Module II

Building Python Programs

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, A case study with lists

Work with Tuples. Sets

Work with Date and Time

Dictionaries

Dictionary functions

Dictionary literals

Adding and removing keys

Accessing and replacing values

Traversing dictionaries, Reverse lookup

Case Study – Data Structure Selection.

Accessing Characters and Substrings in Strings

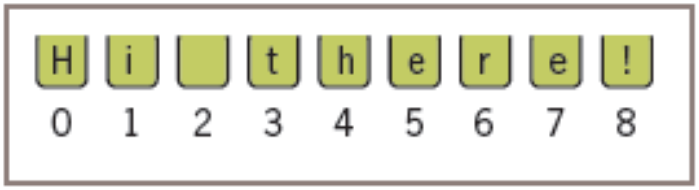
The Structure of Strings

- Strings 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.

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 string is an **immutable data structure.**

Its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.

The Subscript Operator

A simple for loop can access any of the characters in a string.

To inspect one character at a given position without visiting the whole string, use **subscript operator []**

```
<a string>[<an integer expression>]
```

The first part of this operation is the string.

The integer expression in brackets indicates the position of a particular character in that string.

The integer expression is also called an **index.**

```
>>> 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'
```

```
>>> data = "Hi there!"
>>> for index in range(len(data)):
        print(index, data[index])
0 H
1 i
2
3 t
4 h
5 e
6 r
7 e
8 !
```

Slicing for Substrings

Portions of strings is called substrings.

Python's subscript operator is used to obtain a substring through a process called **slicing.**

To extract a substring, place a colon (:) in the subscript.

An integer value can appear on either side of the colon.

```
>>> 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'
```

Testing for a Substring with the in Operator

Binary operator

The left operand of in is a target substring, and the right operand is the string to be searched.

The operator in returns True if the target string is somewhere in the search string, or False otherwise.

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

1. Assume that the variable **data** refers to the string **"myprogram.exe"**. Write the values of the following expressions:

   a. **data[2]**

   b. **data[-1]**

   c. **len(data)**

   d. **data[0:8]**

2. Assume that the variable **data** refers to the string **"myprogram.exe"**. Write the expressions that perform the following tasks:

   a. Extract the substring **"gram"** from **data**.

   b. Truncate the extension **".exe"** from **data**.

   c. Extract the character at the middle position from **data**.

3. Assume that the variable **myString** refers to a string. Write a code segment that uses a loop to print the characters of the string in reverse order.

4. Assume that the variable **myString** refers to a string, and the variable **reversedString** refers to an empty string. Write a loop that adds the characters from **myString** to **reversedString** in reverse order.

Data Encryption

Data encryption to protect information transmitted on networks.

FTPS and HTTPS, which are secure versions of FTP and HTTP for file transfer and Web page transfer, respectively

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.

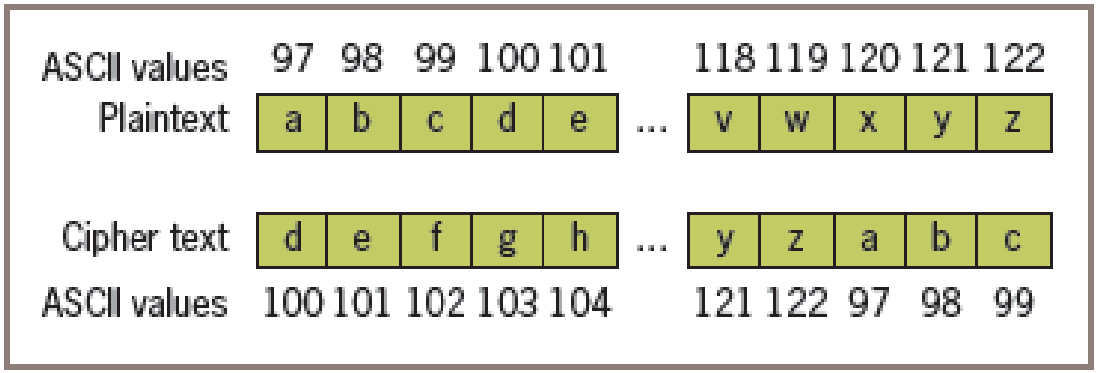
Caesar cipher

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".

To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement.



```
"""
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 Systems

Decimal number system -- base ten number system

Binary number system -- base two number system

octal (base eight) and hexadecimal (base 16)

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

The Positional System for Representing Numbers

$$415_{10} =$$

$$4 * 10^2 + 1 * 10^1 + 5 * 10^0 =$$

$$4 * 100 + 1 * 10 + 5 * 1 =$$

$$400 + 10 + 5 = 415$$

Positional values	100	10	1
Positions	2	1	0

Converting Binary to Decimal

$$1100111_2 =$$

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

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

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

```
"""
File: binarytodecimal.py
Converts a string of bits to a decimal integer.
"""

bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

```
Enter a string of bits: 1111
The integer value is 15

Enter a string of bits: 101
The integer value is 5
```

Converting Decimal to Binary

```
"""
File: decimaltobinary.py
Converts a decimal integer to a string of bits.
"""

decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
    print(0)
else:
    print("Quotient Remainder Binary")
    bitString = ""
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        bitString = str(remainder) + bitString
        print("%5d%8d%12s" % (decimal, remainder,
                              bitString))
  print("The binary representation is", bitString)

  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
```

String Methods

Python includes a set of string operations called **methods**

split to obtain a list of the words contained in an input string.

```
>>> 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
```

A method behaves like a function.

Unlike a function, a method is always called with a given data value called an object.

Which is placed before the method name in the call.

The syntax of a method call is

<an object>.<method name>(<argument-1>,..., <argument-*n*>*)*

Methods can also expect arguments and return values.

In Python, all data values are in fact objects, and every data type includes a set of methods to use with objects of that type.

dir(str) help(str.<method-name>)

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.

```
>>> 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!'
```

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

filename.split('.')[-1]

```
>>>filename="myfile.txt.exe.jpg“
>>>filename.split('.')[-1]
'jpg'
```

Text Files

Programs can receive their input from text files as well.

A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory.

When compared to keyboard input from a human user, the main advantages of taking input data from a file are :

- 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

Python programs can take input from a text file and output data to 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 and on the purposes for which the data are used.

When the data are numbers (either integers or floats), they must be separated by whitespace characters—spaces, tabs, and newlines in the file.

All data output to or input from a text file must be strings.

Thus, numbers must be converted to strings before output, and these strings must be converted back to numbers after input.

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.

This command 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.

To output text to end with a newline, include the escape character '\n' in the string.

```
>>> 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()
```

Failure to close an output file can result in data being lost. The reason for this is that many systems accumulate data values in a **buffer before writing them out as large chunks;** the close operation guarantees that data in the final chunk are output successfully.

Writing Numbers to a File

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

In Python, the values of most data types can be converted to strings by using the str function.

The resulting strings are then written to a file with a space or a newline as a separator character.

```
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

Opening a file for input is similar as opening a file for output.

The only thing that changes is the mode string, which, in the case of opening a file for input, is 'r'.

However, if a file with that name is not accessible, Python raises an error.

```
>>> f = open("myfile.txt", 'r')
```

To use the file method read to input the entire contents of the file as a single string.

```
>>> text = f.read()
>>> text
'First line.\nSecond line.\n'
>>> print(text)
First line.
Second line.
```

If the file contains multiple lines of text, the newline characters will be embedded in this 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.

To repeat an input, the file must be reopened, in order to “rewind” it for another input process.

It is not necessary to close the file.

An application might read and process the text one line at a time.

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.
```

print appears to output an extra newline.

This is because each line of text input from the file retains its newline character.

To read a specified number of lines from a file (say, the first line only), use the file method readline.

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

All of the file input operations return data to the program as strings.

If these strings represent other types of data, such as integers or floating-point numbers, the programmer must convert them to the appropriate types before manipulating them further.

In Python, the string representations of integers and floating-point numbers can be converted to the numbers themselves by using the functions int and float, respectively

To convert the line to the integer contained in it, the programmer runs the string method strip to remove the newline and then runs the int function to obtain the integer value.

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

Obtaining numbers from a text file in which they are separated by spaces is a bit trickier.

One method proceeds by reading lines in a for loop, as before.

But each line now can contain several integers separated by spaces.

Use the string method split to obtain a list of the strings representing these integers, and then process each string in this list with another for loop.

```
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)
```

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.

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

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.

When launch Python from the terminal or from IDLE, the shell is connected to a **current working directory.**

At any point during the execution of a program, a file can be accessed in this directory just by using the file's name.

Any other file or directory within the computer's file system by using a **pathname.**

A file's pathname specifies the chain of directories needed to access a file or directory.

When the chain starts with the root directory, it's called an absolute pathname.

When the chain starts from the current working directory, it's called a relative pathname

An absolute pathname consists of one or more directory names, separated by the '/' character (for a Unix-based system and macOS) or the '\' character (for a Windows-based system).

The root directory is the leftmost name and the target directory or file name is the rightmost name.

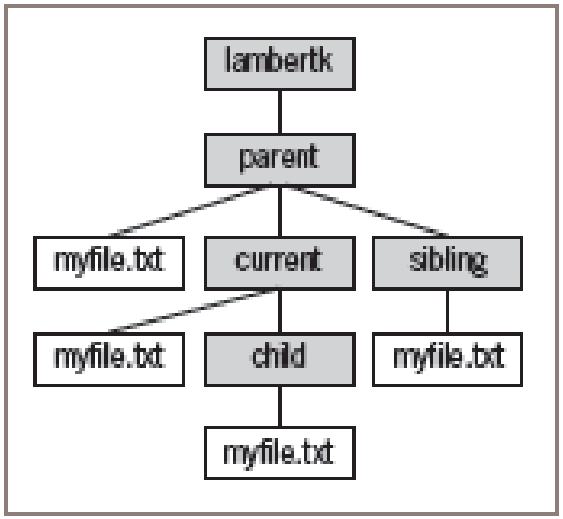
The '/' character must begin an absolute pathname on Unix-based systems, and a disk drive letter must begin an absolute pathname on Windows-based systems.

If you are mentioning a pathname in a Python string, you must escape each '\' character with another '\' character.

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

C:\Users\lambertk\parent\current\child\myfile.txt

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



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

Figure 4-6 A portion of a file system

To open the files named **myfile.txt in the child, parent, and sibling directories, where current is the** current working directory, use relative pathnames as follows:

```
childFile = open("child/myfile.txt", 'r')
parentFile = open("../myfile.txt", 'r')
siblingFile = open("../sibling/myfile.txt", 'r')
```

When designing Python programs that interact with files, it's a good idea to include error recovery.

For example, before attempting to open a file for input, the programmer should check to see if a file with the given pathname exists on the disk.

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

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.

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

In 1949, Dr. Rudolf Flesch published *The Art of Readable Writing*, in which he proposed a measure of text readability known as the Flesch Index. 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

In this case study, we develop a program that computes the Flesch Index for a text file.

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.

During analysis, we consult experts in the problem domain to learn any information that might be relevant in solving the problem. For our problem, this information includes the definitions of *sentence*, *word*, and *syllable*. For the purposes of this program, these terms are defined in Table 4-7.

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.

Note that the definitions of *word* and *sentence* are approximations. Some words, such as *doubles* and *kettles*, end in *-es* but will be counted as having one syllable, and an ellipsis (…) will be counted as three syllables.

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 first and last tasks require no design. Let's assume that the text is input as a single string from the file and is then processed in tasks 2–4. These three tasks can be designed as code segments that use the input string and produce an integer value. Task 5, computing the Flesch Index, uses the three integer results of tasks 2–4 to compute the Flesch Index. Lastly, task 6 is a code segment that uses the same integers and computes the Grade Level Equivalent. The five tasks are listed in Table 4-8, where `text` is a variable that refers to the string read from the file.

All the real work is done in the tasks that count the items:

- Add the number of characters in `text` that end the sentences. These characters were specified in analysis, and the string method `count` is used to count them in the algorithm.
- Split `text` into a list of words and determine the `text` length.
- Count the syllables in each word in `text`.

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.

Table 4-8 The tasks defined in the text analysis program

The last task is the most complex. For each word in the text, we must count the syllables in that word. From analysis, we know that each distinct vowel counts as a syllable, unless it is in the endings *-ed*, *-es*, or *-e* (but not *-le*). For now, we ignore the possibility of consecutive vowels.

Implementation (Coding)

The main tasks are marked off in the program code with a blank line and a comment.

```
"""
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

Although the main tasks all collaborate in the text analysis program, they can be tested more or less independently, before the entire program is tested. After all, there is no point in running the complete program if you are unsure that even one of the tasks does not work correctly.

This kind of procedure is called **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**. For example, here is a driver that tests the code for computing the Flesch Index and the Grade Level Equivalent without using a text file:

```
"""
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)
```

This driver allows the programmer not only to verify the two tasks, but also to obtain some data to use when testing the complete program later on. For example, 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.

In 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, you should use longer files. For example, you might see if plaintext versions of Dr. Seuss's *Green Eggs and Ham* and Shakespeare's *Hamlet* produce grade levels of 5th grade and 12th grade, respectively. Or you could test the program with its own source program file—but we predict that its readability will seem quite low, because it lacks most of the standard end-of-sentence marks!

Design with Functions

Functions as Abstraction Mechanisms

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

Functions Eliminate Redundancy

Functions serve as abstraction mechanisms is by eliminating redundant, or repetitious code.

```
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.

It requires the programmer to laboriously enter or copy the same code over and over, and to get it correct every time.

Then, if the programmer decides to improve the algorithm by adding a new feature or making it more efficient, he or she must **revise each instance** of the redundant code throughout the entire program.

By relying on a single function definition, instead of multiple instances of redundant code, the programmer is free to write only a single algorithm in just one place—say, in a library module.

Any other module or program can then import the function for its use.

Once imported, the function can be called as many times as necessary.

When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition.

There is no need to edit the parts of the program that call the function.

Functions Hide Complexity

Functions serve as abstraction mechanisms is by hiding complicated details.

The idea of summing a range of numbers is simple, the code for computing a summation is not.

Not about the amount or length of the code, but also about the number of interacting components.

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

Functions Support General Methods with Systematic Variations

An algorithm is a general method for solving a class of problems.

The individual problems that make up a class of problems are known as problem instances.

The problem instances for our summation algorithm are the pairs of numbers that specify the lower and upper bounds of the range of numbers to be summed.

The summation function contains both the code for the summation algorithm and the means of supplying problem instances to this algorithm.

The problem instances are the data sent as arguments to the function.

The parameters or argument names in the function's header behave like variables waiting to be assigned data whenever the function is called.

Functions Support the Division of Labour

In a computer program, functions can enforce a division of labour.

Ideally, each function performs a single coherent task, such as computing a summation or formatting a table of data for output.

Each function is responsible for using certain data, computing certain results, and returning these to the parts of the program that requested them.

Each of the tasks required by a system can be assigned to a function, including the tasks of managing or coordinating the use of other functions.

Problem Solving with Top-Down Design

One popular design strategy for programs of any significant size and complexity is called top-down design.

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

As each sub problem is isolated, its solution is assigned to a function.

Problem decomposition may continue down to lower levels, because a sub problem might in turn contain two or more lower-level problems to solve.

As functions are developed to solve each sub problem, the solution to the overall problem is gradually filled out in detail.

This process is also called stepwise refinement.

The program requires simple input and output components, so these can be expressed as statements within a main function.

However, the processing of the input is complex enough to decompose into smaller sub processes, such as obtaining the counts of the sentences, words, and syllables and calculating the readability scores.

Generally, develop a new function for each of these computational tasks.

The relationships among the functions in this design are expressed in the structure chart

The Design of the Text-Analysis Program

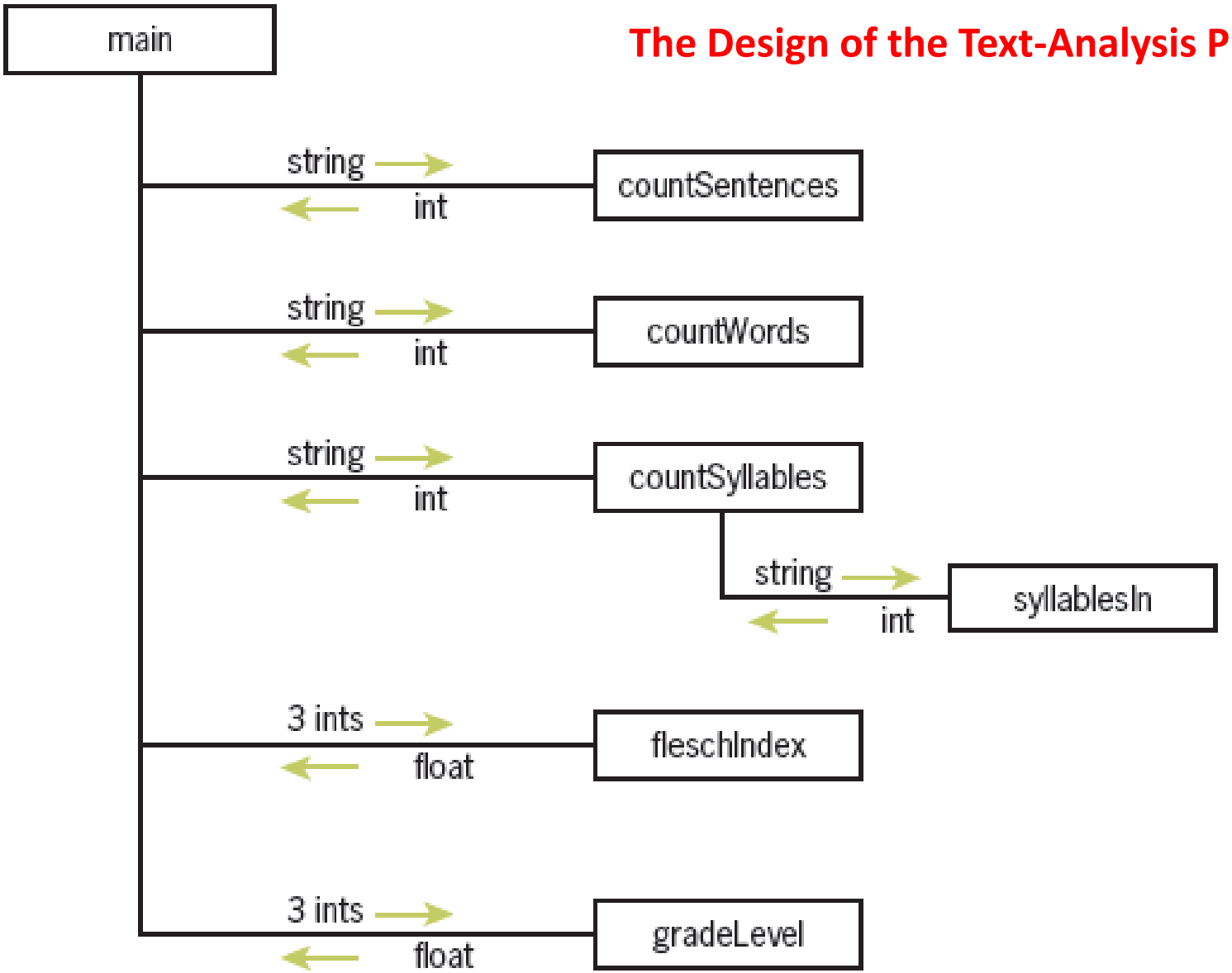


Figure 6-1 A structure chart for the text-analysis program

Each box in the structure chart is labelled with a function name.

The main function at the top is where the design begins, and decomposition leads us to the lower-level functions on which main depends.

The lines connecting the boxes are labelled with data type names, and arrows indicate the flow of data between them.

For example, the function **countSentences** takes a string as an argument and returns the number of sentences in that string.

Note that all functions except one are just one level below main.

Because this program does not have a deep structure, the programmer can develop it quickly just by thinking of the results that main needs to obtain from its collaborators.

The Design of the Sentence-Generator Program

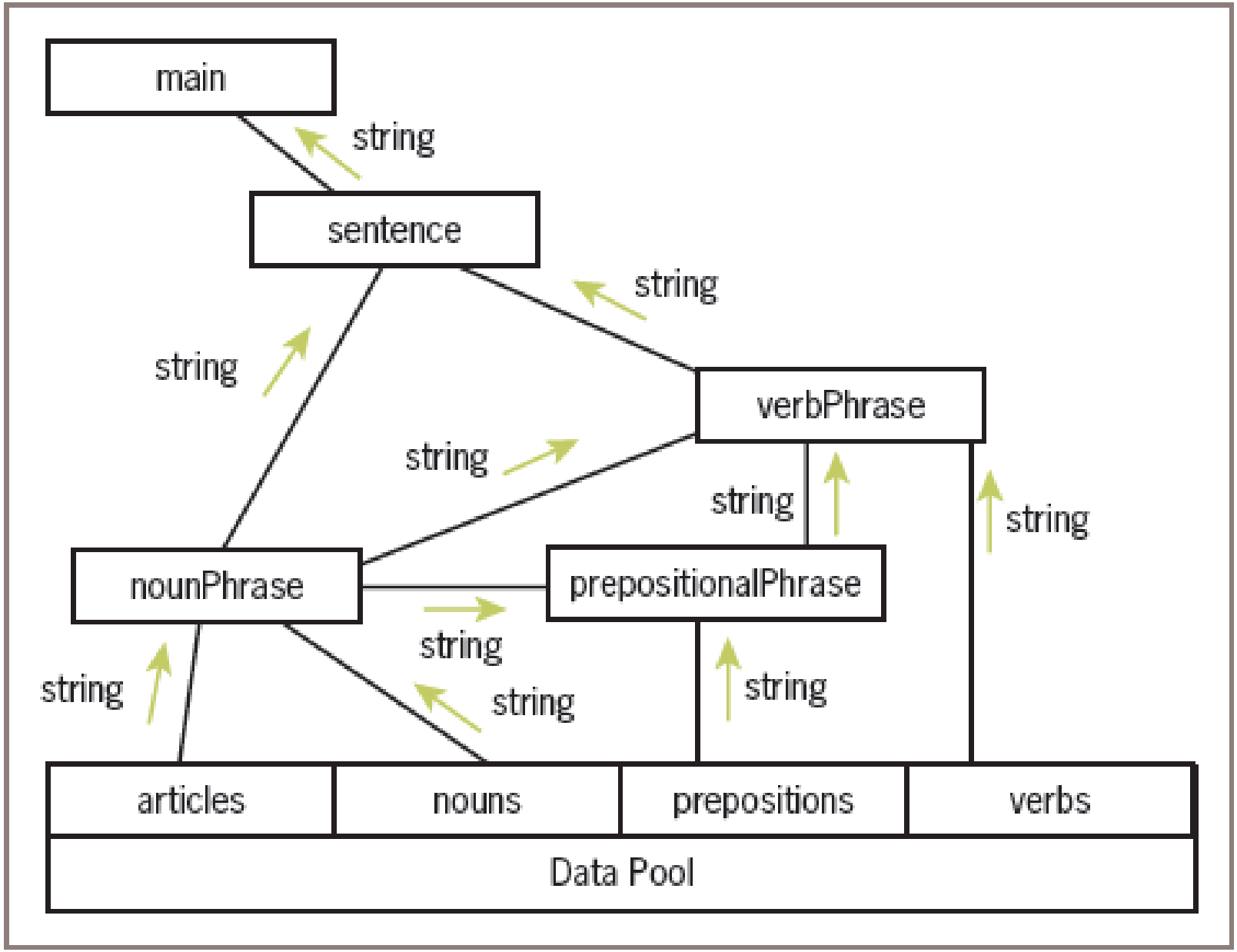


Figure 6-2 A structure chart for the sentence-generator program

Defining Simple Functions

The Syntax of Simple Function Definitions

The definition of this function consists of a header and a body.

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

The header includes the keyword **def** as well as the function name and list of parameters.

The function's body contains one or more statements.

The function's body contains the statements that execute when the function is called.

Function may contain a single return statement, which simply returns the result of operations by the function.

Function also contains a docstring.

This string contains information about what the function does.

It is displayed in the shell when the programmer enters help(square).

Parameters and Arguments

A parameter is the name used in the function definition for an argument that is passed to the function when it is called.

The number and positions of the arguments of a function call should match the number and positions of the parameters in that function's definition.

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

The return Statement

The programmer places a return statement at each exit point of a function when that function should explicitly return a value.

```
return <expression>
```

Upon encountering a return statement, Python evaluates the expression and immediately transfers control back to the caller of the function.

The value of the expression is also sent back to the caller.

If a function contains no return statement, Python transfers control to the caller after the last statement in the function's body is executed, and the special value **None** is automatically returned.

Boolean Functions

A Boolean function usually tests its argument for the presence or absence of some property.

The function returns True if the property is present, or False otherwise.

```
>>> odd(5)
True
>>> odd(6)
False

def odd(x):
    """Returns True if x is odd or False otherwise."""
    if x % 2 == 1:
        return True
    else:
        return False
```

Design with Recursive Functions

The sub problems 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.

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

Iterative algorithm

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

The equivalent recursive function

Most recursive functions expect at least one argument.

This data value is used to test for the base case that ends the recursive process, and it is modified in some way before each recursive step.

The modification of the data value should produce a new data value that allows the function to reach the base case eventually.

```
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):
    """Returns the sum of the numbers from lower through
upper,
    and outputs a trace of the arguments and return values
    on each call"""
    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)
1 4
    2 4
        3 4
            4 4
                5 4
                0
            4
        7
    9
10
10
```

Using Recursive Definitions to Construct Recursive Functions

Recursive functions are frequently used to design algorithms for computing values that have a recursive definition.

A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases.

A recursive definition of the n^{th} ***Fibonacci number is the following:***
Fib(n) = 1, when n = 1 or n = 2
Fib(n) = Fib(n - 1) + Fib(n - 2), for all n > 2

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

Infinite Recursion

Recursive functions tend to be simpler than the corresponding loops, but they still require thorough testing.

One design error that might trip up a programmer occurs when the function can (theoretically) continue executing forever, a situation known as **infinite recursion.**

Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.

In fact, the Python virtual machine eventually runs out of memory resources to manage the process, so it halts execution with a message indicating a **stack overflow error**

The Costs and Benefits of Recursion

The run-time system on a real computer, such as 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**.

In this type of storage, the system places the values of the arguments and the return address for each function call.

Space for the function call's return value is also reserved in its stack frame.

When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code, and the memory for the stack frame is deallocated.

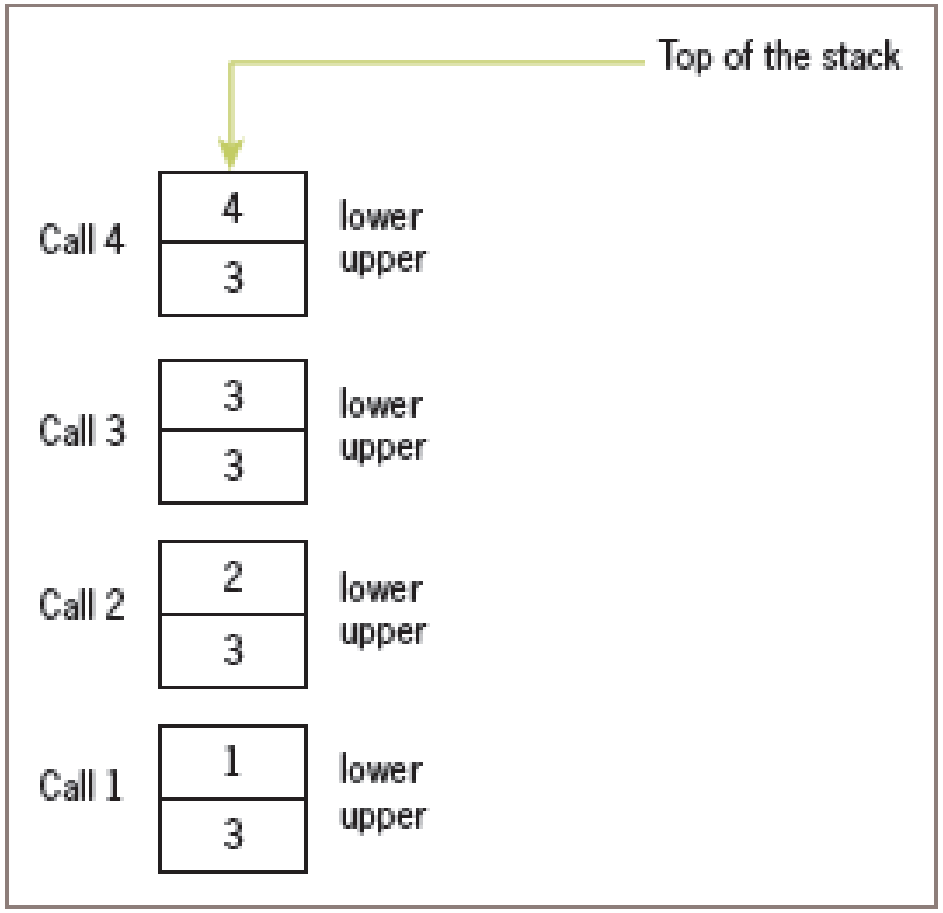


Figure 6-4 The stack frames for `displayRange(1, 3)`

consider developing recursive solutions when have a recursive structure.

Smart compilers also exist that can optimize some recursive functions by translating them to iterative machine code.

Managing a program's namespace

Program's **namespace**—The set of its variables and their values

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)
```

doctor.py

1. Module variables.

The names replacements and changePerson are introduced at the level of the module.

Although replacements names a dictionary and changePerson names a function, they are both considered variables.

Module variables of the doctor module can be used by importing it and entering dir(doctor) at a shell prompt. When module variables are introduced in a program, they are immediately given a value.

2. Parameters.

The name sentence is a parameter of the function changePerson.

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

3. Temporary variables.

The names words, replyWords, and word are introduced in the body of the function changePerson.

Like module variables, temporary variables receive their values as soon as they are introduced.

4. Method names.

The names split and join are introduced or defined in the str type.

A method reference always uses an object, in this case, a string, followed by a dot and the method name.

Scope

In a program, the **context that gives a name, a meaning** is called its **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 words, replyWords, and word is the area of code in the body of the function changePerson, just below where each variable is introduced.

In general, the meanings of temporary variables are restricted to the body of the functions in which they are introduced, and they are invisible elsewhere in a module.

The scope of the parameter sentence is the entire body of the function changePerson.

Like temporary variables, parameters are invisible outside the function definitions where they are introduced.

The scope of the module variables replacements and changePerson includes the entire module below the point where the variables are introduced.

This includes the code nested in the body of the function changePerson.

The scope of these variables also includes the nested bodies of other function definitions that occur *earlier.*

This allows these variables to be referenced by any functions, regardless of where they are defined in the module.

For example, the reply function, which calls changePerson, might be defined before changePerson in the doctor module.

Python function can reference a module variable for its value, it cannot under normal circumstances assign a new value to a module variable.

When such an attempt is made, the PVM creates a new, temporary variable of the same name within the function.

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

Lifetime

A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it.

When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM.

Module variables come into existence when they are introduced via assignment and generally 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.

The module variable x comes into existence before the temporary Variable x and survives the call of function f. During the call of f, storage exists for both variables, so their values remain distinct.

```
x = 5
def f():
    x = 10
f()
print(x)
```

Using Keywords for Default and Optional Arguments

Adding an argument or two to a function can increase its generality by extending the range of situations in which the function can be used.

The use of the extra arguments should be optional for the caller of the Function.

When the function is called without the extra arguments, it provides reasonable default values for those arguments that produce the expected results.

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.

These are the ones that are “essential” for the use of the function by any caller.

Following the required arguments are one or more default arguments or keyword arguments.

These are assignments of values to the argument names.

When the function is called without these arguments, their default values are automatically assigned to them.

When the function is called with these arguments, the default values are overridden by the caller’s values.

```
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
```

Replace header

```
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
```

When using functions that have default arguments, provide the required arguments and place them in the same positions as they are in the function definition's header.

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

1. By position. In this case, the values are supplied in the order in which the arguments occur in the function header. Defaults are used for any arguments that are omitted.

2. By keyword. In this case, one or more values can be supplied in any order, using the syntax <key> = <value> in the function call.

```
>>> 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

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

In Python, functions can be treated as first-class data objects.

This means that they can be assigned to variables (as they are when they are defined), 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
```

Passing a function as an argument to another function is not different from passing any other datum.

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.

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

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.

Suppose, there is a list named words that contains strings that represent integers.

To replace each string with the corresponding integer value the map function can be used.

```
>>> 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 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.

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

Reducing

Takes a list of values and repeatedly apply a function to accumulate a single data value.

Example: summation

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.

```
>>> 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

A lambda is an **anonymous function.**

It has no name of its own, but it 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 very tight and restrictive:

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

All of the code must appear on one line.

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

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data)     # Produce the sum
10
>>> reduce(lambda x, y: x * y, data)     # Produce the product
24
```

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

Lists

List is a sequence of data values called items or elements.

An item can be of any type.

1. A shopping list for the grocery store
2. A to-do list
3. A roster for an athletic team
4. A guest list for a wedding
5. A recipe, which is a list of instructions
6. A text document, which is a list of lines
7. The names in a phone book

The logical structure of a list resembles the structure of a string.

Each of the items in a list is ordered by position.

Like a character in a string, each item in a list has a unique index that specifies its position.

The index of the first item is 0, and the index of the last item is the length of the list minus 1.

List Literals and Basic Operators

A 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
```

Other lists can be used as elements in a list, thereby creating a list of lists.

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

Python interpreter evaluates a list literal.

When an element is a number or a string, that literal is included in the resulting list.

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]
```

```
>>> 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', '!']
```

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
```

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
```

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
```

Operator or Function
`L[<an integer expression>]`
`L[<start>:<end>]`
`L1 + L2`
`print(L)`
`len(L)`
`list(range(<upper>))`
`==, !=, <, >, <=, >=`
`for <variable> in L: <statement>`
`<any value> in L`

Replacing an Element in a List

A string is immutable, its structure and contents cannot be changed.

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

At any point in a list's lifetime, elements can be inserted, removed, or replaced.

The list itself maintains its identity but its internal state—its length and its contents—can change.

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

```
>>> 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]
```

```
>>> 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

help(list.<method name>)

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(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]
```

```
>>> 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

Method index to locate an element's position in a list.

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

Sorting a List

List's elements are always ordered by position, it is possible to impose a natural ordering on them

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

List Comprehension

List comprehension offers a shorter syntax to create a new list based on the values of an existing list.

newlist = [*expression* for *item* in *iterable* if *condition* == True]

```
>>> fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
>>> newlist = [x for x in fruits if "a" in x]
>>> newlist
['apple', 'banana', 'mango']
>>>
```

Mutator Methods

Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object.

Such methods are called **mutators.**

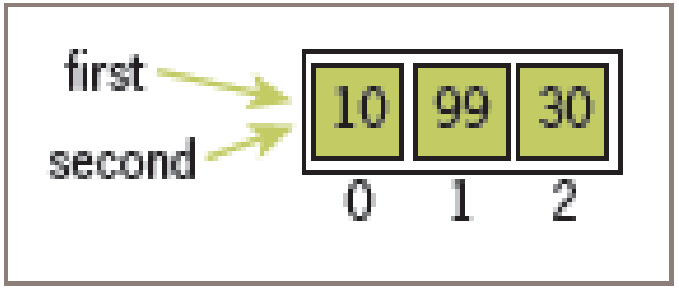
Eg: list methods insert, append, extend, pop, and sort.

Because a change of state is all that is desired, a mutator method usually returns no value of interest to the caller (but note that pop is an exception to this rule).

numbers and strings are immutable. Cannot change their internal structure. Lists are mutable, can replace, insert, or remove elements.

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.

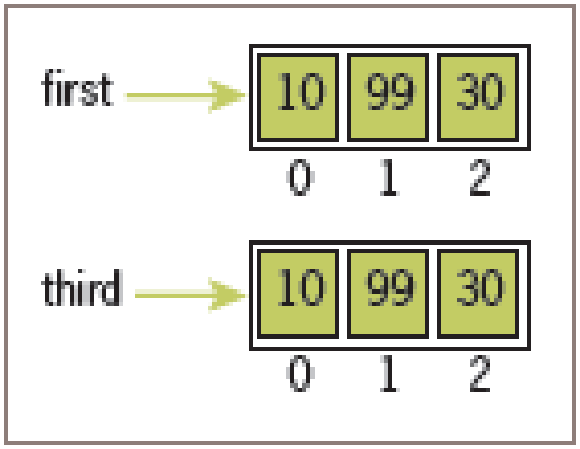
This type of change is what is known as a **side effect.**

This happens because after the assignment second = first, the variables first and Second refer to the exact same list object.

They are **aliases for the same object.** This phenomenon is known as **aliasing.**

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

```
>>> 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]
```



simpler way to copy a list is to pass the source list to a call of the list function

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

Equality: Object Identity and Structural Equivalence

The == operator returns True if the variables are aliases for the same object.

== also returns True if the contents of two different objects are the same.

The first relation is called object identity

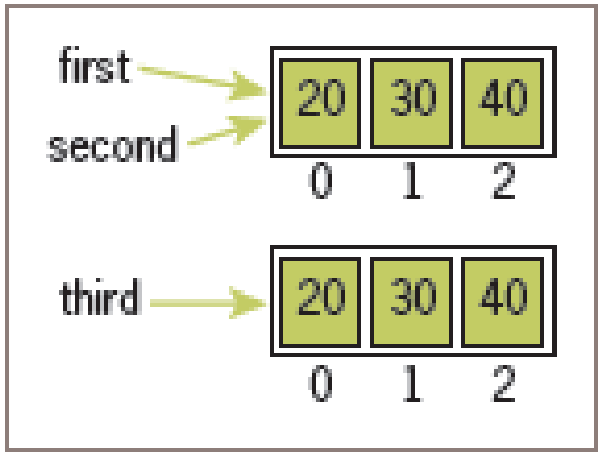
The second relation is called structural equivalence

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,

It returns False if the operands refer to distinct objects (even if they are structurally equivalent).

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



Tuples

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

Tuple literal in Python can be expressed as enclosing its elements in parentheses instead of square brackets.

```
>>> 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')
```

Most of the operators and functions used with lists also apply to tuples

A tuple is a collection which is ordered and **unchangeable**.

Tuples are indexed, they can have items with the same value.

To determine how many items a tuple has, use the **len()**

Access tuple items by referring to the index number, inside square brackets.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Negative indexing means start from the end.
-1 refers to the last item, -2 refers to the second last item etc.

Specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango");
print(thistuple[2:5])

#This will return the items from position 2 to 5.

#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
```

```
('cherry', 'orange', 'kiwi')
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```

To determine if a specified item is present in a tuple use the in keyword:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Change Tuple Values

Once a tuple is created, cannot change its values.

Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround.

Convert the tuple into a list, change the list, and convert the list back into a tuple.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

```
("apple", "kiwi", "cherry")
```

Add Items

Convert into a list

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

Add tuple to a tuple

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Remove Items

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Packing a Tuple and Unpacking a Tuple

Creating a tuple, (normally assign values to it) is called "packing" a tuple:

Extracting the values back into variables is called "unpacking"

```
fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
cherry
```

If the number of variables is less than the number of values, add an * to the variable name and the values will be assigned to the variable as a list:

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
['cherry', 'strawberry', 'raspberry']
```

Loop Through a Tuple

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Join two tuples:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple)
```

```
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

Defining a main Function

In Pythons there is a special function named main, that serves as the entry point for the script.

This function usually expects no arguments and returns no value.

Its purpose might be to take inputs, process them by calling other functions, and print the results.

The definition of the main function and the other function definitions need appear in no particular order in the script, as long as main is called at the very end of the script.

```
"""
File: computesquare.py
Illustrates the definition of a main function.
"""


def main():
    """The main function for this script."""
    number = float(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)

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

# The entry point for program execution
if __name__ == "__main:"__
    main()
```

When the script is imported as a module, the value of the module variable __name__ will be the name of the module, “computeSquare".

When the script is launched from IDLE or a terminal prompt, the value of the module variable __name __ will be "__main__".

In that case, the main function is called and the script runs as a standalone program

Dictionaries

Lists organize their elements by position.

Useful when to locate the first element, the last element, or visit each element in a sequence.

A dictionary organizes information by **association, not position.**

In computer science, data structures organized by association are also called **tables or association lists.**

In Python, a **dictionary associates a set of keys with values.**

The keys in *Webster's Dictionary comprise the set of words, whereas the associated data values are their definitions.*

Dictionary Literals

A Python dictionary is written as a sequence of key/value pairs separated by commas.

These pairs are sometimes called **entries.**

The entire sequence of entries is enclosed in curly braces ({ and }).

A colon (:) separates a key and its value.

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

an empty dictionary—that is, a dictionary that contains no entries.

{ }

The keys in a dictionary can be data of any immutable types, including tuples.

Keys normally are strings or integers.

The associated values can be of any types.

Although the entries may appear to be ordered in a dictionary, this ordering is not significant

Adding Keys and Replacing Values

A new key/value pair can be added to a dictionary by using the subscript operator [].

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

```
>>> 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

Use the subscript 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'
```

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

Use the method get to access values.

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.

```
>>> 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 exception.

```
>>> 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.

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

```
>>> grades
{90: 'A', 80: 'B', 70: 'C'}
>>> for key in grades:
        print(key,grades[key])


90 A
80 B
70 C
>>>
```

Use the dictionary method items() to access the dictionary's entries.

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

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)
```

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


90 A
80 B
70 C
>>>
```

The use of a structure containing variables to access data within another structure is called **pattern matching**.

```
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.

Example: The Hexadecimal System

Python function that uses that method to convert a hexadecimal number to a binary number.

The algorithm visits each digit in the hexadecimal number, selects the corresponding four bits that represent that digit in binary, and adds these bits to a result string.

```
hexToBinaryTable = {'0':'0000', '1':'0001', '2':'0010',
                    '3':'0011', '4':'0100', '5':'0101',
                    '6':'0110', '7':'0111', '8':'1000',
                    '9':'1001', 'A':'1010', 'B':'1011',
                    'C':'1100', 'D':'1101', 'E':'1110',
                    'F':'1111'}
```

```
def convert(number, table):
    """Builds and returns the base two representation of
    number."""
    binary = ""
    for digit in number:
        binary = table[digit] + binary
    return binary

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

Reverse lookup

A reverse dictionary lookup will return a list containing all the keys in a dictionary.

Dictionary keys map values and are used to access values in dictionaries.

```
>>> dict = {1:"One",2:"Two",3:"Three"}
>>> val="Two"
>>> for key,value in dict.items():
        if(val==value):
                print(key)


2
>>>
```

Sets

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in structure in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable*, and *unindexed*.

Sets are unordered, so you cannot be sure in which order the items will appear.

Sets are written with curly brackets.

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time use them, and cannot be referred to by index or key.

```
>>>
>>> theset={"A","B","C"}
>>> theset
{'B', 'A', 'C'}
>>> theset={"A","A","B"}
>>> theset
{'B', 'A'}
>>>
```

Length of a Set

len () is used to determine the number of items in set.

```
print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

```
set1 = {"abc", 34, True, 40, "male"}
```

The set() Constructor

The set() constructor to make a set.

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

Once a set is created, Cannot change its items, but new items can be added.

To add one item to a set use the add() method.

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

thisset.add("orange")

print(thisset)
```

```
{'banana', 'apple', 'cherry', 'orange'}
```

Accessing Items

It is not possible to access items in a set by referring to an index or a key. (?)

Loop through the set items using a for loop, or by using the in keyword.

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

for x in thisset:
  print(x)
```

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

print("banana" in thisset)
```

```
True
```

To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

update() method argument can be any iterable object (tuples, lists, dictionaries etc.).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)

print(thisset)
```

Remove Item

To remove an item in a set, use the remove(), or the discard().

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

thisset.remove("banana")

print(thisset)
```

If the item to remove does not exist, remove() will raise an error.

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

thisset.discard("banana")

print(thisset)
```

If the item to remove does not exist, discard() will **NOT** raise an error.

use the pop() method to remove an item,

This method will remove the *last* item.

Sets are unordered, so it will not know what item that gets removed.

```
MY_set = {3,"ONE","TWO",1,"three",2,4.0,5,6.88}

x = MY_set.pop()

print(x) #removed item

print(MY_set) #the set after removal

x = MY_set.pop()

print(x) #removed item

print(MY_set) #the set after removal

x = MY_set.pop()

print(x) #removed item

print(MY_set) #the set after removal
```

```
1
{2, 3, 4.0, 5, 6.88, 'ONE', 'three', 'TWO'}
2
{3, 4.0, 5, 6.88, 'ONE', 'three', 'TWO'}
3
{4.0, 5, 6.88, 'ONE', 'three', 'TWO'}
```

The clear() method empties the set

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

thisset.clear()

print(thisset)
```

```
set()
```

The del keyword will delete the set completely

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

del thisset

print(thisset) #this will raise an error because the set no longer exists
```

```
Traceback (most recent call last):
  File "demo_set_del.py", line 5, in <module>
    print(thisset) #this will raise an error because the set no longer exists
NameError: name 'thisset' is not defined
```

Join Two Sets

use the union() method that returns a new set containing all items from both set

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}

set3 = set1.union(set2)
print(set3)
```

```
{3, 'a', 1, 'c', 'b', 2}
```

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

Work with Date and Time

A date in Python is not a data type.

Import a module named datetime to work with dates as date objects.

```
import datetime

x = datetime.datetime.now()

print(x)
```

```
2022-05-29 07:34:05.870300
```

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

The datetime module has many methods to return information about the date object.

Get the year and name of weekday

```
import datetime

x = datetime.datetime.now()

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

```
2022
Sunday
```

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).

Creating Date Objects

To create a date, use the datetime() class (constructor).

The datetime() class requires three parameters to create a date: year, month, day.

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

```
2020-05-17 00:00:00
```

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"))
```

```
June
```

```python
import datetime

x = datetime.datetime.now()

print(x)

print("Day ",x.strftime("%A"))
print("Month Short ",x.strftime("%b"))
print("Month Full ",x.strftime("%B"))
print("Week ",x.strftime("%w"))
print("Date ",x.strftime("%d"))
print("Year short",x.strftime("%y"))
print("Year full ",x.strftime("%Y"))
print("Hour ",x.strftime("%H"))
print("Minute ",x.strftime("%M"))
print("Second ",x.strftime("%S"))
```

```
2022-05-29 02:19:36.001180
Day  Sunday
Month Short  May
Month Full  May
Week  0
Date  29
Year short 22
Year full  2022
Hour  02
Minute  19
Second  36
```

End
Module II