MODULE II

(Building Python Programs)

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.

The Structure of Strings

- A string is a sequence of zero or more characters.
- String using either single quote marks or double quote marks.
- The positions of a string's characters are numbered from 0, to the length of the string minus 1.
-

The sequence of characters and their positions in the string "Hi there!" .

The Structure of Strings

- 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 [] : To read character at a given position. `<a string>[<an integer expression>]`
- The integer expression is also called an index .

The Structure of Strings

```
>>> 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 Structure of Strings

For loop can be used to iterate through the string.

```
>>> data = "Hi there!"
>>> for index in range(len(data)):
        print(index, data[index])
```

Python's len function returns length of the string.

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

The Structure of Strings : Slicing for Substrings

- Some applications extract portions of strings called substrings.
- Python's subscript operator helps 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.

The Structure of Strings : Slicing for Substrings

- when two integer positions are included in the slice, the range of characters in the substring extends from the first position up to but not including the second position.

```
name = "myfile.txt"
name[2:6]
```

```
'file'
```

- When the integer is omitted on either side of the colon, all of the characters extending to the end or the beginning are included in the substring.

```
>>> name[0:]
'myfile.txt'
```

The Structure of Strings : Slicing for Substrings

```
>>> 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'
```

The Structure of Strings : Slicing for Substrings : Testing for a Substring with the in Operator

➔ Print the filenames have a .txt extension in a given list.

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
>>> for fileName in fileList:
        if ".txt" in fileName:
            print(fileName)
```

```
myfile.txt
yourfile.txt
```

Sample Questions

- Write a Python code to reverse a string.
- Write a Python code to determine whether the given string is a Palindrome or not using slicing. Do not use any string function.
- Write a Python code to remove i’th character from string.

Data Encryption

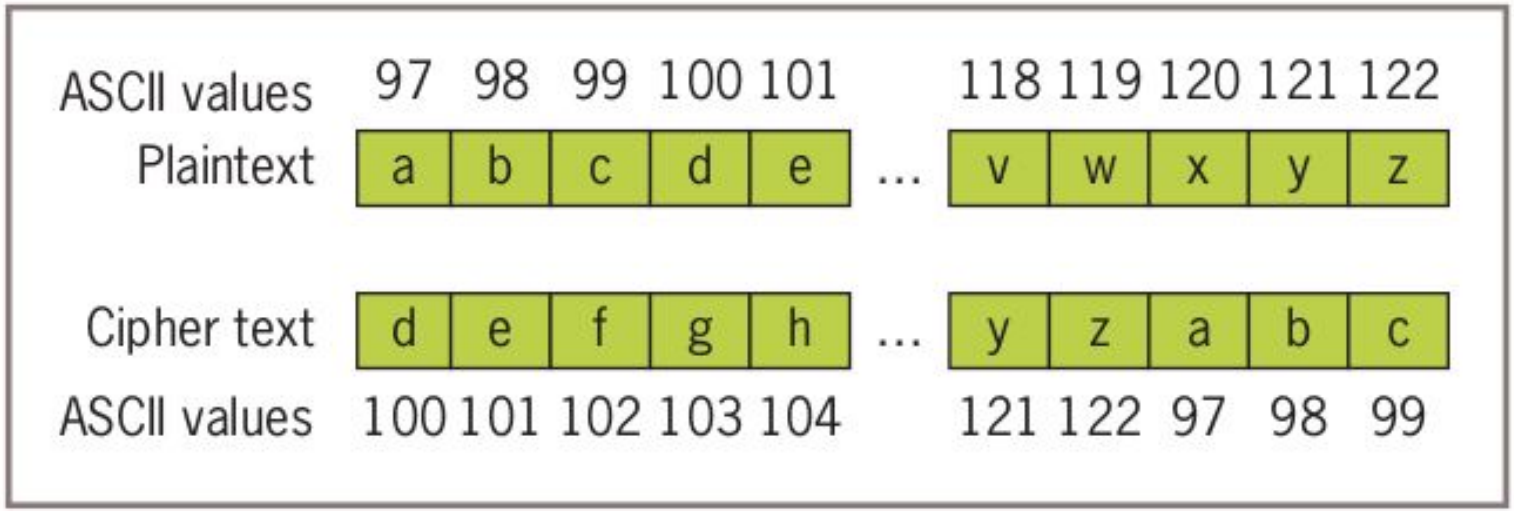
- Data encryption translates data into another form, or code
- so that only people with access to a secret key (formally called a decryption key) or password can read it.
- Encrypted data is commonly referred to as ciphertext, while unencrypted data is called plaintext.

Caesar Cipher

- The Caesar Cipher technique is one of the earliest and simplest method of encryption technique.
- Each letter of a given text is replaced by a letter some fixed number of positions after the alphabet.
- For example with a shift of 1, A would be replaced by B

Caesar Cipher

Caesar cipher with distance 3 for the lowercase alphabet



Caesar Cipher Python implementation

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

Caesar Cipher Python implementation

```
# Decryption
cipherText = code
plainText = ""
for ch in cipherText:
        ascivalue = ord(ch)
        cipherValue = ascivalue - distance
        if cipherValue < ord('a'):
                cipherValue = ord('z') - ((ord('a') - cipherValue - 1))
        plainText += chr(cipherValue)
print(plainText)
```

Strings and Number Systems

- Testing for a Substring with the in Operator:
  - 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.

Strings and Number Systems

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
>>> for fileName in fileList:
        if ".txt" in fileName:
            print(fileName)
myfile.txt
yourfile.txt
```

String Methods

```
>>> sentence = input("Enter a sentence: ")
Enter a sentence: This sentence has no long words.
>>> listOfWords = sentence.split()
>>> print("There are", len(listOfWords), "words.")
There are 6 words.
>>> sum = 0
>>> for word in listOfWords:
        sum += len(word)
>>> print("The average word length is", sum / len(listOfWords))
The average word length is 4.5
```

String Methods

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.

String Methods

`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.

String Methods

```
>>> 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
```

String Methods

```
>>> 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!'
```

String Methods

```
>>> "myfile.txt".split('.')
['myfile', 'txt']
>>> "myfile.py".split('.')
['myfile', 'py']
>>> "myfile.html".split('.')
['myfile', 'html']
```

Number conversions (self study)

Mutual conversions of

- Binary
- Hexadecimal
- Octal
- Decimal

Text files

- A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory.
- advantages of taking input data from a file are the following:
  - The data set can be much larger.
  - The data can be input much more quickly and with less chance of error.
  - The data can be used repeatedly with the same program or with different programs.

Text Files and Their Format

- Using a text editor such as Notepad or TextEdit, you can create, view, and save data in a text file
- The data in a text file can be viewed as characters, words, numbers, or lines of text, depending on the text file's format.
- If the data are numbers (either integers or floats), they must be separated by whitespace character spaces, tabs, and newlines—in the file.
- All data output to or input from a text file must be strings.

Writing Text to a File

- Data can be output to a text file using a file object.
- 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.
- `>>> 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.

Writing Text to a File

- 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.
- If you want the output a newline after text, you must include the escape character '\n' in the string.
- The next statement writes two lines of text to the file:
- ```
  >>> f.write("First line.\nSecond line.\n")
  ```
- file should be closed using the method close after finishing all operations.
- ```
  >>> f.close()
  ```

Writing Numbers to a File

- method write expects a string as an argument. Therefore, other types of data, such as integers or floating-point numbers, must first be converted to strings before being written to an output file.
-

Writing Numbers to a File

Random integers between 1 and 500 are generated and written to a text file named integers.txt.[use newline as separator]

```
import random
f = open("integers.txt", 'w')
for count in range(500):
        number = random.randint(1, 500)
        f.write(str(number) + '\n')
f.close()
```

Reading Text from a File

```
>>> f = open("myfile.txt", 'r')
>>> text = f.read()
>>> text
'First line.\nSecond line.\n'
>>> print(text)
First line.
Second line.
```

- ➔ method read reads the entire contents of the file as a single string.
- ➔ After input is finished, another call to read would return an empty string, to indicate that the end of the file has been reached.

Reading Text from a File

- The for loop views a file object as a sequence of lines of text.
- On each pass through the loop, the loop variable is bound to the next line of text in the sequence.
-
```
>>> f = open("myfile.txt", 'r')
>>> for line in f:
        print(line)
First line.

Second line.
```

Reading Text from a File

- The readline method consumes a line of input and returns this string, including the newline.
- If readline encounters the end of the file, it returns the empty string.

```
>>> f = open("myfile.txt", 'r')
>>> while True:
        line = f.readline()
        if line == "":
            break
        print(line)
First line.

Second line.
```

Reading Numbers from a File

Al read methods will returns string, so it has to be converted to desired format.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    line = line.strip()
    number = int(line)
    theSum += number
print("The sum is", theSum)
```

Reading Numbers from a File

to handle integers separated by spaces and/or newlines.

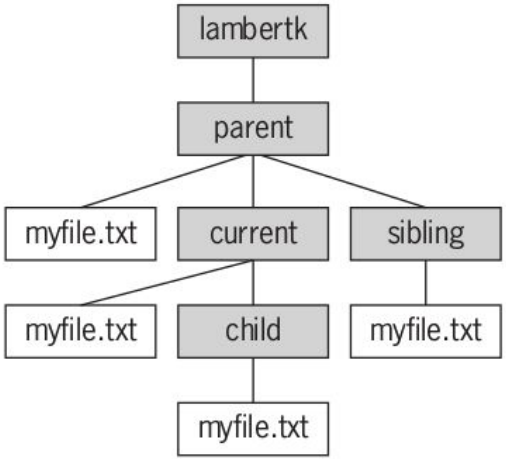
```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        theSum += number
print("The sum is", theSum)
```

File methord - Summary

Method	What it Does
`open(filename, mode)`	Opens a file at the given filename and returns a file object. The **mode** can be `'r'`, `'w'`, `'rw'`, or `'a'`. The last two values, `'rw'` and `'a'`, mean read/write and append, respectively.
`f.close()`	Closes an output file. Not needed for input files.
`f.write(aString)`	Outputs **aString** to a file.
`f.read()`	Inputs the contents of a file and returns them as a single string. Returns "" if the end of file is reached.
`f.readline()`	Inputs a line of text and returns it as a string, including the newline. Returns "" if the end of file is reached.

Accessing and Manipulating Files and Directories on Disk

- The complete set of directories and files forms a tree-like structure, with a single root directory at the top and branches down to nested files and subdirectories.



/Users/lambertk/parent/current/child/myfile.txt

Accessing and Manipulating Files and Directories on Disk

➔ Using absolute path names

```
f = open("/Users/lambertk/parent/current/child/myfile.txt", 'r')
```

➔ Using relative path names

Pathname	Target Directory
myfile.txt	current
child/myfile.txt	child
../myfile.txt	parent
../sibling/myfile.txt	sibling

Accessing and Manipulating Files and Directories on Disk

print all of the names of files in the current working directory that have a .py extension:

```
import os
currentDirectoryPath = os.getcwd()
listOfFileNames = os.listdir(currentDirectoryPath)
for name in listofFileNames:
    if ".py" in name:
        print(name)
```

Accessing and Manipulating Files and Directories on Disk - File system functions

os Module Function	What it Does
`chdir(path)`	Changes the current working directory to **path**.
`getcwd()`	Returns the path of the current working directory.
`listdir(path)`	Returns a list of the names in directory named **path**.
`mkdir(path)`	Creates a new directory named **path** and places it in the current working directory.
`remove(path)`	Removes the file named **path** from the current working directory.
`rename(old, new)`	Renames the file or directory named **old** to **new**.
`rmdir(path)`	Removes the directory named **path** from the current working directory.
`sep`	A variable that holds the separator character (`'/'` or `'\'`) of the current file system.

Accessing and Manipulating Files and Directories on Disk - File system functions

os.path Module Function	What it Does
`exists(path)`	Returns **`True`** if **`path`** exists and **`False`** otherwise.
`isdir(path)`	Returns **`True`** if **`path`** names a directory and **`False`** otherwise.
`isfile(path)`	Returns **`True`** if **`path`** names a file and **`False`** otherwise.
`getsize(path)`	Returns the size of the object names by **`path`** in bytes.
`normcase(path)`	Converts path to a pathname appropriate for the current file system; for example, converts forward slashes to backslashes and letters to lowercase on a Windows system.

Case Study: Text Analysis

develop a program that computes the Flesch Index for

a text file.

This index is based on the average number of syllables per word and the average number of words per sentence in a piece of text. Index scores usually range from 0 to 100, and they indicate readable prose for the following grade levels:

Flesch Index	Grade Level of Readability
0–30	College
50–60	High School
90–100	Fourth Grade

Request

Write a program that computes the Flesch Index and grade level for text stored in a text file.

Analysis

The input to this program is the name of a text file. The outputs are the number of sentences, words, and syllables in the file, as well as the file's Flesch Index and Grade Level Equivalent.

Word	Any sequence of non-whitespace characters.
Sentence	Any sequence of words ending in a period, question mark, exclamation point, colon, or semicolon.
Syllable	Any word of three characters or less; or any vowel (a, e, i, o, u) or pair of consecutive vowels, except for a final *-es*, *-ed*, or *-e* that is not *-le*.

Flesch's formula to calculate the index *F* is the following:

$$F = 206.835 - 1.015 \times (words / sentences) - 84.6 \times (syllables / words)$$

The Flesch-Kincaid Grade Level Formula is used to compute the Equivalent Grade Level *G*:

$$G = 0.39 \times (words / sentences) + 11.8 \times (syllables / words) - 15.59$$

Design

This program will perform the following tasks:

1. Receive the filename from the user, open the file for input, and input the text.
2. Count the sentences in the text.
3. Count the words in the text.
4. Count the syllables in the text.
5. Compute the Flesch Index.
6. Compute the Grade Level Equivalent.
7. Print these two values with the appropriate labels, as well as the counts from tasks 2–4.

The tasks defined

Task	What it Does
count the sentences	Counts the number of sentences in **text**.
count the words	Counts the number of words in **text**.
count the syllables	Counts the number of syllables in **text**.
compute the Flesch Index	Computes the Flesch Index for the given numbers of sentences, words, and syllables.
compute the grade level	Computes the Grade Level Equivalent for the given numbers of sentences, words, and syllables.

Implementation (Coding)

```
"""
Program: textanalysis.py
Author: Ken
Computes and displays the Flesch Index and the Grade
Level Equivalent for the readability of a text file.
"""

# Take the inputs
fileName = input("Enter the file name: ")
inputFile = open(fileName, 'r')
text = inputFile.read()
```

```
# Count the sentences
sentences = text.count('.') + text.count('?') + \
            text.count(':') + text.count(';') + \
            text.count('!')

# Count the words
words = len(text.split())

 # Count the syllables
 syllables = 0
 vowels = "aeiouAEIOU"
 for word in text.split():
     for vowel in vowels:
         syllables += word.count(vowel)
     for ending in ['es', 'ed', 'e']:
         if word.endswith(ending):
             syllables -= 1
     if word.endswith('le'):
         syllables += 1
```

```
# Compute the Flesch Index and Grade Level
index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
level = round(0.39 * (words / sentences) + 11.8 * \
        (syllables / words) - 15.59)

# Output the results
print("The Flesch Index is", index)
print("The Grade Level Equivalent is", level)
print(sentences, "sentences")
print(words, "words")
print(syllables, "syllables")
```

Testing

bottom-up testing .

Each task is coded and tested before it is integrated into the overall program. After you have written code for one or two tasks, you can test them in a short script. This script is called a driver .

```
"""
Program: fleschdriver.py
Author: Ken
Test driver for Flesch Index and Grade level.
"""

sentences = int(input("Sentences: "))
words = int(input("Words: "))
syllables = int(input("Syllables: "))
index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
print("Flesch Index:", index)
level = round(0.39 * (words / sentences) + 11.8 * \
              (syllables / words) - 15.59)
print("Grade Level: ", level)
```

Sample program questions

- Write a code segment that opens a file named myfile.txt for input and prints the number of lines in the file.
- Python code to count the lines, word, and characters within a text file.

The programmer can supply a text file that contains the number of sentences, words, and syllables already tested in the driver, and then compare the two test results.

bottom-up testing

the lower-level tasks must be developed and tested before those tasks that depend on the lower-level tasks.

When you have tested all of the parts, you can integrate them into the complete program.

The test data at that point should be short files that produce the expected results. Then, use longer

Design with Functions

- 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.

Design with Functions : Functions as Abstraction Mechanisms

An abstraction hides detail and thus allows a person to view a many things as just one thing.

- Functions Eliminate Redundancy : The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code.

Design with Functions : Functions as Abstraction Mechanisms

➔ Example : Functions Eliminate Redundancy

```
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
```

Design with Functions : Functions as Abstraction Mechanisms

- Functions Hide Complexity : Another way that functions serve as abstraction mechanisms is by hiding complicated details.
- 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 are the data sent as arguments to the function.

Design with Functions : Functions as Abstraction Mechanisms

Functions Support the Division of Labor : functions can enforce a division of labor. Ideally, each function performs a single coherent task,

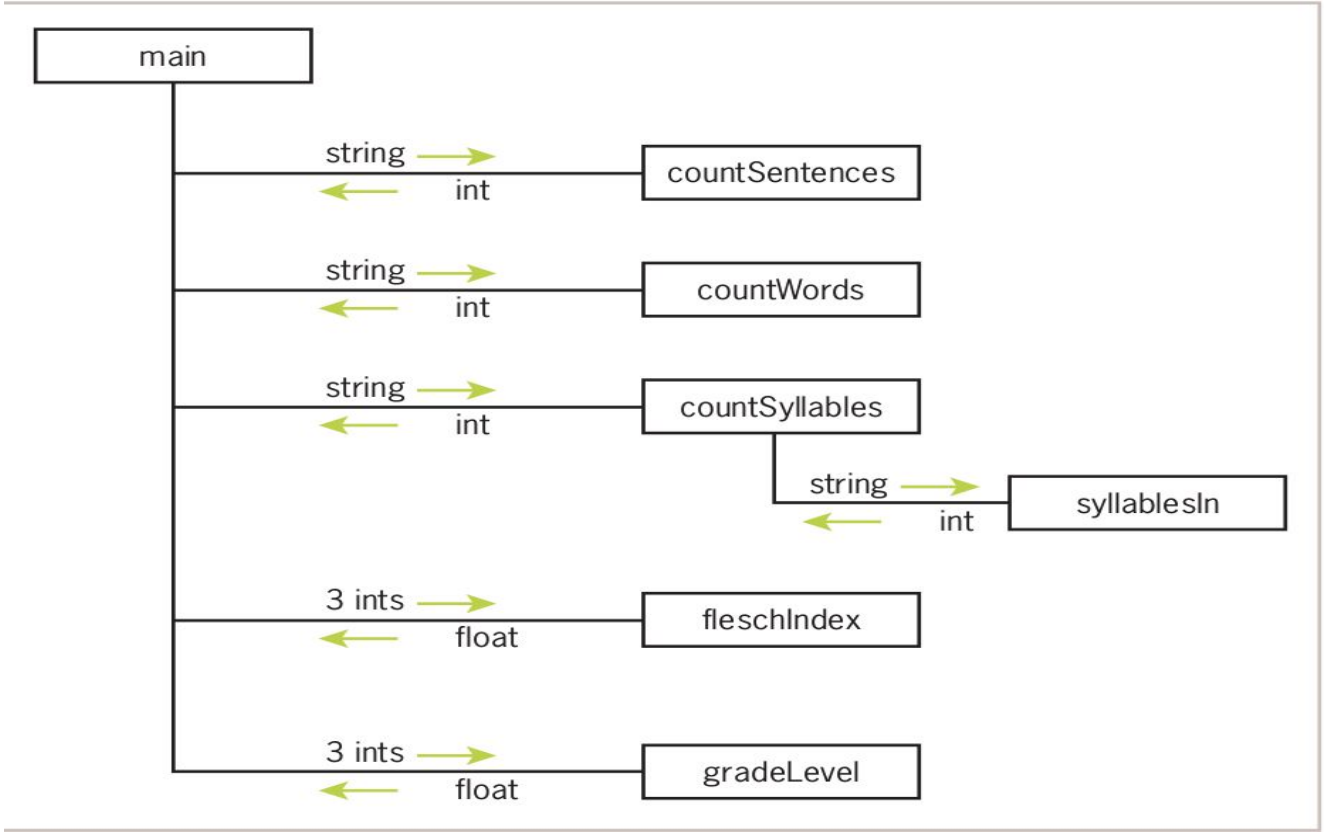
Problem Solving with Top-Down Design

- This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as problem decomposition .
- 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

- develop a new function for counts of the sentences, words, and syllables and calculating the readability scores.
- 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
  - The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them.

The Design of the Text-Analysis Program



Design with Recursive Functions

- In some cases, 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 .

Eg : displayRange that prints the numbers from a lower bound to an upper bound:

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

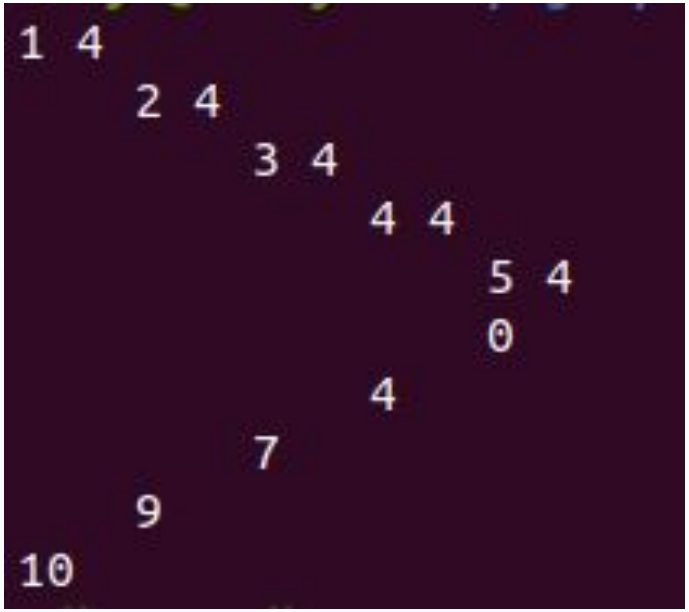
Recursive implementation

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

Recursive Function: with return

```
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)
```

Tracing a Recursive Function




```
def summation(lower, upper, margin):
        blanks = " " * margin
        print(blanks, lower, upper)
        if lower > upper:
                print(blanks, 0)
                return 0
        else:
                result = lower + summation(lower + 1, upper,margin + 4)
                print(blanks, result)
                return result
summation (1, 4, 0)
```

Using Recursive Definitions to Construct Recursive Functions

- Recursive functions are frequently used to design algorithms for computing values that have a recursive definition .
- recursive definition of the nth Fibonacci number is:
  -

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

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Recursion in Sentence Structure

```
def nounPhrase():
    """Returns a noun phrase, which is an article followed
    by a noun, and an optional prepositional phrase."""
    phrase = random.choice(articles) + " " + random.choice(nouns)
    prob = random.randint(1, 4)
    if prob == 1:
        return phrase + " " + prepositionalPhrase()
    else:
        return phrase

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

Infinite Recursion

when the function can (theoretically) continue executing forever, a situation known as infinite recursion .

Infinite recursion arises when the programmer fails to specify the base case

```
>>> def runForever(n):
        if n > 0:
            runForever(n)
        else:
            runForever(n - 1)

>>> runForever(1)
```

The Costs and Benefits of Recursion

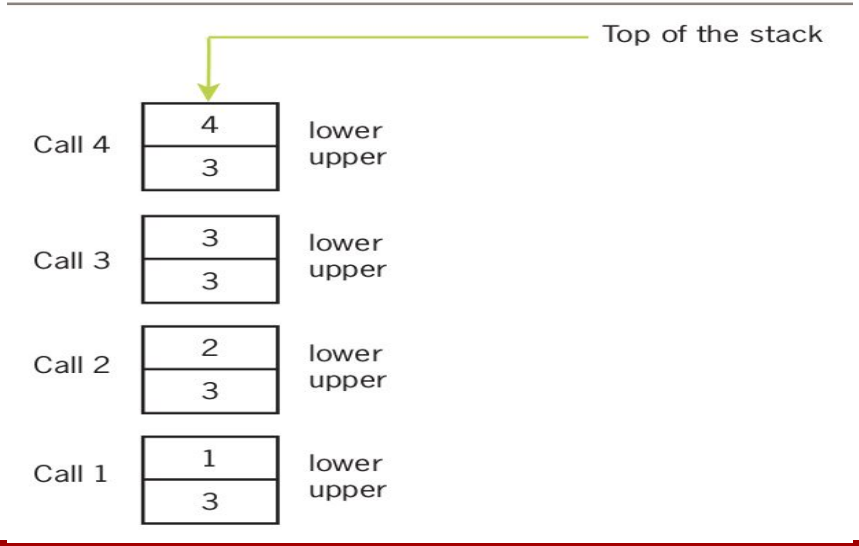
the PVM, must devote some overhead to recursive function calls. At program startup,

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 .

The stack frames for the process generated by displayRange(1, 3)



Sample program questions

- Write a Python function to calculate the factorial of a Number.
- Write a Python function that takes a number as a parameter and check the number is prime or not.
- Write a Python function that checks whether a passed string is palindrome or not.
- Write a Python function to find the Max of three numbers.

Managing a Program's Namespace

- How a program's namespace that is, the set of its variables and their values is structured and how you can control it via good design principles.
-

Module Variables, Parameters, and Temporary Variables

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

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

Module Variables, Parameters, and Temporary Variables

- This code appears in the file doctor.py, so its module name is doctor .
  - Module variables : The names replacements and changePerson are introduced at the level of the module.
  - Parameters: The name sentence is a parameter of the function changePerson .The parameter does not receive a value until the function is called.
  - Temporary variables:The names words , replyWords , and word are introduced in the body of the function changePerson .
  - Method names:The names split and join are introduced or defined in the str type.

Scope :

In Python, a name's scope is the area of program text in which the name refers to a given value.

- ➔ The scope of the temporary variables are restricted to the body of the functions
- ➔ The scope of the function parameter is the entire body of the function
- ➔ The scope of the module variables is in the entire module

Scope :

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

once the temporary variable is introduced, the module variable is no longer visible within function f .

Lifetime

- Scope of a variable that determines their visibility.
- A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it.
- Module variables come into existence when they are introduced via assignment
  - exist for the lifetime of the program that introduces or imports those module variables.
- Parameters and temporary variables come into existence when they are bound to values during a function call but go out of existence when the function call terminates.

Using Keywords for Default and Optional Arguments

- The programmer can also specify optional arguments with default values in any function definition.

```
def <function name>(<required arguments>,
                    <key-1> = <val-1>, ... <key-n> = <val-n>)
```

- The required arguments are listed first in the function header.

Example: repToInt , function to convert string representations of numbers in a given base to their integer values

```
def repToInt(repString, base):
    """Converts the repString to an int in the base
    and returns this int."""
    decimal = 0
    exponent = len(repString) - 1
    for digit in repString:
        decimal = decimal + int(digit) * base ** exponent
        exponent -= 1
    return decimal
```

Alter the definition to

```
def repToInt(repString, base = 2):
```

```
>>> repToInt("10", 10)
10
>>> repToInt("10", 8) # Override the default to here
8
>>> repToInt("10", 2) # Same as the default, not necessary
2
>>> repToInt("10")    # Base 2 by default
2
```

Using Keywords for Default and Optional Arguments

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.

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.

Using Keywords for Default and Optional Arguments

```
>>> 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
```

Higher-Order Functions

- A higher-order function expects a function and a set of data values as arguments.
- The argument function is applied to each data value, and a set of results or a single data value is returned.

Functions as First-Class Data Objects

➔ Function can be assigned to variables , passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries.

```
>>> abs                          # See what abs looks like
<built-in function abs>
>>> import math
>>> math.sqrt
<built-in function sqrt>
>>> f = abs                      # f is an alias for abs
>>> f                            # Evaluate f
<built-in function abs>
>>> f(-4)                        # Apply f to an argument
4
>>> funcs = [abs, math.sqrt]     # Put the functions in a list
>>> funcs
[<built-in function abs>,   <built-in function sqrt>]
>>> funcs[1](2)                  # Apply math.sqrt to 2
1.4142135623730951
```

Functions as First-Class Data Objects

```
>>> def example(functionArg, dataArg):
        return functionArg(dataArg)
>>> example(abs, -4)
4
>>> example(math.sqrt, 2)
1.4142135623730951
```

- ➔ The function argument is first evaluated, producing the function itself, and then the parameter name is bound to this value.
- ➔ The function can then be applied to its own argument with the usual syntax.

Mapping

- ➔ This process applies a function to each value in a sequence
- ➔ map builds and returns a new map object, which we feed to the list function to view the results.

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)          # Convert all strings to ints
<map object at 0x14cbd90>
>>> 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 .
- Otherwise, the value is dropped from consideration.

```
>>> def odd(n): return n % 2 == 1
>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
```

Reducing

- ➔ take a list of values and repeatedly apply a function to accumulate a single data value.
- ➔ Python functools module includes a reduce function

```
>>> 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
```

Using lambda to Create Anonymous Functions

- ➔ lambda allows the programmer to create anonymous functions .
- ➔ Lambda functions can have any number of arguments but only one 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>
```

- ➔ lambda cannot include a selection statement, because selection statements are not expressions.

Lists

- A list allows the programmer to manipulate a sequence of data values of any types.
- 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
- real-world examples of lists:
  - A shopping list for the grocery store
  - A to-do list
  - A guest list for a wedding

List Literals and Basic Operators

- list literal 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
```

List Literals and Basic Operators

- can also use other lists as elements in a list, thereby creating a list of lists.

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

- when the Python interpreter evaluates a list literal, each of the elements is evaluated.

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

List Literals and Basic Operators

- can build lists of integers using the range

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

- 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]
```

- Concatenation (+) and equality (==) also work as expected for lists:

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

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

List Literals and Basic Operators

To print the contents of a list without the brackets and commas, you can use a for loop

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

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
```

List Literals and Basic Operators

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.
`L1 + L2`	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

- a list is changeable—that is, it is mutable.
-

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

Replacing an Element in a List

replace each number in a list with its square:

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> for index in range(len(numbers)):
        numbers[index] = numbers[index] ** 2
>>> numbers
[4, 9, 16, 25]
```

extract a list of the words in a sentence and then converted to uppercase letters within the list:

```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> for index in range(len(words)):
        words[index] = words[index].upper()
>>> words
['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
```

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.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]
```

append expects just the new element as an argument and adds the new element to the end of the list.

extend performs a similar operation, but adds the elements of its list argument to the end of the list.

+ operator builds and returns a brand new list containing the elements of the two operands

```
>>> example
[1, 2, 3, 11, 12, 13]
>>> example.insert(3, 25)
>>> example
[1, 2, 3, 25, 11, 12, 13]
```

➔ **insert expects an integer index and the new element as arguments. When the index is less than the length of the list, this method places the new element before the existing element at that index, after shifting elements to the right by one position.**

➔ **When the index is greater than or equal to the length of the list, the new element is added to the end of the list.**

```
>>> 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

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

- ➔ index to locate an element's position in a list.
- ➔ index raises an exception when the target element is not found.
- ➔ To guard against this ,first use the in operator to test for presence and then the index method if this test returns True .

Sorting a List

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

sort mutates a list by arranging its elements in ascending order.

Mutator Methods and the Value None

- some methods devoted entirely to modifying the internal state of the object. Such methods are called mutators .

Eg: insert , append , extend , pop , and sort .

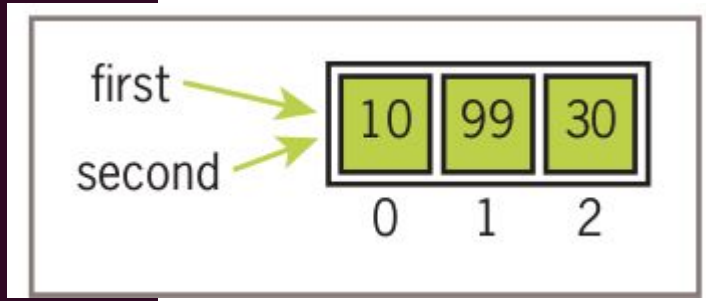
- These methods returns nothing
- Python automatically returns the special value None even when a method does not explicitly return a value.

Aliasing and Side Effects

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
[10, 99, 30]
>>> second
[10, 99, 30]
```

When the second element of the list named first is replaced, the second element of the list named second is replaced also. This type of change is what is known as a side effect .

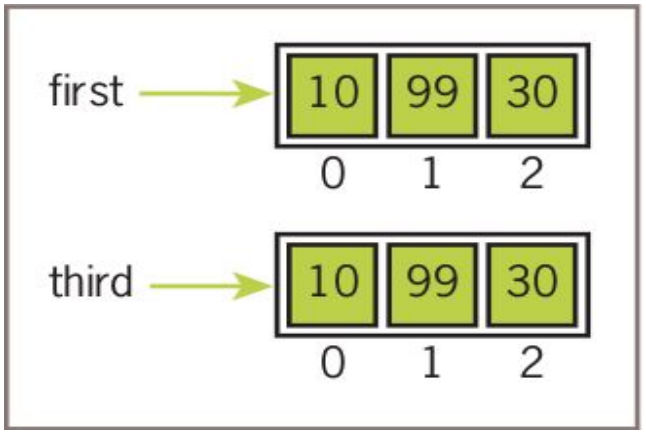
after the assignment second = first , the variables first and second refer to the exact same list object. They are aliases for the same object



Aliasing and Side Effects

```
>>> third = []
>>> for element in first:
        third.append(element)
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```

To prevent aliasing, create a new object and copy the contents of the original to it




```
>>> third = list(first)
```

Work……

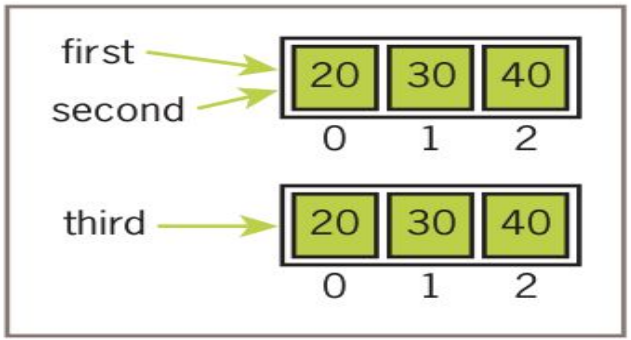
Using a List to Find the Median of a Set of Numbers

Equality: Object Identity and Structural Equivalence

- ➔ if the variables are aliases for the same object the == operator returns True. This relation is called **object identity**
- ➔ == also returns True if the contents of two different objects are the same. This relation is called structural **equivalence** .
- ➔ Python's **is** operator can be used to test for object identity.
- ➔ It returns True if the two operands refer to the exact same object, and it returns False if the operands refer to distinct objects (even if they are structurally equivalent).

Equality: Object Identity and Structural Equivalence

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first)          # Or first[:]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



Program sample question

- Python program to interchange first and last elements in a list.
- Python program for Reversing a List
- Python program to Count occurrences of an element in a list .
- Python program to count positive and negative numbers in a list.
- Write a Python program to read a list of numbers and sort the list in a non-decreasing order without using any built in functions. Separate function should be written to sort the list wherein the name of the list is passed as the parameter.

Tuples

A tuple is a type of sequence that resembles a list, except that, unlike a list, a tuple is immutable.

```
>>> 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')
```

Tuples

- ➔ Ordered: the items have a defined order, and that order will not change.
- ➔ Unchangeable: we cannot change, add or remove items after the tuple has been created.
- ➔ Allow Duplicates: Since tuples are indexed, they can have items with the same value.

Python Datetime

- 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.
- The date contains year, month, day, hour, minute, second, and microsecond.

```
import datetime

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

```
2022-05-19 09:22:30.898319
```

Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

```
2022
Thursday
```

Python Datetime

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec

Dictionaries

- A dictionary organizes information by association , not position
- dictionary associates a set of keys with values .
- Dictionary Literals:
  - Python dictionary is written as a sequence of key/value pairs separated by commas.
  - These pairs are called entries .
  - The entire sequence of entries is enclosed in curly braces ({ and }).
  - A colon (:) separates a key and its value.

Personal information: `{"Name":"Molly", "Age":18}`

Dictionaries

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

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

Dictionaries

- Accessing Values: the subscript can be used to obtain the value associated with a key.
- 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'
```

Dictionaries

the existence of a key can test using the operator in

```
>>> if "job" in info:
        print(info.["job"])
```

Dictionaries

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.
- ➔ if the key is absent, the default value passed to get is returned.

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

Dictionaries

To delete an entry from a dictionary, remove its key using the method pop.

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

Dictionaries

Traversing a Dictionary:

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

the dictionary method items() can be used to access the dictionary's entries

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

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

Copyright 2019 Cengage Learning. All Rights Reserved. May not

Dictionaries

If a special ordering of the keys is needed, you can obtain a list of keys using the keys

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

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.

Sample program questions

- Write a Python script to check whether a given key already exists in a dictionary.
- How to do a reverse dictionary lookup in Python.
- Write a Python script to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.
- Write a Python script to merge two Python dictionaries.
- Write a Python program to get the maximum and minimum value in a dictionary.

Python Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed and do not allow duplicate values.
- **myset = {"apple", "banana", "cherry"}**
- Set items are unchangeable, but can remove items and add new items.

Python Sets

- ➔ Unordered: Unordered means that the items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- ➔ Unchangeable: Set items are unchangeable, meaning that we cannot change the items after the set has been created.
- ➔ Duplicates Not Allowed: Sets cannot have two items with the same value.