower=lower+1
```

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):
    if lower<=upper:
        print(lower)
        displayRange(lower+1,upper)
displayRange(4,10)
```

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.
- 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**.

```
>>> 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
```

Variable Scopes and Life time

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.
- Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
    x = 10
    print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

Output

```
Value inside function: 10
Value outside function: 20
```

Higher-Order Functions

- A function that is having another function as an argument or a function that returns another function as a return in the output is called the High order

function. High order functions operate with other functions given in the program.

Properties of High order functions in Python

1. In high order function, we can store a function inside a variable.
2. In high order function, a function can act as an instant of an object type.
3. We can return a function as a result of another function.
4. We can pass a function as a parameter or argument inside another function.
5. We can store Python high order functions in data structures format such as lists, hash tables, etc.

Mapping

The first type of useful higher-order function to consider is called a **mapping**. This process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results. Python includes a map function for this purpose.

```
>>> words = ["231", "20", "-45", "99"]
 >>> map(int, words)       # Convert all strings to int
>>> words                  # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words))    # Reset variable to change it
 >>> words
[231, 20, -45, 99]
```

Filtering

A second type of higher-order function is called a filtering. In this process, a function called a predicate is applied to each value in a list. If the predicate returns True, the value passes the test and is added to a filter object (similar to a map object). Otherwise, the value is dropped from consideration. The process is a bit like pouring hot water into a filter basket with coffee. The good stuff to drink comes into the cup with the water, and the coffee grounds left behind can be thrown on the garden.

```
>>> def odd(n):
        return n % 2 == 1

        list(filter(odd, range(10)))
```

OUTPUT:

```
[1, 3, 5, 7, 9]
```

As with the function map, the result of the function filter can be passed directly to another call of filter or map. List processing often consists of several mappings and filterings of data, which can be expressed as a series of nested function calls.

Reducing

Our final example of a higher-order function is called a reducing. Here we take a list of values and repeatedly apply a function to accumulate a single data value. A summation is a good example of this process. The first value is added to the second value, then the sum is added to the third value, and so on, until the sum of all the values is produced. 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
```

Lambda functions

- A lambda is an anonymous function. It has no name of its own, but contains the names of its arguments as well as a single expression.
- When the lambda is applied to its arguments, its expression is evaluated, and its value is returned.
- The syntax of a lambda is

  lambda <argname-1, ..., argname-n>: <expression>

- All of the code must appear on one line and a lambda cannot include a selection statement, because selection statements are not expressions.

- Example:

  z=lambda x,y:x+y

  print(z(2,3))

  output will be 5

  Example:

  def myfunc(n):

  return lambda a : a * n

  mydoubler = myfunc(2)

  mytripler = myfunc(3)

  print(mydoubler(11))

  print(mytripler(11))

Lists

- A list is a sequence of data values called **items or elements**. An item can be of any type. Each of the items in a list is ordered by position.
- 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.
- In Python, a list is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets (**[** and **]**).

```
[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.

```
[[5, 9], [541, 78]]
```

- When the element is a variable or any other expression, its value is included in the list.

```
>>> 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

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

- The function **len** and the subscript operator **[]** work just as they do for strings:

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

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

- Concatenation (+) and equality (==) also work as expected for lists:
- The **print** function strips the quotation marks from a string, but does not alter the look of a list:

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

- To print the contents of a list without the brackets and commas, you can use a **For** loop, as follows:

```
>>> for element in [1, 2, 3, 4]:
        print(element, end=" ")

1 2 3 4
>>>
```

- Finally, you can use the **in** operator to detect the presence or absence of a given element:

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

- Table summarizes these operators and functions, where **L** refers to a list.

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

Replacing an Element in a List

- The subscript operator is used to replace an element at a given position.

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

- Much of list processing involves replacing each element, with the result of applying some operation to that element.

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> index = 0
>>> while index < len(numbers):
        numbers[index] = numbers[index] ** 2
        index += 1

>>> numbers
[4, 9, 16, 25]
>>>
```

- This example replaces the first three elements of a list with new ones:

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

List Methods for Inserting and Removing Elements

LIST METHOD	WHAT IT DOES
`L.append(element)`	Adds **element** to the end of **L**.
`L.extend(aList)`	Adds the elements of **aList** to the end of **L**.
`L.insert(index, element)`	Inserts **element** at **index** if **index** is less than the length of **L**. Otherwise, inserts **element** at the end of **L**.
`L.pop()`	Removes and returns the element at the end of **L**.
`L.pop(index)`	Removes and returns the element at **index**.

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

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

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

Searching a List

- After elements have been added to a list, a program can search for a given element.
- The **in** operator determines an element‘s presence or absence, but programmers often are more interested in the position of an element if it is found (for replacement, removal, or other use). Unfortunately, the **list** type does not include the convenient **find** method that is used with strings.
- Instead of **find**, you must use the method **index** to locate an element‘s position in a list.
- It is unfortunate that **index** raises an error when the target element is not found.
- To guard against this unpleasant consequence, you must first use the **in** operator to test for presence and then the **index** method if this test returns **True**.

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

Sorting a List

- Although a list‘s elements are always ordered by position, it is possible to impose a natural ordering on them as well.
- In other words, you can arrange some elements in numeric or alphabetical order. A list of numbers in ascending order and a list of names in

alphabetical order are sorted lists.

- 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]
```

List comprehension

- List comprehensions are used for creating new lists from other iterables.
- As list comprehensions return lists, they consist of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.
- This is the basic syntax:

  new_list = [expression for_loop_one_or_more conditions]

- For example, find the squares of a number using the for loop: numbers=[1,2,3,4]

```
squares=[]
for n in numbers:
squares.append(n**2)
print(squares)
```

Output:

[1, 4, 9, 16]

*Finding squares using list comprehensions: numbers = [1, 2, 3, 4]

```
squares = [n**2 for n in numbers]
print(squares)
```

Output: [1, 4, 9, 16]

- Find common numbers from two lists using list comprehension:???????????

```
first = [1, 2, 3, 5]
second = [2, 4, 5, 7]
common = [x for x in first if x in second]
print(common)
```

Tuples

- A tuple is a type of sequence that resembles a list, except that, unlike a list, a tuple is **immutable**. You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets. The next session shows how to create several tuples:

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

- You can use most of the operators and functions used with lists in a similar fashion with tuples.

What are the benefit of tuple?

- Tuples are faster than lists.
- If you know that some data doesn't have to be changed, you should use tuples instead of lists, because this protects your data against accidental changes.
- Tuples can be used as keys in dictionaries, while lists can't.

Tuple creation

- It is defined within parentheses () where items are separated by commas.

>>> t = ("tuples", "are", "immutable")

- To create a tuple with a single element, we have to include the final comma:

>>> t1 = ('a',)

>>> type(t1)

<type 'tuple'>

- Without the comma, Python treats ('a') as a string in parentheses:

```
>>>t2=('a')
>>>type(t2)
<type 'str'>
```

Converting lists into tuples and tuples into lists

```
>>>L=[10,20,30,40]
>>>T=tuple(L)
>>>T (10,20,30,40)

>>> T=(10,20,30)
>>> L=list(T)
>>> L
[10, 20, 30]
```

Example Programs

? Program to read numbers and find minimum, maximum and sum usingTuple

```
n=int(input("Enter how many numbers .... "))
print("Enter numbers")

t=tuple()
for i in range(n):
    x=int(input())
    t=t+(x,)
```

```
print("minimum=",min(t))

print("maximum=",max(t))

print("sum=",sum(t))
```

Dictionaries

- Lists organize their elements by position.
- However, in some situations, the position of a datum in a structure is irrelevant; we're interested in its association with some other element in the structure.
- A dictionary organizes information by **association**, not position.
- For example, when you use a 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.
- In Python, a **dictionary** associates a set of **keys** with data values.
- 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:

A phone book: {'Savannah':'476-3321','Nathaniel':'351-7743'}

Personal information: {'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'}
>>>
```

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 error. Here are some examples, using the info dictionary, which was set up earlier:

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

- If the existence of a key is uncertain, the programmer can test for it using the dictionary method **get**. This method expects two arguments, a possible key and a default value. If the key is in the dictionary, the associated value is returned. However, if the key is absent, the default value passed to **get** is returned.

Here is an example of the use of **get** with a default value of **None**:

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

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.
- If pop is used with just one argument, and this key is absent from the dictionary, Python raises an error.

- The next session attempts to remove two keys and prints the values 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 a list of 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"}
>>> grades.items()
[(80, 'B'), (90, 'A'), (70, 'C')]
```

- Note that 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 grades.items():
   print(key, value)
```

- If a special ordering of the keys is needed, you can obtain a list of keys using the **keys** method and process this list to rearrange the keys.
- For example, you can sort the list and then traverse it to print the entries of the dictionary in alphabetical order:

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

- Table below summarizes the commonly used dictionary operations, where **d** refers to a dictionary.

DICTIONARY OPERATION	WHAT IT DOES
`len(d)`	Returns the number of entries in **d**.
`aDict[key]`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
`d.get(key [, default])`	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.pop(key [, default])`	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`list(d.keys())`	Returns a list of the keys.
`list(d.values())`	Returns a list of the values.
`list(d.items())`	Returns a list of tuples containing the keys and values for each entry.
`d.clear()`	Removes all the keys.
`for key in d:`	**key** is bound to each key in **d** in an unspecified order.

Set

- A set is a collection which is unordered and un-indexed. In Python, sets are written with curly brackets.

Create a Set:

```
thisset = {"apple", "banana", "cherry"}

print(thisset)
```

- Sets are unordered, so you cannot be sure in which order the items will appear.
- **Access Items**: You cannot access items in a set by referring to an index or akey.
- But you can loop through the set items using a **for loop**, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:

    print(x)
```

- Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

- **Add Items:** To add one item to a set use the add() method. To add more than one item to a set use the update() method.

```
#using the add() method
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)

#using the update() method

thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

Get the Length of a Set: To determine how many items a set has, use the len() method.

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Remove Item: To remove an item in a set, use the remove(), or the discard() method. If the item to remove does not exist, remove() will raise an error. If the item to remove does not exist, discard() will NOT raise an error. You can also use the pop(), method to remove an item, but this method will remove the last item. The return value of the pop() method is the removed item.

#using the **remove()** method

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

#using the **discard()** method

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

#using the **pop()** method

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Python Sets Methods	
Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Datetime

- **Dates:** A date in Python is not a data type of its own, but we can import a module named **datetime** to work with dates as date objects.
- Import the datetime module and display the current date:

```
import datetime
x = datetime.datetime.now()
print(x)
```

- **Creating Date Objects:** To create a date, we can use the datetime() class (constructor) of the datetime module. The datetime() class requires three parameters to create a date: year, month, day.

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

- The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

- **The strftime() Method**: The datetime object has a method for formatting date objects into readable strings. The method is called strftime(), and takes one parameter,**format,** to specify the format of the returned string:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

  - A reference of all the legal format codes:

Format	Meaning	Example
%a	abbreviated weekday name	Mon
%A	full weekday name	Monday
%b	abbreviated month name	Dec
%B	full month name	December
%c	the locale's date and time	Sun Oct 22 00:00:00 2019
%d	zero-padded day of the month	[01,31]
%e	number [1,31]; equivalent to %_d.	[1,31]
%H	hour in 24-hour clock	[00,23]
%I	hour in 12-hour clock	[01,12]
%j	day of the year	[001,366]
%m	month as a number	[01,12]
%M	minute as a number	[00,59]
%S	seconds as a number	[000, 999]
%p	either AM or PM	PM
%U	Sunday-based week of the year	[00,53]
%w	Sunday-based weekday	[0,6]
%W	Monday-based week of the year	[00,53]
%x	the locale's date as %-m/%-d/%Y	07/17/2019
%X	the locale's time as %-I:%M:%S %p	11:24:45
%y	year without century	[00,99]
%Y	year with century	2019
%Z	time zone offset	-0700, -07:00, -07, or Z
%%	a literal percent sign	%