Chapter 1
Why should you learn to write programs?

Writing programs is a very creative and rewarding activity.
You can write programs for many reasons

- ranging from making your living to solving a difficult data analysis problem
- having fun to helping someone else solve a problem.

The hardware in our current-day computers is essentially built to continuously ask us the question, **“What would you like me to do next?”**

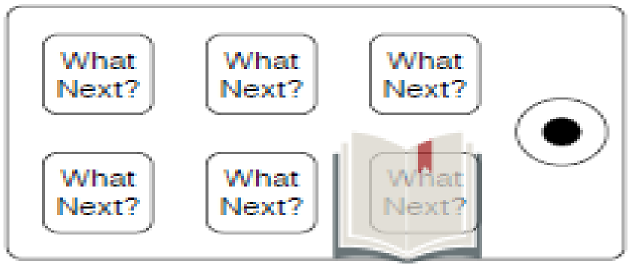


Figure 1.1: Personal Digital Assistant

Programmers add an operating system and a set of applications to the hardware and we end up with a Personal Digital Assistant that is quite helpful and capable of helping us do many different things.

Our computers are fast and have vast amounts of memory and could be very helpful to us if we only knew the language to speak to explain to the computer what we would like it to “do next”.

1.1 Creativity and motivation

- Building useful, elegant, and clever programs for others to use is a very creative activity.

  Eg Your computer or Personal Digital Assistant (PDA) usually contains many different programs from many different groups of programmers, each competing for your attention and interest. They try their best to meet your needs and give you a great user experience in the process.

- Primary motivation is to be more productive in handling the data and information that we will encounter in our lives

1.2 Computer hardware architecture

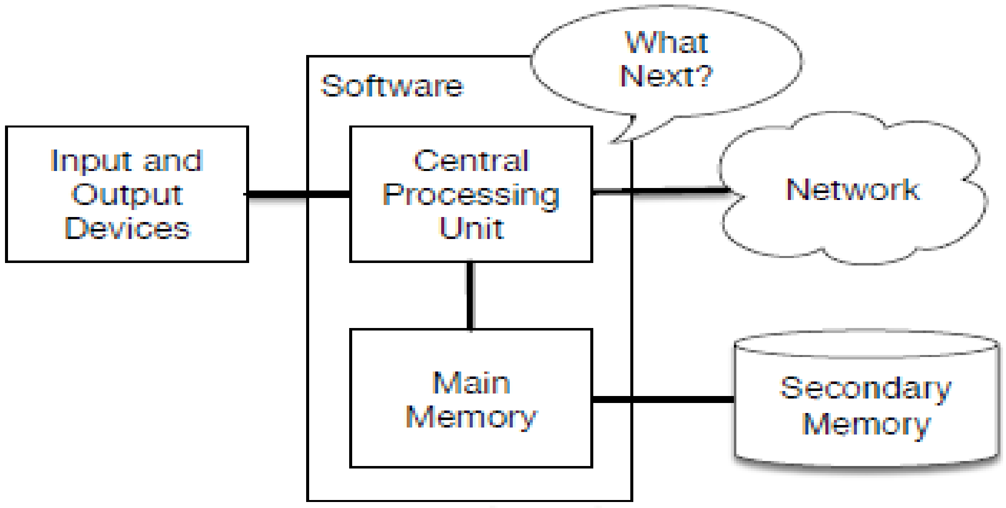


Figure 1.3: Hardware Architecture

- **The *Central Processing Unit* (or CPU)**
  - is the part of the computer that is built to be obsessed with "what is next?"
  - If your computer is rated at 3.0 Gigahertz, it means that the CPU will ask "What next?" three billion times per second.
  - You are going to have to learn how to talk fast to keep up with the CPU.
- **The *Main Memory***
  - is used to store information that the CPU needs in a hurry.
  - The main memory is nearly as fast as the CPU.
  - But the information stored in the main memory vanishes when the computer is turned off.
- **The *Secondary Memory***
  - is also used to store information, but it is much slower than the main memory.
  - The advantage of the secondary memory is that it can store information even when there is no power to the computer.
  - Examples of secondary memory are disk drives or flash memory (typically found in USB sticks and portable music players).
- **The *Input and Output Devices***
  - are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc.

- They are all of the ways we interact with the computer.

• These days, most computers also have a ***Network Connection***
- to retrieve information over a network.
- We can think of the network as a very slow place to store and retrieve data that might not always be "up".
- The network is a slower and at times unreliable form of ***Secondary Memory***.

1.3 Understanding programming

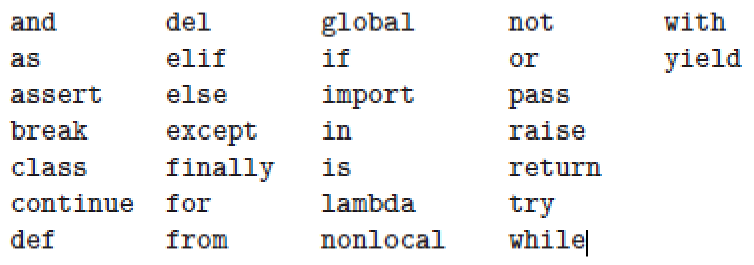
You need two skills to be a programmer:

• First, you need to know the programming language (Python) -
- You need to know the vocabulary and the grammar.
- You need to be able to spell the words in this new language properly and know how to construct well-formed "sentences" in this new language.

• Second, you need to "tell a story".
- In writing a story, you combine words and sentences to convey an idea to the reader.
- There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback.
- In programming, our program is the "story" and the problem you are trying to solve is the "idea".

Once you learn one programming language such as Python, you will find it much easier to learn a second programming language such as JavaScript or C++.

1.4 Words and sentences
The reserved words in the language where humans talk to Python include the following:


```
and       del       global    not       with
as        elif      if        or        yield
assert    else      import    pass
break     except    in        raise
class     finally   is        return
continue  for       lambda    try
def       from      nonlocal  while
```

Eg for a sentence in python

```
print('Hello world!')
```

Sentence starts with the function *print* followed by a string of text of our choosing enclosed in single quotes.

1.5 Conversing with Python

- The >>> prompt is the Python interpreter's way of asking you, "What do you want me to do next?"

  Eg
  ```
  >>> print('Hello world!')
  Hello world!

  >>> print('You must be the legendary god that comes from the sky')
  You must be the legendary god that comes from the sky
  >>> print('We have been waiting for you for a long time')
  We have been waiting for you for a long time
  >>> print('Our legend says you will be very tasty with mustard')
  Our legend says you will be very tasty with mustard
  >>> print 'We will have a feast tonight unless you say
      File "<stdin>", line 1
      print 'We will have a feast tonight unless you say
            ^
      SyntaxError: Missing parentheses in call to 'print'
  >>>
  ```
- Python is amazingly complex and powerful and very picky about the syntax you use to communicate with it
- Python is *not* intelligent. You are really just having a conversation with yourself, but using proper syntax.

The proper way to quit python

```
>>> quit()
```

The proper way to say "good-bye" to Python is to enter *quit()* at the interactive chevron >>> prompt.

1.6 Terminology: interpreter and compiler

- The CPU understands a language we call *machine language*.
- Machine language is very simple and frankly very tiresome to write because it is represented all in zeros and ones:

```
001010001110100100101010000001111
11100110000011101010010101101101
```

- Machine language seems quite simple on the surface, given that there are only zeros and ones, but its syntax is even more complex and far more intricate than Python.
- Instead we build various translators to allow programmers to write in high-level languages like Python or JavaScript and these translators convert the programs to machine language for actual execution by the CPU.
- Since machine language is tied to the computer hardware, machine language is not *portable* across different types of hardware.
- Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

These programming language translators fall into two general categories:
(1) interpreters
(2) compilers.

(1) Interpreters

- An *interpreter* reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly.
- Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.
- Some of the lines of Python tell Python that you want it to remember some value for later.
- We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later.
- We use the term *variable* to refer to the labels we use to refer to this stored data.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

- In this example, we ask Python to remember the value six and use the label *x* so we can retrieve the value later.
- We verify that Python has actually remembered the value using *print*.
- Then we ask Python to retrieve *x* and multiply it by seven and put the newly computed value in *y*.
- Then we ask Python to print out the value currently in *y*.
- Even though we are typing these commands into Python one line at a time, Python is treating them as an ordered sequence of statements with later statements able to retrieve data created in earlier statements.
- The Python interpreter is written in a high-level language called "C".

(2) Compilers.

- A *compiler* needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.

1.7 Writing a program

- When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a *script*.
- By convention, Python scripts have names that end with .py.
- To execute the script, you have to tell the Python interpreter the name of the file.
- In a Unix or Windows command window, you would type python hello.py as follows:

```
csev$ cat hello.py
print('Hello world!')
csev$ python hello.py
Hello world!
csev$
```

- The “csev$” is the operating system prompt, and the “cat hello.py” is showing us that the file “hello.py” has a one-line Python program to print a string.
- We call the Python interpreter and tell it to read its source code from the file “hello.py” instead of prompting us for lines of Python code interactively.

1.8 What is a program?

- The definition of a *program* at its most basic is a sequence of Python statements that have been crafted to do something. Even our simple *hello.py* script is a program.
- It is a one-line program and is not particularly useful, but in the strictest definition, it is a Python program.
- For example, look at the following text about a clown and a car.
- Look at the text and figure out the most common word and how many times it occurs.

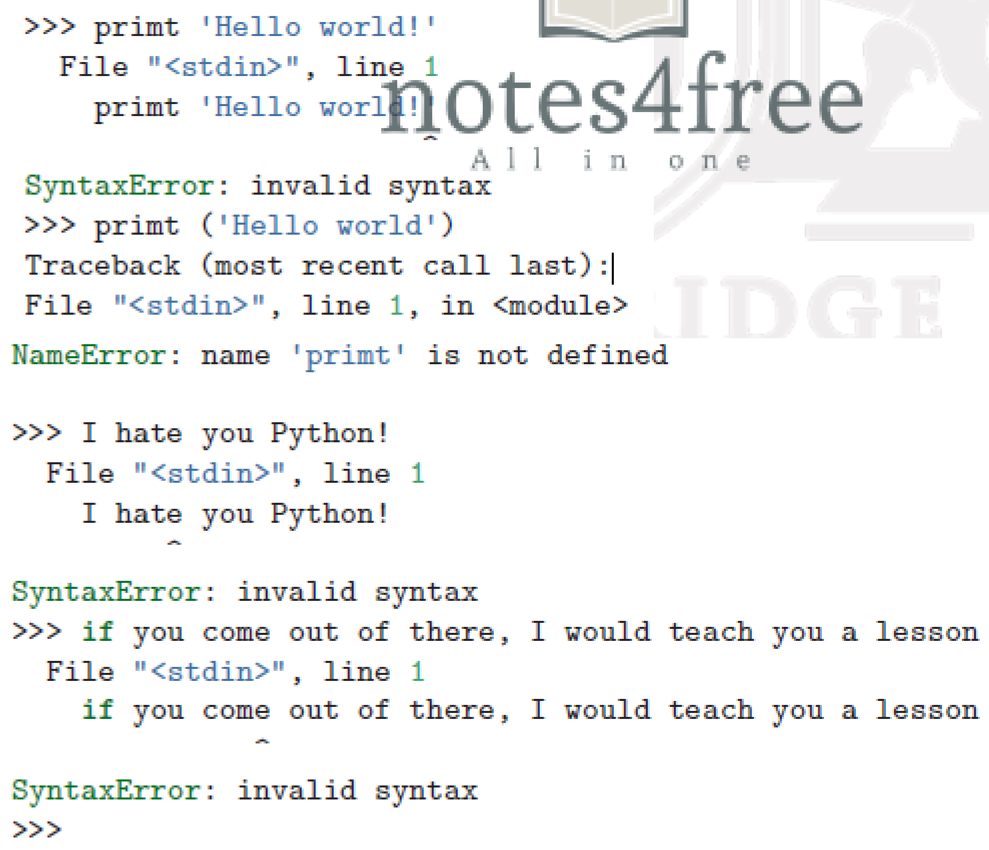
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car

```
name = input('Enter file:')
handle = open(name, 'r')
counts = dict()

for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

1.9 The building blocks of programs

- **input** Get data from the “outside world”. This might be reading data from a file, or even some kind of sensor like a microphone or GPS. In our initial programs, our input will come from the user typing data on the keyboard.
- **output** Display the results of the program on a screen or store them in a file or perhaps write them to a device like a speaker to play music or speak text.
- **sequential execution** Perform statements one after another in the order they are encountered in the script.
- **conditional execution** Check for certain conditions and then execute or skip a sequence of statements.
- **repeated execution** Perform some set of statements repeatedly, usually with some variation.
- **reuse** Write a set of instructions once and give them a name and then reuse those instructions as needed throughout your program.

1.10 What could possibly go wrong?



```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt ('Hello world')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined

>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
              ^
SyntaxError: invalid syntax
>>>
```

You will encounter three general types of errors:

Syntax errors

- These are the first errors you will make and the easiest to fix.
- A syntax error means that you have violated the “grammar” rules of Python.
- Python does its best to point right at the line and character where it noticed it was confused.
- The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python *noticed* it was confused.
- So the line and character that Python indicates in a syntax error may just be a starting point for your investigation.

Logic errors

- A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another.
- A good example of a logic error might be, “take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle.”

Semantic errors

- A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program.
- The program is perfectly correct but it does not do what you *intended* for it to do

Chapter 2
Variables,expressions, and statements

2.1 Values and types

- A *value* is one of the basic things a program works with, like a letter or a number.
- The values we have seen so far are 1, 2, and “Hello, World!”
- These values belong to different *types*: 2 is an integer, and “Hello, World!” is a *string*, so called because it contains a “string” of letters.
- The print statement also works for integers.
- We use the python command to start the interpreter.

```
python
>>> print(4)
4
```

- If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

- Strings belong to the type str and integers belong to the type int.
- Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called *floating point*.

```
>>> type(3.2)
<class 'float'>

>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

- When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000.
- This is not a legal integer in Python, but it is legal:

```
>>> print(1,000,000)
1 0 0
```

- Python interprets 1,000,000 as a comma separated sequence of integers, which it prints with spaces between.
- This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

2.2 Variables

- A variable is a name that refers to a value.
- An *assignment statement* creates new variables and gives them values:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

- This example makes three assignments.
- The first assigns a string to a new variable named message;
- the second assigns the integer 17 to n;
- the third assigns the (approximate) value of _ to pi.

To display the value of a variable, you can use a print statement:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

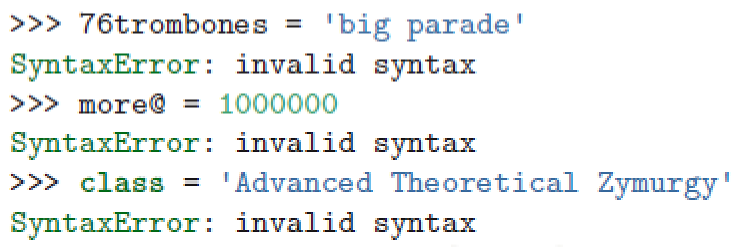
```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

2.3 Variable names and keywords

- Programmers generally choose names for their variables that are meaningful and document what the variable is used for.
- Variable names can be arbitrarily long.

- They can contain both letters and numbers, but they cannot start with a number.
- It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.
- The underscore character (_) can appear in a name. It is often used in names with multiple words, such as my_name or airspeed_of_unladen_swallow.
- Variable names can start with an underscore character, but we generally avoid doing this unless we are writing library code for others to use.

  If you give a variable an illegal name, you get a syntax error:



```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

- 76trombones is illegal because it begins with a number. more@ is illegal because it contains an illegal character, @.
- The class is one of Python's *keywords*. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

2.4 Statements

- A *statement* is a unit of code that the Python interpreter can execute.
- We have seen two kinds of statements:
  - print being an expression statement and assignment.
- When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.
  - A script usually contains a sequence of statements.
- If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

The assignment statement produces no output.

2.5 Operators and operands

- *Operators* are special symbols that represent computations like addition and multiplication.
- The values the operator is applied to are called *operands*.
- The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

```
20+32    hour-1    hour*60+minute    minute/60    5**2        (5+9)*(15-7)
```

- There has been a change in the division operator between Python 2.x and Python 3.x.
- In Python 3.x, the result of this division is a floating point result:
  ```
  >>> minute = 59
  >>> minute/60
  0.9833333333333333
  ```
- The division operator in Python 2.0 would divide two integers and truncate the result to an integer:
  ```
  >>> minute = 59
  >>> minute/60
  0
  ```
- To obtain the same answer in Python 3.0 use floored (// integer) division.

```
>>> minute = 59
>>> minute//60
0
```

2.6 Expressions

- An *expression* is a combination of values, variables, and operators.
- A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions

```
17
x
x + 17
```

- If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

```
>>> 1 + 1
2
```

2.7 Order of operations

- When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*.
- For mathematical operators, Python follows mathematical convention.
- The acronym *PEMDAS* is a useful way to remember the rules:
- ***Parentheses***
  - have the highest precedence
  - and can be used to force an expression to evaluate in the order you want.
  - Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8.
  - You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.
- ***E*xponentiation** has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
- ***M*ultiplication and *D*ivision** have the same precedence, which is higher than ***A*ddition and *S*ubtraction**, which also have the same precedence.
- So 2*3-1 is 5, not 4, and 6+4/2 is 8.0, not 5.

- Operators with the same precedence are evaluated from left to right.
- So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

2.8 Modulus operator

- The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second.
- In Python, the modulus operator is a percent sign (%).
- The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

- So 7 divided by 3 is 2 with 1 left over.
- The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if x % y is zero, then x is divisible by y.
- You can also extract the right-most digit or digits from a number.
- For example, x % 10 yields the right-most digit of x (in base 10).
- Similarly, x % 100 yields the last two digits.

2.9 String operations

- The + operator works with strings, but it is not addition in the mathematical sense.
- Instead it performs *concatenation*, which means joining the strings by linking them end to end.
- For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
```

- The output of this program is 100150.

2.10 Asking the user for input

- Sometimes we would like to take the value for a variable from the user via their keyboard.
- Python provides a built-in function called input that gets input from the keyboard.
- When this function is called, the program stops and waits for the user to type something.
- When the user presses Return or Enter, the program resumes and input returns what the user typed as a string.
- In Python 2.0, this function was named raw_input.

```
>>> input = input()
Some silly stuff
>>> print(input)
Some silly stuff
```

- Before getting input from the user, it is a good idea to print a prompt telling the user what to input.
- You can pass a string to input to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Chuck
>>> print(name)
Chuck
```

- The sequence \n at the end of the prompt represents a *newline*, which is a special character that causes a line break.
- That's why the user's input appears below the prompt.
- If you expect the user to type an integer, you can try to convert the return value to int using the int() function:

```
>>> prompt = 'What...is the airspeed velocity of an unladen
    swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

- But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

2.11 Comments

- ➢ As programs get bigger and more complicated, they get more difficult to read.
- ➢ Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
- ➢ For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing.
- ➢ These notes are called *comments*, and in Python they start with the # symbol:

```
# compute the percentage of the hour that has elapsed
        percentage = (minute * 100) / 60
```

- In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
        percentage = (minute * 100) / 60 # percentage of an hour
```

- Everything from the \# to the end of the line is ignored; it has no effect on the program.
- Comments are most useful when they document non-obvious features of the code.
- It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*.
- This comment is redundant with the code and useless:

```
v = 5 # assign 5 to v
```

- This comment contains useful information that is not in the code:

```
v = 5 # velocity in meters/second.
```

- Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.

2.12 Choosing mnemonic variable names

- ➢ As long as you follow the simple rules of variable naming, and avoid reserved words, you have a lot of choice when you name your variables.
- ➢ In the beginning, this choice can be confusing both when you read a program and when you write your own programs.

- For example, the following three programs are identical in terms of what they accomplish, but very different when you read them and try to understand them.

```
a = 35.0
b = 12.50
c = a * b
print(c)

hours = 35.0
rate = 12.50
pay = hours * rate
print(pay)

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

- The Python interpreter sees all three of these programs as *exactly the same* but humans see and understand these programs quite differently.
- Humans will most quickly understand the *intent* of the second program because the programmer has chosen variable names that reflect their intent regarding what data will be stored in each variable.
- We call these wisely chosen variable names “mnemonic variable names”.
- The word *mnemonic* means “memory aid”.
- We choose mnemonic variable names to help us remember why we created the variable in the first place.

2.13 Debugging

- At this point, the syntax error you are most likely to make is an illegal variable name, like class and yield, which are keywords, or odd~job and US$, which contain illegal characters.
- If you put a space in a variable name, Python thinks it is two operands without an operator:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
File "<stdin>", line 1
month = 09
```

```
^
SyntaxError: invalid token
```

- For syntax errors, the error messages don't help much. The most common messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.
- The runtime error you are most likely to make is a "use before def;" that is, trying to use a variable before you have assigned a value.
- This can happen if you spell a variable name wrong:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

- Variables names are case sensitive, so LaTeX is not the same as latex.
- At this point, the most likely cause of a semantic error is the order of operations.
- For example, to evaluate 1/2_, you might be tempted to write

```
>>> 1.0 / 2.0 * pi
```

- But the division happens first, so you would get _/2, which is not the same thing!
- There is no way for Python to know what you meant to write, so in this case you don't get an error message; you just get the wrong answer.

Chapter 3
Conditional execution

3.1 Boolean expressions

- A *boolean expression* is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
{}
```

- True and False are special values that belong to the class bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

- The == operator is one of the *comparison operators*; the others are:

```
x != y        # x is not equal to y
x > y         # x is greater than y
x < y         # x is less than y
x >= y        # x is greater than or equal to y
x <= y        # x is less than or equal to y
x is y        # x is the same as y
x is not y    # x is not the same as y
```

- Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations.
- A common error is to use a single equal sign (=) instead of a double equal sign (==).
- Remember that = is an assignment operator and == is a comparison operator.
- There is no such thing as =< or =>.

3.2 Logical operators

- There are three *logical operators*: and, or, and not.
- The semantics (meaning) of these operators is similar to their meaning in English.

- For example, x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.
- n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.
- Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.
- Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict.
- Any nonzero number is interpreted as "true."

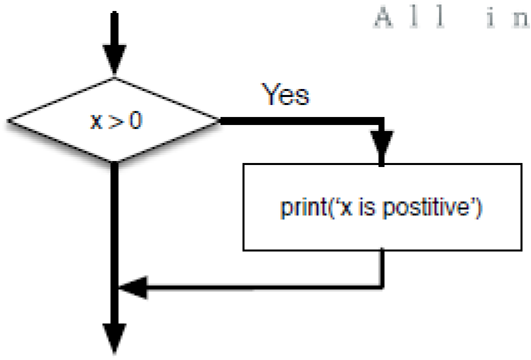
```
>>> 17 and True
True
```

3.3 Conditional execution

- In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
- *Conditional statements* give us this ability.
- The simplest form is the if statement:

```
if x > 0 :
print('x is positive')
```

- The boolean expression after the if statement is called the *condition*.
- We end the if statement with a colon character (:) and the line(s) after the if statement are indented.



- If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.
- if statements have the same structure as function definitions or for loops
- The statement consists of a header line that ends with the colon character (:) followed by an indented block.
- Statements like this are called *compound statements* because they stretch across more than one line.

- If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

```
>>> x = 3
>>> if x < 10:
... print('Small')
...
Small
>>>
```

3.4 Alternative execution

- A second form of the if statement is *alternative execution*, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x%2 == 0 :
        print('x is even')
else :
        print('x is odd')
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect.
- If the condition is false, the second set of statements is executed.
- Since the condition must either be true or false, exactly one of the alternatives will be executed.
- The alternatives are called *branches*, because they are branches in the flow of execution.

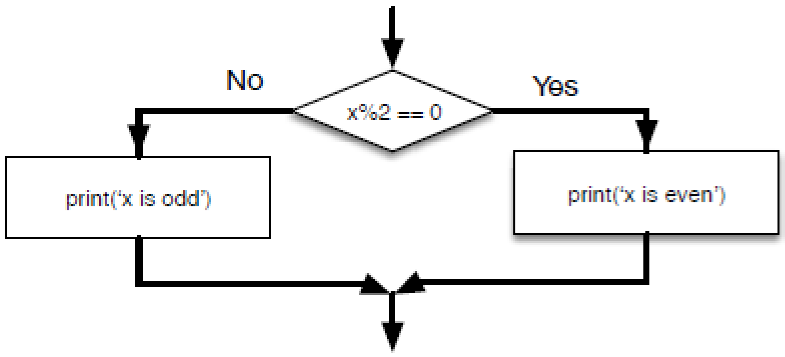


Figure 3.2: If-Then-Else Logic

3.5 Chained conditionals

- Sometimes there are more than two possibilities and we need more than two branches.
- One way to express a computation like that is a *chained conditional*:

```
if x < y:
        print('x is less than y')
elif x > y:
        print('x is greater than y')
else:
        print('x and y are equal')
```

- elif is an abbreviation of “else if.” Again, exactly one branch will be executed.

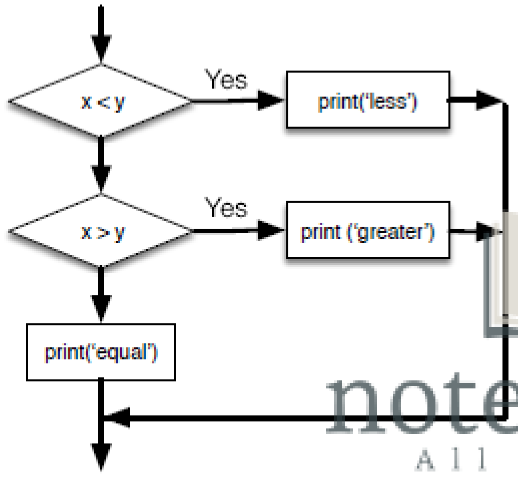


Figure 3.3: If-Then-ElseIf Logic

- There is no limit on the number of elif statements. If there is an else clause, it has to be at the end

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

- Each condition is checked in order. If the first is false, the next is checked, and so on.
- If one of them is true, the corresponding branch executes, and the statement ends.
- Even if more than one condition is true, only the first true branch executes.

3.6 Nested conditionals

- One conditional can also be nested within another. We could have written the three-branch example like this:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

- The outer conditional contains two branches. The first branch contains a simple statement.
- The second branch contains another if statement, which has two branches of its own.
- Those two branches are both simple statements, although they could have been conditional statements as well.

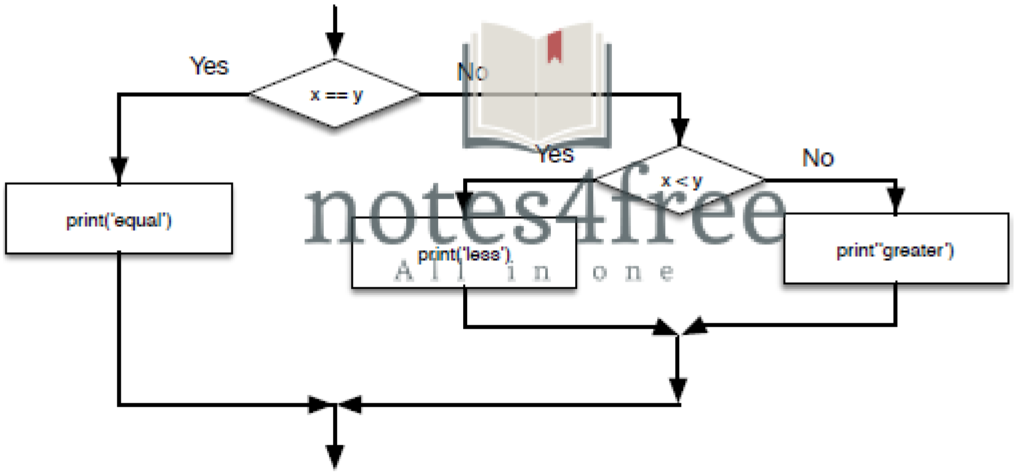


Figure 3.4: Nested If Statements

- Logical operators often provide a way to simplify nested conditional statements.
- For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

- The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

- The print statement is executed only if we make it past both conditionals, so we

can get the same effect with the and operator:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

3.7 Catching exceptions using try and except

- Here is a sample program to convert a Fahrenheit temperature to a Celsius temperature:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)
```

- If we execute this code and give it invalid input, it simply fails with an unfriendly error message:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

- There is a conditional execution structure built into Python to handle these types of expected and unexpected errors called "try / except".
- The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs.
- These extra statements (the except block) are ignored if there is no error.

- We can rewrite our temperature converter as follows:

```
inp = input('Enter Fahrenheit Temperature:')
try:
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
```

```
print(cel)
except:
print('Please enter a number')
```

- Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds.
- If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

- Handling an exception with a try statement is called *catching* an exception.
- In this example, the except clause prints an error message.
- In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

3.8 Short-circuit evaluation of logical expressions

- When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right.
- Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.
- When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression.
- When the evaluation of a logical expression stops because the overall value is already known, it is called *short-circuiting* the evaluation.
- The short-circuit behavior leads to a clever technique called the *guardian pattern*.
- Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

- The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error.
- But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit* rule and there was no error.
- We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

- In the first logical expression, x >= 2 is False so the evaluation stops at the and.
- In the second logical expression, x >= 2 is True but y != 0 is False so we never reach (x/y).
- In the third logical expression, the y != 0 is *after* the (x/y) calculation so the expression fails with an error.
- In the second expression, we say that y != 0 acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

Chapter 4
Functions

4.1 Function calls

- In the context of programming, a *function* is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can "call" the function by name.

```
>>> type(32)
<class 'int'>
```

- The name of the function is type.
- The expression in parentheses is called the *argument* of the function.
- The argument is a value or variable that we are passing into the function as input to the function.
- The result, for the type function, is the type of the argument.
- It is common to say that a function "takes" an argument and "returns" a result.
- The result is called the *return value*.

4.2 Built-in functions

- Python provides a number of important built-in functions that we can use without needing to provide the function definition.
- The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.
- The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

- The max function tells us the "largest character" in the string (which turns out to be the letter "w") and the min function shows us the smallest character (which turns out to be a space).

- Another very common built-in function is the len function which tells us how many items are in its argument.
- If the argument to len is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

- These functions are not limited to looking at strings. They can operate on any set of values
- You should treat the names of built-in functions as reserved words (i.e., avoid using “max” as a variable name).

4.3 Type conversion functions

- Python also provides built-in functions that convert values from one type to another.
- The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

- int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- float converts integers and strings to floating-point numbers:

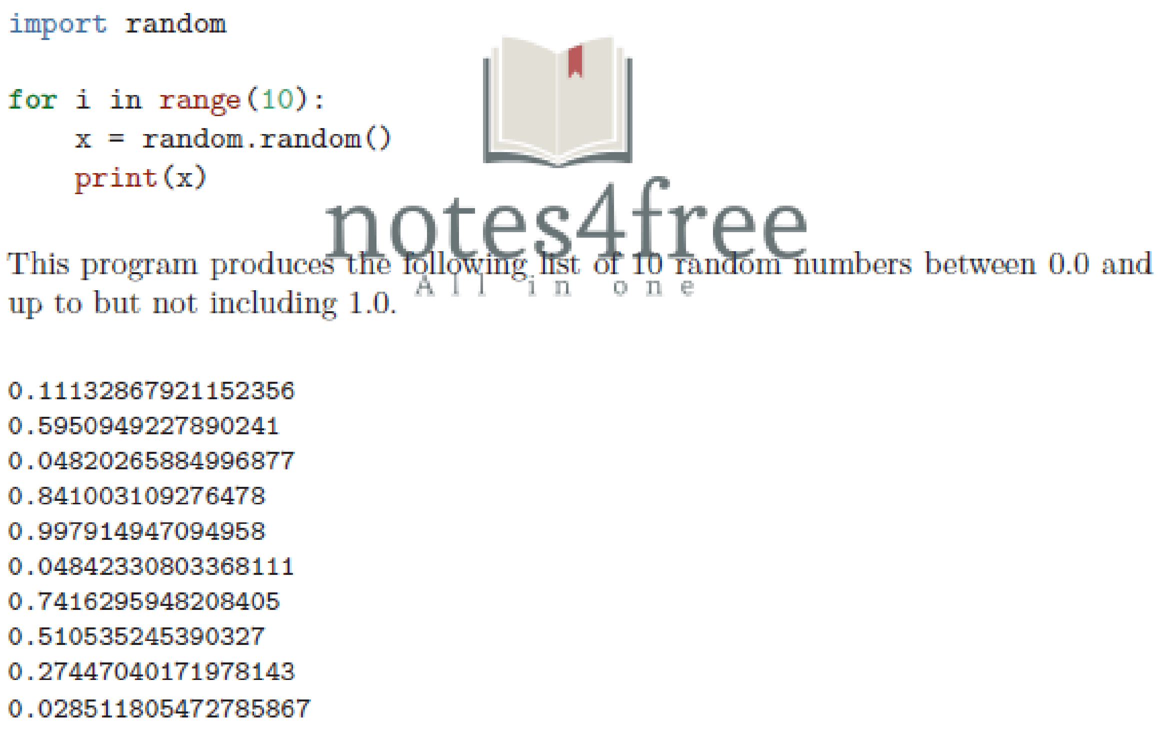
```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Random numbers

- Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*.
- For some applications, though, we want the computer to be unpredictable.
- Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic.
- One of them is to use *al- gorithms* that generate *pseudorandom* numbers.
- Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.
- The random module provides functions that generate pseudorandom numbers
- The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0).
- Each time you call random, you get the next number in a long series.



```
import random

for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

- The random function is only one of many functions that handle random numbers.
- The function randint takes the parameters low and high, and returns an integer between low and high (including both).

```
>>> random.randint(5, 10)
5
```

```
>>> random.randint(5, 10)
9
```

- To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

- The random module also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.

4.5 Math functions

- Python has a math module that provides most of the familiar mathematical functions.
- Before we can use the module, we have to import it:

```
>>> print(math)
<module 'math' (built-in)>
```

- The module object contains the functions and variables defined in the module.
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period).
- This format is called *dot notation*.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

- The first example computes the logarithm base 10 of the signal-to-noise ratio.
- The math module also provides a function called log that computes logarithms base e.
- The second example finds the sine of radians.
- The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians.
- To convert from degrees to radians, divide by 360 and multiply by 2_:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

- The expression math.pi gets the variable pi from the math module. The value of this variable is an approximation of _, accurate to about 15 digits.
- you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.6 Adding new functions

- So far, we have only been using the functions that come with Python, but it is also possible to add new functions.
- A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called.
- Once we define a function, we can reuse the function over and over throughout our program.
  Here is an example:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

- def is a keyword that indicates that this is a function definition.
- The name of the function is print_lyrics.
- The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number.
- You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.
- The empty parentheses after the name indicate that this function doesn't take any arguments.
- Later we will build functions that take arguments as their inputs.
- The first line of the function definition is called the *header*; the rest is called the *body*.
- The header has to end with a colon and the body has to be indented.
- By convention, the indentation is always four spaces.
- The body can contain any number of statements.
- The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.
- If you type a function definition in interactive mode, the interpreter prints ellipses (. . .) to let you know that the definition isn't complete:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

- To end the function, you have to enter an empty line
- Defining a function creates a variable with the same name.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

- The value of print_lyrics is a *function object*, which has type “function”.
- The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

- Once you have defined a function, you can use it inside another function.
- For example, to repeat the previous refrain, we could write a function called repeat_lyrics:

```
def repeat_lyrics():
print_lyrics()
print_lyrics()
```

And then call repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

But that’s not really how the song goes.

4.7 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
print("I'm a lumberjack, and I'm okay.")
print('I sleep all night and I work all day.')
def repeat_lyrics():
print_lyrics()
print_lyrics()
```

4.8. FLOW OF EXECUTION 49

```
repeat_lyrics()
# Code: http://www.py4e.com/code3/lyrics.py
```
This program contains two function definitions: print_lyrics and repeat_lyrics.
Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not get executed until the function is called, and the function definition generates no output.
As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.
Exercise 2: Move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.
Exercise 3: Move the function call back to the bottom and move the definition of print_lyrics after the definition of repeat_lyrics. What happens when you run this program?

4.8 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the *flow of execution*.
Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom.
Function *definitions* do not alter the flow of execution of the program, but remember
that statements inside the function are not executed until the function is called.
A function call is like a detour in the flow of execution. Instead of going to the next
statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
That sounds simple enough, until you remember that one function can call another.
While in the middle of one function, the program might have to execute the statements
in another function. But while executing that new function, the program might have to execute yet another function!
Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it.
When it gets to the end of the program, it terminates.
What's the moral of this sordid tale? When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

4.9 Parameters and arguments

Some of the built-in functions we have seen require arguments. For example, when you call math.sin you pass a number as an argument. Some functions take more than one argument: math.pow takes two, the base and the exponent.

Inside the function, the arguments are assigned to variables called *parameters*.

Here is an example of a user-defined function that takes an argument:

```
def print_twice(bruce):
print(bruce)
print(bruce)
```

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print_twice:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

The argument is evaluated before the function is called, so in the examples the expressions “Spam ’*4andmath.cos(math.pi)‘ are only evaluated once.

You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn’t matter what the value was called back home (in the caller); here in print_twice, we call everybody bruce.

4.10 Fruitful functions and void functions

Some of the functions we are using, such as the math functions, yield results; for lack of a better name, I call them *fruitful functions*. Other functions, like print_twice, perform an action but don't return a value. They are called *void functions*.

When you call a fruitful function, you almost always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

```
math.sqrt(5)
```

This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

The value None is not the same as the string "None". It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

To return a result from a function, we use the return statement in our function. For example, we could make a very simple function called addtwo that adds two numbers together and returns a result.

```
def addtwo(a, b):
added = a + b
return added
x = addtwo(3, 5)
```

```
print(x)
# Code: http://www.py4e.com/code3/addtwo.py
```

When this script executes, the print statement will print out “8” because the addtwo function was called with 3 and 5 as arguments. Within the function, the parameters a and b were 3 and 5 respectively. The function computed the sum of the two numbers and placed it in the local function variable named added. Then it used the return statement to send the computed value back to the calling code as the function result, which was assigned to the variable x and printed out.

4.11 Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

• Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.

• Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

• Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

• Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Throughout the rest of the book, often we will use a function definition to explain a concept. Part of the skill of creating and using functions is to have a function properly capture an idea such as “find the smallest value in a list of values”. Later we will show you code that finds the smallest in a list of values and we will present it to you as a function named min which takes a list of values as its argument and returns the smallest value in the list.

MODULE II

2.1 ITERATION

The while statement, Infinite loops, “Infinite loops” and break, Finishing iterations with Continue, Definite loops using for, Loop pattern ,Counting and summing loops, Maximum and minimum loops

2.2 STRINGS

A string is a sequence, Getting the length of a string using len, Traversal through a string with a loop, String slices, Strings are immutable, Looping and counting, The in operator, String comparison string methods, Parsing strings, Format operator

2.3 FILES

Files, Persistence, Opening files, Text files and lines, Reading files, Searching through a file, Letting the user choose the file name, Using try, except, and open, Writing files, Debugging

MODULE II

2.1 ITERATION

Iteration is a processing of repeating some task. In a real time programming, we require a set of statements to be repeated certain number of times and/or till a condition is met. Every programming language provides certain constructs to achieve the repetition of tasks. In this section, various such looping structures are discussed.

→ The *while* Statement

The *while* loop has the syntax as below –

```
while condition:
      statement_1
      statement_2
      …………….
      statement_n

statements_after_while
```

- Here, ***while*** is a keyword, the flow of execution for a while statement is as below.
- The condition is evaluated first, yielding True or False
- If the condition is false, the loop is terminated and statements after the loop will be executed.
- If the condition is true, the body will be executed which comprises of the statement_1 to statement_n and then goes back to condition evaluation.
- Consider an example –

```
n=1
while n<=5:
      print(n)          #observe indentation
      n=n+1
print("over")
```

The output of above code segment would be –

```
1
2
3
4
5
over
```

- In the above example, a variable n is initialized to 1. Then the condition n<=5 is being checked. As the condition is true, the block of code containing print statement print(n) and increment statement (n=n+1)are executed. After these two lines, condition is checked again. The procedure continues till condition becomes false, that is when n becomes 6. Now, the while-loop is terminated and next statement after the loop will be executed. Thus, in this example, the loop is ***iterated*** for 5 times.

- Consider another example –

```
n=5
while n>0:
        print(n)            #observe indentation
        n=n-1
print("Blast off!")
```

  The output of above code segment would be –

```
5
4
3
2
1
Blast off!
```

- Iteration is referred to each time of execution of the body of loop.
- Note that, a variable n is initialized before starting the loop and it is incremented/decremented inside the loop. Such a variable that changes its value for every iteration and controls the total execution of the loop is called as ***iteration variable*** or ***counter variable***. If the count variable is not updated properly within the loop, then the loop may not terminate and keeps executing infinitely.

→ Infinite Loops, *break* and *continue*

- A loop may execute infinite number of times when the condition is never going to become false.
- For example,

```
n=1
while True:
        print(n)
        n=n+1
```

- Here, the condition specified for the loop is the constant True, which will never get terminated. Sometimes, the condition is given such a way that it will never become false and hence by restricting the program control to go out of the loop. This situation may happen either due to wrong condition or due to not updating the counter variable.
- In some situations, we deliberately want to come out of the loop even before the normal termination of the loop. For this purpose ***break*** statement is used.
- The following example depicts the usage of *break***.** Here, the values are taken from keyboard until a negative number is entered. Once the input is found to be negative, the loop terminates.

```
while True:
        x=int(input("Enter a number:"))
         if x>= 0:
```

```
        print("You have entered ",x)
    else:
        print("You have entered a negative number!!")
        break               #terminates the loop
```

Output:

```
Enter a number:23
You have entered 23
Enter a number:12
You have entered 12
Enter a number:45
You have entered 45
Enter a number:0
You have entered 0
 Enter a number:-2
You have entered a negative number!!
```

- In the above example, we have used the constant True as condition for while-loop, which will never become false. So, there was a possibility of infinite loop. This has been avoided by using *break* statement with a condition.
- The condition is kept inside the loop such a way that, if the user input is a negative number, the loop terminates. This indicates that, the loop may terminate with just one iteration (if user gives negative number for the very first time) or it may take thousands of iteration (if user keeps on giving only positive numbers as input). Hence, the number of iterations here is unpredictable.
- But, we are making sure that it will not be an infinite-loop, instead, the user has control on the loop.
- Another example for usage of while with break statement: the below code takes input from the user until they type done:

```
while True:
   line = input(">")
   if line == 'done':
       break
   print(line)
print('Done!')
```

- In the above example, since the loop condition is True, so the loop runs repeatedly until it hits the break statement.
- Each time it prompts the user to enter the data. If the user types done, the brak statement exits the loop. Otherwise the program echoes whatever the user types and goes back ti the top of the loop.
- **Output will be:**

```
>hello
hello
>finished
```

```
finished
>done
Done!
```

- Sometimes, programmer would like to move to next iteration by skipping few statements in the loop, based on some condition with current iteration. For this purpose ***continue*** statement is used. For example, we would like to find the sum of 5 even numbers taken as input from the keyboard. The logic is –
  - Read a number from the keyboard
  - If that number is odd, without doing anything else, just move to next iteration for reading another number
  - If the number is even, add it to *sum* and increment the accumulator variable.
  - When accumulator crosses 5, stop the program
  - The program for the above task can be written as –

```
sum=0
count=0
while True:
        x=input("Enter a number:")
        if x%2!=0:
            continue
        else:
            sum+=x
            count+=1
        if count==5:
            break
print("Sum= ", sum)
```

Output:

```
Enter a number: 23
Enter a number: 67
Enter a number: 789
Enter a number: 78
Enter a number: 5
Enter a number: 7
Sum=  891
```

- Example of a loop that copies its input until the user types “done”, but treats lines that start with the hash character as lines not to be printed

```
while True:
  line=input('>')
  if line[0] == '#':
    continue
  if line =='done':
    break
  print(line)
print('Done!')
```

Output:

```
> hello there
hello there
> #dont print this
> print this!
print this!
> done
Done!
```

- Above, all lines are printed except the one that starts with '#' because whenth econtinue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

→ Definite Loops using *for*

- The *while* loop iterates till the condition is met and hence, the number of iterations are usually unknown prior to the loop. Hence, it is sometimes called as *indefinite loop*.
- When we know total number of times the set of statements to be executed, ***for*** loop will be used. This is called as a *definite loop***.** The for-loop iterates over a set of numbers, a set of words, lines in a file etc. The syntax of for-loop would be –

```
for var in list/sequence:
     statement_1
     statement_2
     ..................
     statement_n

statements_after_for
```

Here,	*for* and *in*	are keywords
	list/sequence	is a set of elements on which the loop is iterated. That is, the loop will be executed till there is an element in list/sequence
	statements	constitutes body of the loop

- **Example:** In the below given example, a ***list*** names containing three strings has been created. Then the counter variable x in the *for*-loop iterates over this *list*. The variable x takes the elements in names one by one and the body of the loop is executed.

```
names=["Ram", "Shyam", "Bheem"]
for x in names:
        print("Happy New Year",x)
        print('Done!')
```

The output would be –

```
Happy New Year Ram
Happy New Year Shyam
Happy New Year Bheem
Done!
```

NOTE: In Python, list is an important data type. It can take a sequence of elements of different types. It can take values as a comma separated sequence enclosed within square brackets. Elements in the list can be extracted using index (just similar to extracting array elements in C/C++ language). Various operations like indexing, slicing, merging, addition and deletion of elements etc. can be applied on lists. The details discussion on Lists will be done in Module 3.

- The *for* loop can be used to print (or extract) all the characters in a string as shown below –

```
for i in "Hello":
        print(i, end='\t')
```

Output:

```
H       e       l       l       o
```

- When we have a fixed set of numbers to iterate in a *for* loop, we can use a function ***range()***. The function *range()* takes the following format –

```
range(start, end, steps)
```

- The start and end indicates starting and ending values in the sequence, where end is excluded in the sequence (That is, sequence is up to end-1). The default value of start is 0. The argument steps indicates the increment/decrement in the values of sequence with the default value as 1. Hence, the argument steps is optional.
- Let us consider few examples on usage of *range()* function.

Ex1. Printing the values from 0 to 4 –

```
for i in range(5):
        print(i, end= '\t')
```

Output:

```
0       1       2       3       4
```

Here, 0 is the default starting value. The statement range(5)is same as range(0,5) and range(0,5,1).

Ex2. Printing the values from 5 to 1 –

```
for i in range(5,0,-1):
        print(i, end= '\t')
```

Output:

```
5       4       3       2       1
```

The function range(5,0,-1)indicates that the sequence of values are 5 to 0(excluded) in steps of -1 (downwards).

Ex3. Printing only even numbers less than 10 –

```
for i in range(0,10,2):
        print(i, end= '\t')
```

Output:

```
0       2       4       6       8
```

→ Loop Patterns

The *while*-loop and *for*-loop are usually used to go through a list of items or the contents of a file and to check maximum or minimum data value. These loops are generally constructed by the following procedure –

- Initializing one or more variables before the loop starts
- Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop
- Looking at the resulting variables when the loop completes

The construction of these loop patterns are demonstrated in the following examples.

Counting and Summing Loops: One can use the *for* loop for counting number of items in the list as shown –

```
count = 0
for i in [4, -2, 41, 34, 25]:
       count = count + 1
print("Count:", count)
```

- Here, the variable count is initialized before the loop. Though the counter variable is not being used inside the body of the loop, it controls the number of iterations.
- The variable count is incremented in every iteration, and at the end of the loop the total number of elements in the list is stored in it.
- One more loop similar to the above is finding the sum of elements in the list –

```
total = 0
for x in [4, -2, 41, 34, 25]:
       total = total + x
print("Total:", total)
```

- Here, the variable total is called as ***accumulator*** because in every iteration, it accumulates the sum of elements. In each iteration, this variable contains *running total of values so far*.

NOTE: In practice, both of the counting and summing loops are not necessary, because there are built-in functions len()and sum()for the same tasks respectively.

Maximum and Minimum Loops: To find maximum element in the list, the following code can be used –

```
big = None
print('Before Loop:', big)
for x in [12, 0, 21,-3]:
      if big is None or x > big :
            big = x
      print('Iteration Variable:', x, 'Big:', big)
print('Biggest:', big)
```

Output:

```
Before Loop: None
Iteration Variable: 12          Big: 12
Iteration Variable: 0           Big: 12
Iteration Variable: 21          Big: 21
Iteration Variable: -3          Big: 21
Biggest: 21
```

- Here, we initialize the variable big to None. It is a special constant indicating empty.
- Hence, we cannot use relational operator == while comparing it with big. Instead, the ***is*** operator must be used.
- In every iteration, the counter variable x is compared with previous value of big. If x > big, then x is assigned to big.
- Similarly, one can have a loop for finding smallest of elements in the list as given below –

```
small = None
print('Before Loop:', small)
for x in [12, 0, 21,-3]:
      if small is None or x < small :
            small = x
      print('Iteration Variable:', x, 'Small:', small)
print('Smallest:', small)
```

Output:

```
Before Loop: None
Iteration Variable: 12     Small: 12
Iteration Variable: 0       Small: 0
Iteration Variable: 21      Small: 0
Iteration Variable: -3      Small: -3
Smallest: -3
```

NOTE: In Python, there are built-in functions max() and min()to compute maximum and minimum values among. Hence, the above two loops need not be written by the programmer explicitly. The inbuilt function min()has the following code in Python –

```
def min(values):
        smallest = None
        for value in values:
                if smallest is None or value < smallest:
                 smallest = value
        return smallest
```

2.2 STRINGS

- A string is a sequence of characters, enclosed either within a pair of single quotes or double quotes.
- Each character of a string corresponds to an index number, starting with zero as shown below:

S= “Hello World”

character	H	e	l	l	o		w	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

- The characters of a string can be accessed using index enclosed within square brackets.
- So, H is the 0th letter, e is the 1th letter and l is the 2th letter of “Hello world”
- For example,

```
>>> word1="Hello"
>>> word2='hi'
>>> x=word1[1]                    #2nd character of word1 is extracted
>>> print(x)
 e
>>> y=word2[0]                    #1st character of word1 is extracted
>>> print(y)
 h
```

- Python supports negative indexing of string starting from the end of the string as shown below:

S= “Hello World”

character	H	e	l	l	o		w	o	r	l	d
Negative index	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- The characters can be extracted using negative index also, which count backward from the end of the string.
- For example:

```
>>> var=“Hello”
>>> print(var[-1])
 o
>>> print(var[-4])
 e
```

- Whenever the string is too big to remember last positive index, one can use negative index to extract characters at the end of string.

→ Getting Length of a String using *len()*

- The *len()* function is a built-in function that can be used to get length of a string, which returns the number of characters in a string
- Example:

```
>>> var="Hello"
>>> ln=len(var)
>>> print(ln)
      5
```

- The index for string varies from 0 to length-1. Trying to use the index value beyond this range generates error.

```
>>> var="Hello"
>>> ln=len(var)
>>> ch=var[ln]
IndexError: string index out of range
```

→ Traversal through String with a Loop

- Extracting every character of a string one at a time and then performing some action on that character is known as *traversal.*
- A string can be traversed either using *while* loop or using *for* loop in different ways. Few of such methods is shown here –

❖ **Using *for* loop:**

```
st="Hello"
for i in st:
        print(i, end='\t')
```

Output:

```
H       e       l       l       o
```

- In the above example, the *for* loop is iterated from first to last character of the string st. That is, in every iteration, the counter variable i takes the values as H, e, l, l and o. The loop terminates when no character is left in st.

❖ **Using *while* loop:**

```
st="Hello"
i=0
while i<len(st):
        print(st[i], end='\t')
        i+=1
```

Output:

```
H	e	l	l	o
```

- In this example, the variable i is initialized to 0 and it is iterated till the length of the string. In every iteration, the value of i is incremented by 1 and the character in a string is extracted using i as index.
- Example: Write a while loop that starts at the last character in the string and traverses backwards to the first character in the string, printing each letter on separate line

```
str="Hello"
i=-1
while i>=-len(str):
   print(str[i])
   i-=1
```

Output:

```
o
l
l
e
H
```

→ String Slices

- A segment or a portion of a string is called as *slice*.
- Only a required number of characters can be extracted from a string using colon (:) symbol.
- The basic syntax for slicing a string would be – st[i:j:k]
- This will extract character from i^{th} character of st till $(j-1)^{th}$ character in steps of k.
- If first index is not present, it means that slice should start from the beginning of the string. I
- f the second index j is not mentioned, it indicates the slice should be till the end of the string.
- The third parameter k, also known as ***stride***, is used to indicate number of steps to be incremented after extracting first character. The default value of stride is 1.
- Consider following examples along with their outputs to understand string slicing.

```
st="Hello World"                #refer this string for all examples
```

1. print("st[:] is", st[:]) **#output Hello World**

As both index values are not given, it assumed to be a full string.

2. print("st[0:5] is ", st[0:5]) **#output is Hello**

Starting from 0th index to 4th index (5 is exclusive), characters will be printed.

3. print("st[0:5:1] is", st[0:5:1]) **#output is Hello**

This code also prints characters from 0th to 4th index in the steps of 1. Comparing this example with previous example, we can make out that when the stride value is 1, it is optional to mention.

4. print("st[3:8] is ", st[3:8]) **#output is lo Wo**

Starting from 3rd index to 7th index (8 is exclusive), characters will be printed.

5. print("st[7:] is ", st[7:]) **#output is orld**

Starting from 7th index to till the end of string, characters will be printed.

6. print(st[::2]) **#output is HloWrd**

This example uses stride value as 2. So, starting from first character, every alternative character (char+2) will be printed.

7. print("st[4:4] is ", st[4:4]) **#gives empty string**

Here, st[4:4] indicates, slicing should start from 4th character and end with (4-1)=3rd character, which is not possible. Hence the output would be an empty string.

8. print(st[3:8:2]) #output is l o

Starting from 3rd character, till 7th character, every alternative index is considered.

9. print(st[1:8:3]) **#output is eoo**

Starting from index 1, till 7th index, every 3rd character is extracted here.

10. print(st[-4:-1]) **#output is orl**

Refer the diagram of negative indexing given earlier. Excluding the -1st character, all characters at the indices -4, -3 and -2 will be displayed. Observe the role of stride with default value 1 here. That is, it is computed as -4+1 =-3, -3+1=-2 etc.

11. print(st[-1:]) **#output is d**

Here, starting index is -1, ending index is not mentioned (means, it takes the index
10) and the stride is default value 1. So, we are trying to print characters from -1 (which is the last character of negative indexing) till 10th character (which is also the last character in positive indexing) in incremental order of 1. Hence, we will get only last character as output.

12. print(st[:-1]) **#output is Hello Worl**

Here, starting index is default value 0 and ending is -1 (corresponds to last character in negative indexing). But, in slicing, as last index is excluded always, -1st character is omitted and considered only up to -2nd character.

13. print(st[::]) **#outputs Hello World**

Here, two colons have used as if stride will be present. But, as we haven't mentioned stride its default value 1 is assumed. Hence this will be a full string.

14. print(st[::-1]) **#output is dlroW olleH**

This example shows the power of slicing in Python. Just with proper slicing, we could able to ***reverse the string***. Here, the meaning is *a full string to be extracted in the order of -1*. Hence, the string is printed in the reverse order.

15. print(st[::-2]) **#output is drWolH**

Here, the string is printed in the reverse order in steps of -2. That is, every alternative character in the reverse order is printed. Compare this with example (6) given above.

By the above set of examples, one can understand the power of string slicing and of Python script. The slicing is a powerful tool of Python which makes many task simple pertaining to data types like strings, Lists, Tuple, Dictionary etc. (Other types will be discussed in later Modules)

→ Strings are Immutable

- The objects of string class are immutable.
- That is, once the strings are created (or initialized), they cannot be modified.
- No character in the string can be edited/deleted/added.
- Instead, one can create a new string using an existing string by imposing any modification required.
- Try to attempt following assignment –

```
>>> st= "Hello World"
>>> st[3]='t'
TypeError: 'str' object does not support item assignment
```

- The error message clearly states that an assignment of new *item* ('t') is not possible on string object(st).
- The reason for this is strings are immutable
- So, to achieve our requirement, we can create a new string using slices of existing string as below

```
>>> st= "Hello World"
>>> st1= st[:3]+ 't' + st[4:]
>>> print(st1)
        Helto World                  # l is replaced by t in new string st1
```

→ Looping and Counting

- Using loops on strings, we can count the frequency of occurrence of a character within another string.
- The following program demonstrates such a pattern on computation called as a ***counter***.
- Initially, we accept one string and one character (single letter). Our aim to find the total number of times the character has appeared in string.
- A variable *count* is initialized to zero, and incremented each time 'a' character is found. The program is given below –

```
word="banana"
count=0
for letter in word:
    if letter =='a':
        count=count+1
print("The occurences of character 'a' is %d "%(count))
```

Output:

```
The occurences of character 'a' is 3
```

- Encapsulate the above code in a function named count and generalize it so that it accepts the string and the letter as arguments

```
def count(st,ch):
    cnt=0
    for i in st:
        if i==ch:
            cnt+=1
    return cnt

st=input("Enter a string:")
ch=input("Enter a character to be counted:")
c=count(st,ch)
print("%s appeared %d times in %s"%(ch,c,st))
```

Output:

```
Enter a string: hello how are you?
Enter a character to be counted: h
h appeared 2 times in hello how are you?
```

→ The *in* Operator

- The ***in*** operator of Python is a Boolean operator which takes two string operands.
- It returns True, if the first operand appears as a substring in second operand, otherwise returns False.

- For example,

```
>>> 'el' in 'hello'                    #el is found in hello
 True
>>> 'x' in 'hello'                     #x is not found in hello
False
```

→ String Comparison

- Basic comparison operators like < (less than), > (greater than), == (equals) etc. can be applied on string objects.
- Such comparison results in a Boolean value True or False.
- Internally, such comparison happens using ASCII codes of respective characters.
- Consider following examples –

Ex1.
```
st= "hello"
if st== 'hello':
        print('same')
```

Output is same. As the value contained in st and hello both are same, the equality results in True.

Ex2.
```
st= "hello"
if st<= 'Hello':
            print('lesser')
else:
        print('greater')
```

Output is greater. The ASCII value of **h** is greater than ASCII value of **H**. Hence, hello is greater than Hello.

NOTE: A programmer must know ASCII values of some of the basic characters. Here are few –

A – Z	: 65 – 90
a – z	: 97 – 122
0 – 9	: 48 – 57
Space	: 32
Enter Key	: 13

→ String Methods

- String is basically a ***class*** in Python.
- When we create a string in program, an ***object*** of that class will be created.
- A class is a collection of member variables and member methods (or functions).

- When we create an object of a particular class, the object can use all the members (both variables and methods) of that class.
- Python provides a rich set of built-in classes for various purposes. Each class is enriched with a useful set of utility functions and variables that can be used by a Programmer.
- A programmer can create a class based on his/her requirement, which are known as user-defined classes.
- The built-in set of members of any class can be accessed using the dot operator as shown–

  objName.memberMethod(arguments)
- The dot operator always binds the member name with the respective object name. This is very essential because, there is a chance that more than one class has members with same name. To avoid that conflict, almost all Object oriented languages have been designed with this common syntax of using dot operator.
- Python provides a function (or method) ***dir*** to list all the variables and methods of a particular class object. Observe the following statements –

```
>>> s="hello" # string object is created with the name s
>>> type(s)    #checking type of s
<class 'str'>    #s is object of type class str
>>> dir(s)        #display all methods and variables of object s
```

```
['_add_', '_class_', '_contains_', '_delattr_', '_dir_',  '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_',
'_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_', '_init_subclass_',  '_iter_',  '_le_',  '_len_',  '_lt__',
'_mod_',  '_mul_',  '_ne_',  '_new_',  '_reduce_',  '_reduce_ex_', '_repr_', '_rmod_', '_rmul_', '_setattr
', '_sizeof_', '_str_',  '__subclasshook__',  'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs',  'find',  'format',  'format_map',  'index',  'isalnum',  'isalpha',  'isdecimal',   'isdigit',
'isidentifier',  'islower',  'isnumeric', 'isprintable',   'isspace',     'istitle', 'isupper',     'join', 'ljust',
'lower',        'lstrip',          'maketrans',          'partition',          'replace',          'rfind',
'rindex',  'rjust',  'rpartition',  'rsplit',  'rstrip',  'split',  'splitlines',  'startswith',  'strip',  'swapcase',  'title',
'translate', 'upper', 'zfill']
```

- Note that, the above set of variables and methods are common for any object of string class that we create.
- Each built-in method has a predefined set of arguments and return type.
- To know the usage, working and behavior of any built-in method, one can use the command ***help***.
- For example, if we would like to know what is the purpose of islower() function (refer above list to check its existence!!), how it behaves etc, we can use the statement –

```
>>> help(str.islower)
Help on method_descriptor:

islower(...)
```

S.islower() -> bool

Return True if all cased characters in S are lowercase and if there is at least one upper cased character in S, returns False otherwise.

- This is built-in help-service provided by Python. Observe the className.memberName format while using ***help***.
- The methods are usually called using the object name. This is known as ***method invocation.*** We say that a method is invoked using an object.
- Now, we will discuss some of the important methods of string class.

❖ **capitalize(s) :** This function takes one string argument *s* and returns a capitalized version of that string. That is, the first character of *s* is converted to upper case, and all other characters to lowercase. Observe the examples given below –

Ex1.
```
>>> s="hello"
>>> s1=str.capitalize(s)
>>> print(s1)
        Hello                #1st character is changed to uppercase
```

Ex2.
```
>>> s="hello World"
>>> s1=str.capitalize(s)
>>> print(s1)
        Hello world
```

Observe in Ex2 that the first character is converted to uppercase, and an in-between uppercase letter W of the original string is converted to lowercase.

❖ **s.upper():** This function returns a copy of a string s to uppercase. As strings are immutable, the original string s will remain same.

```
>>> st= “hello”
>>> st1=st.upper()
>>> print(st1)
            'HELLO'
>>> print( st)              #no change in original string
      'hello'
```

❖ **s.lower():** This method is used to convert a string s to lowercase. It returns a copy of original string after conversion, and original string is intact.

```
>>> st='HELLO'
>>> st1=st.lower()
>>> print(st1) hello
>>> print(st)                    #no change in original string
    HELLO
```

- **s.find(s1) :** The find() function is used to search for a substring s1 in the string s. If found, the index position of first occurrence of s1 in s, is returned. If s1 is not found in s, then -1 is returned.

```
>>> st='hello'
>>> i=st.find('l')
>>> print(i)                              #output is 2
>>> i=st.find('lo')
>>> print(i)                              #output is 3
>>> print(st.find('x'))                   #output is -1
```

The find() function can take one more form with two additional arguments viz. start and end positions for search.

```
>>> st="calender of Feb. cal of march"
>>> i= st.find('cal')
>>> print(i)                        #output is 0
```

Here, the substring 'cal'is found in the very first position of st, hence the result is 0.

```
>>> i=st.find('cal',10,20)
>>> print(i)                        #output is 17
```

Here, the substring cal is searched in the string st between 10th and 20th position and hence the result is 17.

```
>>> i=st.find('cal',10,15)
>>> print(i)                        #output is -1
```

In this example, the substring 'cal' has not appeared between 10th and 15th character of st. Hence, the result is -1.

- **s.strip():** Returns a copy of string *s* by removing leading and trailing white spaces.

```
>>> st="          hello world          "
>>> st1 = st.strip()
>>> print(st1)
              hello world
```

The *strip()* function can be used with an argument *chars*, so that specified *chars* are removed from beginning or ending of *s* as shown below –

```
>>> st="###Hello##"
>>> st1=st.strip('#')
>>> print(st1)                   #all hash symbols are removed
    Hello
```

We can give more than one character for removal as shown below –

```
>>> st="Hello world"
>>> st1=st.strip("Hld")
    ello wor
```

- **S.startswith(prefix, start, end):** This function has 3 arguments of which *start* and *end* are option. This function returns True if S starts with the specified *prefix*, False otherwise.

```
>>> st="hello world"
>>> st.startswith("he")                 #returns True
```

When *start* argument is provided, the search begins from that position and returns True or False based on search result.

```
>>> st="hello world"
>>> st.startswith("w",6)                #True because w is at 6th position
```

When both *start* and *end* arguments are given, search begins at *start* and ends at *end*.

```
>>> st="xyz abc pqr ab mn gh“
>>> st.startswith("pqr ab mn",8,12)     #returns False
>>> st.startswith("pqr ab mn",8,18)     #returns True
```

The startswith() function requires case of the alphabet to match. So, when we are not sure about the case of the argument, we can convert it to either upper case or lowercase and then use startswith() function as below –

```
>>> st="Hello"
>>> st.startswith("he")                       #returns False
>>> st.lower().startswith("he")               #returns True
```

- **S.count(s1, start, end):** The count() function takes three arguments – *string, starting position* and *ending position*. This function returns the number of non-overlapping occurrences of substring s1 in string S in the range of *start* and *end*.

```
>>> st="hello how are you? how about you?"
>>> st.count('h')               #output is 3
>>> st.count(‘how’)             #output is 2
>>> st.count(‘how’,3,10)        #output is 1 because of range given
```

Example:

```
st=input("Enter a string:")
ch=input("Enter a character to be counted:")
c=st.count(ch)
print("%s appeared %d times in %s"%(ch,c,st))
```

→ Parsing Strings

- Sometimes, we may want to search for a substring matching certain criteria.

- For example, finding domain names from email-Ids in the list of messages is a useful task in some projects.
- Consider a string below and we are interested in extracting only the domain name.

“From mamatha.a@*saividya.ac.in* *Wed Feb 21 09:14:16 2018”*

Now, aim is to extract only *saividya.ac.in*, which is the domain name.
We can think of logic as–

- Identify the position of @, because all domain names in email IDs will be after the symbol @
- Identify a white space which appears after @ symbol, because that will be the end of domain name.
- Extract the substring between @ and white-space.

The concept of string slicing and *find()* function will be useful here.
Consider the code given below –

```
st="From mamatha.a@saividya.ac.in     Wed Feb 21 09:14:16 2018"
atpos=st.find('@')                   #finds the position of @

print('Position of @ is', atpos)
spacePos=st.find(‘ ‘, atpos)                    #position of white-space after @
print('Position of space after @ is', spacePos)

host=st[atpos+1:spacePos]     #slicing from @ till white-space

print(host)
```

Output:

```
Position of @ is 14
Position of space after @ is 29
saividya.ac.in
```

→ Format Operator

- The format operator, % allows us to construct strings, replacing parts of the strings with the data stored in variables.
- The first operand is the format string, which contains one or more *format sequences* that specify how the second operand is formatted.
  Syntax: “<format>” % (<values>)
- The result is a string.

```
>>> sum=20
>>> '%d' %sum
        ‘20’                     #string ‘20’, but not integer 20
```

- Note that, when applied on both integer operands, the % symbol acts as a modulus operator. When the first operand is a string, then it is a format operator.
- Consider few examples illustrating usage of format operator.

Ex1.
```
>>> "The sum value %d is originally integer"%sum
          'The sum value 20 is originally integer'
```

Ex2.
```
>>> '%d %f %s'%(3,0.5,'hello')
          '3 0.500000 hello'
```

Ex3.
```
>>> '%d %g %s'%(3,0.5,'hello')
          '3 0.5 hello'
```

Ex4.
```
>>> '%d'% 'hello'
    TypeError: %d format: a number is required, not str
```

Ex5.
```
>>> '%d %d %d'%(2,5)
    TypeError: not enough arguments for format string
```

2.3 FILES

- File handling is an important requirement of any programming language, as it allows us to store the data permanently on the secondary storage and read the data from a permanent source.
- Here, we will discuss how to perform various operations on files using the programming language Python.

→ Persistence

- The programs that we have considered till now are based on console I/O. That is, the input was taken from the keyboard and output was displayed onto the monitor.
- When the data to be read from the keyboard is very large, console input becomes a laborious job.
- Also, the output or result of the program has to be used for some other purpose later, it has to be stored permanently.
- Hence, reading/writing from/to files are very essential requirement of programming.
- We know that the programs stored in the hard disk are brought into main memory to execute them.
- These programs generally communicate with CPU using conditional execution, iteration, functions etc.
- But, the content of main memory will be erased when we turn-off our computer.
- Here we will discuss about working with secondary memory or files. The files stored on the

secondary memory are permanent and can be transferred to other machines using pen-drives/CD.

→ Opening Files

- To perform any operation on a file, one must open a file.
- File opening involves communication with operating system.
- In Python, a file can be opened using a built-in function ***open()***.
- While opening a file, we must specify the name of the file to be opened. Also, we must inform the OS about the purpose of opening a file, which is termed as *file opening mode*.
- The syntax of ***open()*** function is as below –

  fhand= open("filename", "mode")

  Here, filename is name of the file to be opened. This string may be just a name of the file, or it may include pathname also. Pathname of the file is optional when the file is stored in current working directory

mode	This string indicates the purpose of opening a file. It takes a pre- defined set of values as given in Table below
fhand	It is a reference to an object of ***file*** class, which acts as a handler or tool for all further operations on files.

- When our Python program makes a request to open a specific file in a particular mode, then OS will try to serve the request.
- When a file gets opened successfully, then a file object is returned. This is known as *file handle* and is as shown in Figure below.
- It will help to perform various operations on a file through our program. If the file cannot be opened due to some reason, then error message (*traceback*) will be displayed.

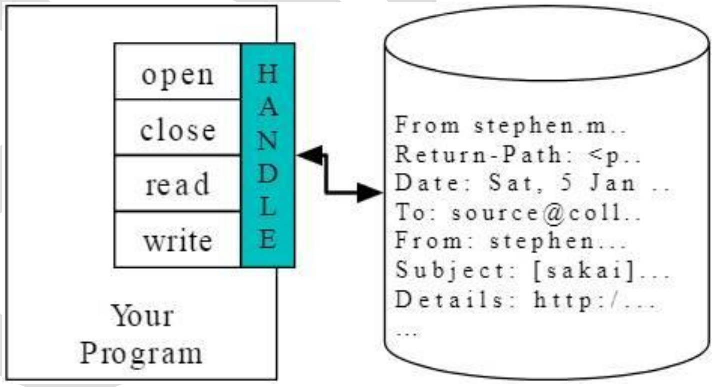


Figure A File Handle

- A file opening may cause an error due to some of the reasons as listed below –
  - File may not exist in the specified path (when we try to read a file)
  - File may exist, but we may not have a permission to read/write a file
  - File might have got corrupted and may not be in an opening state

- Since, there is no guarantee about getting a file handle from OS when we try to open a file, it is always better to write the code for file opening using *try-except* block.
- This will help us to manage error situation.

Mode	**Meaning**
r	Opens a file for reading purpose. If the specified file does not exist in the specified path, or if you don't have permission, error message will be displayed. This is the default mode of *open()* function in Python.
w	Opens a file for writing purpose. If the file does not exist, then a new file with the given name will be created and opened for writing. If the file already exists, then its content will be over-written.
a	Opens a file for appending the data. If the file exists, the new content will be appended at the end of existing content. If no such file exists, it will be created and new content will be written into it.
r+	Opens a file for reading and writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
rb	Opens a file for reading only in binary format
wb	Opens a file for writing only in binary format
ab	Opens a file for appending only in binary format

→ Text Files and Lines

- A text file is a file containing a sequence of lines
- It contains only the plain text without any images, tables etc.
- Different lines of a text file are separated by a newline character \n.
- In the text files, this newline character may be invisible, but helps in identifying every line in the file. There will be one more special entry at the end to indicate end of file (EOF).

NOTE: There is one more type of file called binary file, which contains the data in the form of bits. These files are capable of storing text, image, video, audio etc. All these data will be stored in the form of a group of bytes whose formatting will be known. The supporting program can interpret these files properly, whereas when opened using normal text editor, they look like messy, unreadable set of characters.

→ Reading Files

- When we successfully open a file to read the data from it, the ***open()*** function returns the file handle (or an object reference to *file* object) which will be pointing to the first character in the file.
- A text file containing lines can be iterated using a for-loop starting from the beginning with the help of this file handle. Consider the following example of counting number of lines in a file.

NOTE: Before executing the below given program, create a text file (using Notepad or similar editor) *myfile.txt* in the current working directory (The directory where you are going store your Python program). Open this text file and add few random lines to it and then close. Now, open a Python script file, say *countLines.py* and save it in the same directory as that of your text file *myfile.txt*. Then, type the following code in Python script *countLines.py* and execute the program. (You can store text file and Python script file in different directories. But, if you do so, you have to mention complete path of text file in the *open()* function.)

Sample Text file *myfile.txt:*

```
hello how are you? I
am doing fine what
about you?
```

Python script file *countLines.py*

```
fhand=open('myfile.txt','r') count =0
for line in fhand:
    count+=1
    print("Line Number ",count, ":", line)

print("Total lines=",count)
 fhand.close()
```

Output:

```
Line Number 1 : hello how are you?
 Line Number 2 : I am doing fine
Line Number 3 : what about you?
Total lines= 3
```

- In the above program, initially, we will try to open the file 'myfile.txt. As we have already created that file, the file handler will be returned and the object reference to this file will be stored in fhand.
- Then, in the for-loop, we are using fhand as if it is a sequence of lines. For each line in the file, we are counting it and printing the line.
- In fact, a line is identified internally with the help of new-line character present at the end of each line.

- Though we have not typed \n anywhere in the file myfile.txt, after each line, we would have pressed enter-key. This act will insert a \n, which is invisible when we view the file through notepad.
- Once all lines are over, fhandwill reach end-of-file and hence terminates the loop.
- Note that, when end of file is reached (that is, no more characters are present in the file), then an attempt to read will return Noneor empty character ‘’(two quotes without space in between).
- Once the operations on a file is completed, it is a practice to close the file using a function *close()*.
- Closing of a file ensures that no unwanted operations are done on a file handler.
- Moreover, when a file was opened for writing or appending, closure of a file ensures that the last bit of data has been uploaded properly into a file and the end-of-file is maintained properly.
- If the file handler variable (in the above example, fhand) is used to assign some other file object (using *open()* function), then Python closes the previous file automatically.
- If you run the above program and check the output, there will be a gap of two lines between each of the output lines. This is because, the new-line character \n is also a part of the variable line in the loop, and the *print()* function has default behavior of adding a line at the end (due to default setting of *end* parameter of *print()*).
- To avoid this double-line spacing, we can remove the new-line character attached at the end of variable line by using built-in string function *rstrip()* as below –

```
print("Line Number ",count, ":", line.rstrip())
```

- It is obvious from the logic of above program that from a file, each line is read one at a time, processed and discarded.
- Hence, there will not be a shortage of main memory even though we are reading a very large file.
- But, when we are sure that the size of our file is quite small, then we can use ***read()*** function to read the file contents.
- This function will read entire file content as a single string. Then, required operations can be done on this string using built-in string functions. Consider the below given example –

```
fhand=open('myfile.txt')
s=fhand.read()
print(“Total number of characters:”,len(s))
 print(“String up to 20 characters:”, s[:20])
```

- After executing above program using previously created file *myfile.txt*, then the output would be –

```
Total number of characters:50
String up to 20 characters: hello how are you? I
```

→ Writing Files

- To write a data into a file, we need to use the mode ***w*** in *open()* function.

```
>>> fhand=open(“mynewfile.txt","w")
>>> print(fhand)
<_io.TextIOWrapper name='mynewfile.txt' mode='w' encoding='cp1252'>
```

- If the file specified already exists, then the old contents will be erased and it will be ready to write new data into it.
- If the file does not exists, then a new file with the given name will be created.
- The ***write()*** method is used to write data into a file.
- This method returns number of characters successfully written into a file. For example,

```
>>> s="hello how are you?"
>>> fhand.write(s)
18
```

- Now, the file object keeps track of its position in a file.
- Hence, if we write one more line into the file, it will be added at the end of previous line.
- Here is a complete program to write few lines into a file –

```
fhand=open('f1.txt','w')
for i in range(5):
        line=input("Enter a line: ")
        fhand.write(line+"\n")
fhand.close()
```

- The above program will ask the user to enter 5 lines in a loop.
- After every line has been entered, it will be written into a file. Note that, as ***write()*** method doesn’t add a new-line character by its own, we need to write it explicitly at the end of every line.
- Once the loop gets over, the program terminates. Now, we need to check the file f1.txt on the disk (in the same directory where the above Python code is stored) to find our input lines that have been written into it.

→ Searching through a File

- Most of the times, we would like to read a file to search for some specific data within it.
- This can be achieved by using some string methods while reading a file. For example, we may be interested in printing only the line which starts with a character ***h.***
- Then we can use *startswith()* method.

```
fhand=open('myfile.txt')
for line in fhand:
        if line.startswith('h'):
                print(line)
fhand.close()
```

- Assume the input file *myfile.txt* is containing the following lines –

```
hello how are you?
I am doing fine
how about you?
```

- Now, if we run the above program, we will get the lines which starts with *h* –

```
hello how are you?
how about you?
```

→ Letting the User Choose the File Name

- In a real time programming, it is always better to ask the user to enter a name of the file which he/she would like to open, instead of hard-coding the name of a file inside the program.

```
fname=input("Enter a file name:")
fhand=open(fname)

count =0
for line in fhand:
        count+=1
        print("Line Number ",count, ":", line)

print("Total lines=",count)
fhand.close()
```

- In this program, the user input filename is received through variable fname, and the same has been used as an argument to open() method.
- Now, if the user input is *myfile.txt* (discussed before), then the result would be

```
Total lines=3
```

- Everything goes well, if the user gives a proper file name as input. But, what if the input filename cannot be opened (Due to some reason like – file doesn't exists, file permission denied etc)?
- Obviously, Python throws an error. The programmer need to handle such run- time errors as discussed in the next section.

→ Using *try, except* to Open a File

- It is always a good programming practice to write the commands related to file opening within a *try* block. Because, when a filename is a user input, it is prone to errors.
- Hence, one should handle it carefully. The following program illustrates this –

```
fname=input("Enter a file name:")
try:
      fhand=open(fname)
except:
      print("File cannot be opened") exit()
count =0
for line in fhand:
       count+=1
      print("Line Number ",count, ":", line)

print("Total lines=",count)
 fhand.close()
```

- In the above program, the command to open a file is kept within *try* block. If the specified file cannot be opened due to any reason, then an error message is displayed saying File cannot be opened, and the program is terminated.
- If the file could able to open successfully, then we will proceed further to perform required task using that file.

→ Debugging

- While performing operations on files, we may need to extract required set of lines or words or characters.
- For that purpose, we may use string functions with appropriate delimiters that may exist between the words/lines of a file.
- But, usually, the invisible characters like white-space, tabs and new-line characters are confusing and it is hard to identify them properly. For example,

```
>>> s="1 2\t 3\n 4"
>>> print(s)
  1 2      3
  4
```

- Here, by looking at the output, it may be difficult to make out where there is a space, where is a tab etc.
- Python provides a utility function called as ***repr()*** to solve this problem.
- This method takes any object as an argument and returns a string representation of that object.
- For example, the *print()* in the above code snippet can be modified as –

```
>>> print(repr(s))
 '1 2\t3\n4'
```

Note that, some of the systems use \n as new-line character, and few others may use \r (carriage return) as a new-line character. The ***repr()*** method helps in identifying that too.

MODULE III

3.1 LISTS

A list is a sequence, Lists are mutable, Traversing a list, List operations, List slices, List Methods, Deleting elements, Lists and functions, Lists and strings, Parsing lines, Objects and values , Aliasing, List arguments, Debugging

3.2 DICTIONARIES

Introduction, Dictionary as a set of counters, Dictionaries and files, Looping and Advanced text parsing, Debugging

3.3 TUPLES

Tuples are immutable, Comparing tuples, Tuple assignment Dictionaries and tuples, Multiple assignment with dictionaries, The most common words, Using tuples as keys in dictionaries, Sequences: strings, lists, and tuples, Debugging

3.4 REGULAR EXPRESSIONS

Character matching in regular expressions, Extracting data using regular expressions, Combining searching and extracting Escape character, Summary, Bonus section for Unix / Linux users

MODULE III

3.1 LISTS

- A list is an ordered sequence of values.
- It is a data structure in Python. The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are called as *elements* or *items*.
- The elements of lists are enclosed within square brackets.
- For example,

```
ls1=[10,-4, 25, 13]
ls2=["Tiger", "Lion", "Cheetah"]
```

- Here, ls1 is a list containing four integers, and ls2 is a list containing three strings.
- A list need not contain data of same type.
- We can have mixed type of elements in list.
- For example,

```
ls3=[3.5, 'Tiger', 10, [3,4]]
```

- Here, ls3 contains a float, a string, an integer and a list.
- This illustrates that a list can be nested as well.
- An empty list can be created any of the following ways –

```
>>> ls =[]
>>> type(ls)
        <class 'list'>
        or
>>> ls =list()
>>> type(ls)
        <class 'list'>
```

- In fact, list() is the name of a method (special type of method called as constructor – which will be discussed in Module 4) of the class *list*.
- Hence, a new list can be created using this function by passing arguments to it as shown below –

```
>>> ls2=list([3,4,1])
>>> print(ls2)
        [3, 4, 1]
```

→ Lists are Mutable

- The elements in the list can be accessed using a numeric index within square-brackets.
- It is similar to extracting characters in a string.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[1])
        hi
>>> print(ls[2])
      [2, 3]
```

- Observe here that, the inner list is treated as a single element by outer list. If we would like to access the elements within inner list, we need to use double-indexing as shown below –

```
>>> print(ls[2][0]) 2
>>> print(ls[2][1]) 3
```

- Note that, the indexing for inner-list again starts from 0.
- Thus, when we are using double- indexing, the first index indicates position of inner list inside outer list, and the second index means the position particular value within inner list.
- Unlike strings, lists are mutable. That is, using indexing, we can modify any value within list.
- In the following example, the 3rd element (i.e. index is 2) is being modified –

```
>>> ls=[34, 'hi', [2,3],-5]
>>> ls[2]='Hello'
>>> print(ls)
        [34, 'hi', 'Hello', -5]
```

- The list can be thought of as a relationship between indices and elements. This relationship is called as a ***mapping***. That is, each index maps to one of the elements in a list.
- The index for extracting list elements has following properties –

➢ Any integer expression can be an index.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[2*1])
        [2,3]
```

➢ Attempt to access a non-existing index will throw and IndexError.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[4])
IndexError: list index out of range
```

➢ A negative indexing counts from backwards.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[-1])
        -5
>>> print(ls[-3])
        hi
```

- The ***in*** operator applied on lists will results in a Boolean value.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> 34 in ls
        True
>>> -2 in ls
         False
```

→ Traversing a List

- A list can be traversed using *for* loop.
- If we need to use each element in the list, we can use the *for* loop and *in* operator as below

```
>>> ls=[34, 'hi', [2,3],-5]
>>> for item in ls:
            print(item)


    34
    hi
    [2,3]
    -5
```

- List elements can be accessed with the combination of *range()* and *len()* functions as well –

```
ls=[1,2,3,4]
for i in range(len(ls)):
    ls[i]=ls[i]**2

print(ls)

#output is
[1, 4, 9, 16]
```

- Here, we wanted to do modification in the elements of list. Hence, referring indices is suitable than referring elements directly.
- The *len()* returns total number of elements in the list (here it is 4).
- Then *range()* function makes the loop to range from 0 to 3 (i.e. 4-1).
- Then, for every index, we are updating the list elements (replacing original value by its square).

→ List Operations

- Python allows to use operators + and * on lists.
- The operator + uses two list objects and returns concatenation of those two lists.
- Whereas * operator take one list object and one integer value, say n, and returns a list by repeating itself for n times.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6,7]
>>> print(ls1+ls2)                    #concatenation using +
[1, 2, 3, 5, 6, 7]

>>> ls1=[1,2,3]
>>> print(ls1*3)                      #repetition using *
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> [0]*4                             #repetition using *
 [0, 0, 0, 0]
```

→ List Slices

- Similar to strings, the slicing can be applied on lists as well. Consider a list t given below, and a series of examples following based on this object.

```
t=['a','b','c','d','e']
```

- Extracting full list without using any index, but only a slicing operator –

```
>>> print(t[:])
        ['a', 'b', 'c', 'd', 'e']
```

- Extracting elements from 2nd position –

```
>>> print(t[1:])
        ['b', 'c', 'd', 'e']
```

- Extracting first three elements –

```
>>> print(t[:3])
        ['a', 'b', 'c']
```

- Selecting some middle elements –

```
>>> print(t[2:4])
        ['c', 'd']
```

- Using negative indexing –

```
>>> print(t[:-2])
        ['a', 'b', 'c']
```

- **Reversing a list** using negative value for stride –

```
>>> print(t[::-1])
        ['e', 'd', 'c', 'b', 'a']
```

- **Modifying (reassignment) only required set of values –**

```
>>> t[1:3]=['p','q']
>>> print(t)
        ['a', 'p', 'q', 'd', 'e']
```

Thus, slicing can make many tasks simple.

→ List Methods

There are several built-in methods in *list* class for various purposes. Here, we will discuss some of them.

- **append():** This method is used to add a new element at the end of a list.

```
>>> ls=[1,2,3]
>>> ls.append('hi')
>>> ls.append(10)
>>> print(ls)
        [1, 2, 3, 'hi', 10]
```

- **extend():** This method takes a list as an argument and all the elements in this list are added at the end of invoking list.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>> print(ls2)
        [5, 6, 1, 2, 3]
```

Now, in the above example, the list ls1 is unaltered.

- **sort():** This method is used to sort the contents of the list. By default, the function will sort the items in ascending order.

```
>>> ls=[3,10,5, 16,-2]
>>> ls.sort()
>>> print(ls)
        [-2, 3, 5, 10, 16]
```

When we want a list to be sorted in descending order, we need to set the argument as shown

```
>>> ls.sort(reverse=True)
>>> print(ls)
[16, 10, 5, 3, -2]
```

- **reverse():** This method can be used to reverse the given list.

```
>>> ls=[4,3,1,6]
>>> ls.reverse()
>>> print(ls)
        [6, 1, 3, 4]
```

- **count():** This method is used to count number of occurrences of a particular value within list.

```
>>> ls=[1,2,5,2,1,3,2,10]
>>> ls.count(2)
        3                       #the item 2 has appeared 3 tiles in ls
```

- **clear():** This method removes all the elements in the list and makes the list empty.

```
>>> ls=[1,2,3]
>>> ls.clear()
>>> print(ls)
          []
```

- **insert():** Used to insert a value before a specified index of the list.

```
>>> ls=[3,5,10]
>>> ls.insert(1,"hi")
>>> print(ls)
        [3, 'hi', 5, 10]
```

- **index():** This method is used to get the index position of a particular value in the list.

```
>>> ls=[4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
```

```
                    1
```

Here, the number 2 is found at the index position 1. Note that, this function will give index of only the first occurrence of a specified value. The same function can be used with two more arguments *start* and *end* to specify a range within which the search should take place.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
        2
>>> ls.index(2,3,7) 6
```

If the value is not present in the list, it throws ValueError.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(53)
        ValueError: 53 is not in list
```

Few important points about List Methods:

1. There is a difference between *append()* and *extend()* methods. The former adds the argument as it is, whereas the latter enhances the existing list. To understand this, observe the following example –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.append(ls1)
>>> print(ls2)
    [5, 6, [1, 2, 3]]
```

   Here, the argument ls1 for the *append()* function is treated as one item, and made as an inner list to ls2. On the other hand, if we replace *append()* by *extend()* then the result would be –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>> print(ls2)
     [5, 6, 1, 2, 3]
```

2. The ***sort()*** function can be applied only when the list contains elements of compatible types. But, if a list is a mix non-compatible types like integers and string, the comparison cannot be done. Hence, Python will throw TypeError.

   For example,

```
>>> ls=[34, 'hi', -5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

   Similarly, when a list contains integers and sub-list, it will be an error.

```
>>> ls=[34,[2,3],5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'list' and 'int'
```

Integers and floats are compatible and relational operations can be performed on them. Hence, we can sort a list containing such items.

```
>>> ls=[3, 4.5, 2]
>>> ls.sort()
>>> print(ls)
        [2, 3, 4.5]
```

3. The ***sort()*** function uses one important argument ***keys***. When a list is containing tuples, it will be useful. We will discuss tuples later in this Module.

4. Most of the list methods like *append(), extend(), sort(), reverse()* etc. modify the list object internally and return None.

```
>>> ls=[2,3]
>>> ls1=ls.append(5)
>>> print(ls)
         [2,3,5]
>>> print(ls1)
        None
```

→ Deleting Elements

Elements can be deleted from a list in different ways. Python provides few built-in methods for removing elements as given below –

- **pop():** This method deletes the last element in the list, by default.

```
>>> ls=[3,6,-2,8,10]
>>> x=ls.pop()                  #10 is removed from list and stored in x
>>> print(ls)
    [3, 6, -2, 8]
>>> print(x)
    10
```

When an element at a particular index position has to be deleted, then we can give that position as argument to *pop()* function.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)                #item at index 1 is popped
>>> print(t)
          ['a', 'c']
>>> print(x) b
```

- **remove():** When we don’t know the index, but know the value to be removed, then this function can be used.

```
>>> ls=[5,8, -12,34,2]
>>> ls.remove(34)
>>> print(ls)
          [5, 8, -12, 2]
```

Note that, this function will remove only the first occurrence of the specified value, but not all occurrences.

```
>>> ls=[5,8, -12, 34, 2, 6, 34]
>>> ls.remove(34)
>>> print(ls)
     [5, 8, -12, 2, 6, 34]
```

Unlike *pop()* function, the *remove()* function will not return the value that has been deleted.

- **del:** This is an operator to be used when more than one item to be deleted at a time. Here also, we will not get the items deleted.

```
>>> ls=[3,6,-2,8,1]
>>> del ls[2]                    #item at index 2 is deleted
>>> print(ls)
          [3, 6, 8, 1]

>>> ls=[3,6,-2,8,1]
>>> del ls[1:4]                  #deleting all elements from index 1 to 3
>>> print(ls)
             [3, 1]
```

Example: Deleting all odd indexed elements of a list –

```
>>> t=['a', 'b', 'c', 'd', 'e']
>>> del t[1::2]
>>> print(t)
             ['a', 'c', 'e']
```

→ Lists and Functions

- The utility functions like *max(), min(), sum(), len()* etc. can be used on lists.
- Hence most of the operations will be easy without the usage of loops.

```
>>> ls=[3,12,5,26, 32,1,4]
>>> max(ls)                      # prints     32
>>> min(ls)                      # prints     1
>>> sum(ls)                      # prints     83
>>> len(ls)                      # prints     7
>>> avg=sum(ls)/len(ls)
>>> print(avg)
     11.857142857142858
```

- When we need to read the data from the user and to compute sum and average of those numbers, we can write the code as below –

```
ls= list()

while (True):
     x= input('Enter a number: ')
```

```
        if x== 'done':
            break

        x= float(x)
        ls.append(x)

    average = sum(ls) / len(ls)
    print('Average:', average)
```

- In the above program, we initially create an empty list.
- Then, we are taking an infinite *while*- loop.
- As every input from the keyboard will be in the form of a string, we need to convert x into float type and then append it to a list.
- When the keyboard input is a string 'done', then the loop is going to get terminated.
- After the loop, we will find the average of those numbers with the help of built-in functions *sum()* and *len()*.

→ Lists and Strings

- Though both lists and strings are sequences, they are not same.
- In fact, a list of characters is not same as string.
- To convert a string into a list, we use a method ***list()*** as below –

```
>>> s="hello"
>>> ls=list(s)
>>> print(ls)
        ['h', 'e', 'l', 'l', 'o']
```

- The method ***list()*** breaks a string into individual letters and constructs a list.
- If we want a list of words from a sentence, we can use the following code –

```
>>> s="Hello how are you?"
>>> ls=s.split()
>>> print(ls)
        ['Hello', 'how', 'are', 'you?']
```

- Note that, when no argument is provided, the ***split()*** function takes the delimiter as white space.
- If we need a specific delimiter for splitting the lines, we can use as shown in following example –

```
>>> dt="20/03/2018"
>>> ls=dt.split('/')
>>> print(ls)
        ['20', '03', '2018']
```

- There is a method ***join()*** which behaves opposite to ***split()*** function.
- It takes a list of strings as argument, and joins all the strings into a single string based on the delimiter provided.
- For example –

```
>>> ls=["Hello", "how", "are", "you"]
>>> d=' '
>>> d.join(ls)
      'Hello how are you'
```

- Here, we have taken delimiter d as white space. Apart from space, anything can be taken as delimiter. When we don't need any delimiter, use empty string as delimiter.

→ Parsing Lines

- In many situations, we would like to read a file and extract only the lines containing required pattern. This is known as ***parsing***.
- As an illustration, let us assume that there is a log file containing details of email communication between employees of an organization.
- For all received mails, the file contains lines as –

  From stephen.marquard@uct.ac.za *Fri Jan 5 09:14:16 2018*
  From georgek@uct.ac.za *Sat Jan 6 06:12:51 2018*
  ………………
- Apart from such lines, the log file also contains mail-contents, to-whom the mail has been sent etc.
- Now, if we are interested in extracting only the days of incoming mails, then we can go for parsing.
- That is, we are interested in knowing on which of the days, the mails have been received. The code would be –

```
fhand = open('logFile.txt')
for line in fhand:
        line = line.rstrip()
        if not line.startswith('From '):
                continue
        words = line.split()
        print(words[2])
```

- Obviously, all received mails starts from the word From. Hence, we search for only such lines and then split them into words.
- Observe that, the first word in the line would be From, second word would be email-ID and the 3rd word would be day of a week. Hence, we will extract words[2]which is 3rd word.

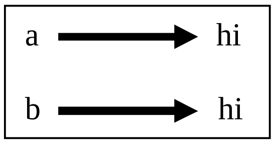
→ Objects and Values

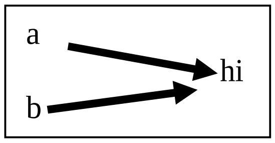
- Whenever we assign two variables with same value, the question arises – whether both the variables are referring to same object, or to different objects.
- This is important aspect to know, because in Python everything is a class object.
- There is nothing like elementary data type.

Consider a situation –

```
a= "hi"
b= "hi"
```

- Now, the question is whether both a and b refer to the ***same string***.
- There are two possible states –





- In the first situation, a and b are two different objects, but containing same value. The modification in one object is nothing to do with the other.
- Whereas, in the second case, both a and b are referring to the same object.
- That is, a is an ***alias name*** for b and vice- versa. In other words, these two are referring to same memory location.
- To check whether two variables are referring to same object or not, we can use ***is*** operator.

```
>>> a= "hi"
>>> b= "hi"
>>> a is b                    #result is True
>>> a==b                      #result is True
```

- When two variables are referring to same object, they are called as ***identical objects.***
- When two variables are referring to different objects, but contain a same value, they are known as ***equivalent objects***.
- For example,

```
>>> s1=input("Enter a string:")        #assume you entered hello
>>> s2= input("Enter a string:")       #assume you entered hello

>>> s1 is s2                           #check s1 and s2 are identical False
>>> s1 == s2                           #check s1 and s2 are equivalent True
```

Here **s1** and **s2** are equivalent, but not identical.

- If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.
- String literals are ***interned*** by default. That is, when two string literals are created in the program with a same value, they are going to refer same object. But, string variables read from the key-board will not have this behavior, because their values are depending on the user's choice.
- Lists are not interned. Hence, we can see following result –

```
>>> ls1=[1,2,3]
>>> ls2=[1,2,3]
>>> ls1 is ls2                #output is False
>>> ls1 == ls2                #output is True
```

→ Aliasing

- When an object is assigned to other using assignment operator, both of them will refer to same object in the memory.
- The association of a variable with an object is called as *reference.*

```
>>> ls1=[1,2,3]
>>> ls2= ls1
>>> ls1 is ls2                #output is True
```

- Now, ls2 is said to be ***reference*** of ls1. In other words, there are two references to the same object in the memory.

- An object with more than one reference has more than one name, hence we say that object is ***aliased.*** If the aliased object is mutable, changes made in one alias will reflect the other.

```
>>> ls2[1]= 34
>>> print(ls1)                    #output is [1, 34, 3]
```

Strings are safe in this regards, as they are immutable.

→ List Arguments

- When a list is passed to a function as an argument, then function receives reference to this list.
- Hence, if the list is modified within a function, the caller will get the modified version.
- Consider an example –

```
def del_front(t):
        del t[0]

ls = ['a', 'b', 'c']
del_front(ls)
print(ls)

# output is
['b', 'c']
```

- Here, the argument ls and the parameter t both are aliases to same object.
- One should understand the operations that will modify the list and the operations that create a new list.
- For example, the ***append()*** function modifies the list, whereas the + operator creates a new list.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)                    #output is [1 2 3]
>>> print(t2)                    #prints None

>>> t3 = t1 + [5]
>>> print(t3)                    #output is [1 2 3 5]
>>> t2 is t3                     #output is False
```

- Here, after applying *append()* on t1 object, the t1 itself has been modified and t2 is not going to get anything.
- But, when + operator is applied, t1 remains same but t3 will get the updated result.
- The programmer should understand such differences when he/she creates a function intending to modify a list.
- For example, the following function has no effect on the original list –

```
def test(t):
         t=t[1:]
```

```
ls=[1,2,3]
test(ls)
print(ls)              #prints [1, 2, 3]
```

- One can write a return statement after slicing as below –

```
def test(t):
        return t[1:]

ls=[1,2,3]
ls1=test(ls)
print(ls1)             #prints [2, 3]
print(ls)              #prints [1, 2, 3]
```

- In the above example also, the original list is not modified, because a return statement always creates a new object and is assigned to LHS variable at the position of function call.

3.2 DICTIONARIES

- A dictionary is a collection of unordered set of ***key:value*** pairs, with the requirement that keys are unique in one dictionary.
- Unlike lists and strings where elements are accessed using index values (which are integers), the values in dictionary are accessed using keys.
- A key in dictionary can be any immutable type like strings, numbers and tuples. (The tuple can be made as a key for dictionary, only if that tuple consist of string/number/ sub-tuples).
- As lists are mutable – that is, can be modified using index assignments, slicing, or using methods like *append(), extend()* etc, they cannot be a key for dictionary.
- One can think of a dictionary as a mapping between set of indices (which are actually keys) and a set of values.
- Each key maps to a value.
- An empty dictionary can be created using two ways –

```
d= {}
```
OR
```
d=dict()
```

- To add items to dictionary, we can use square brackets as –

```
>>> d={}
>>> d["Mango"]="Fruit"
>>> d["Banana"]="Fruit"
>>> d["Cucumber"]="Veg"
>>> print(d)
{'Mango': 'Fruit', 'Banana': 'Fruit', 'Cucumber': 'Veg'}
```

- ,,To initialize a dictionary at the time of creation itself, one can use the code like –

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135}
>>> print(tel_dir)
        {'Tom': 3491, 'Jerry': 8135}

>>> tel_dir['Donald']=4793
>>> print(tel_dir)
        {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
```

NOTE that the order of elements in dictionary is unpredictable. That is, in the above example, don't assume that 'Tom': 3491 is first item, 'Jerry': 8135 is second item etc. As dictionary members are not indexed over integers, the order of elements inside it may vary. However, using a *key,* we can extract its associated value as shown below –

```
>>> print(tel_dir['Jerry']) 8135
```

- Here, the key 'Jerry'maps with the value 8135, hence it doesn't matter where exactly it is inside the dictionary.
- If a particular key is not there in the dictionary and if we try to access such key, then the *KeyError* is generated.

```
        >>> print(tel_dir['Mickey']) KeyError:
                'Mickey'
```

- The ***len()*** function on dictionary object gives the number of key-value pairs in that object.

```
>>> print(tel_dir)
        {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
>>> len(tel_dir)
        3
```

- The ***in*** operator can be used to check whether any ***key*** (not value) appears in the dictionary object.

```
>>> 'Mickey' in tel_dir                 #output is False
>>> 'Jerry' in tel_dir                  #output is True
>>> 3491 in tel_dir                     #output is False
```

- We observe from above example that the value 3491 is associated with the key 'Tom' in tel_dir. But, the ***in*** operator returns False.
- The dictionary object has a method ***values()*** which will ***return a list*** of all the values associated with keys within a dictionary.
- If we would like to check whether a particular value exist in a dictionary, we can make use of it as shown below –

```
>>> 3491 in tel_dir.values()            #output is True
```

- The ***in*** operator behaves differently in case of lists and dictionaries as explained hereunder:
- When ***in*** operator is used to search a value in a list, then *linear search* algorithm is used internally. That is, each element in the list is checked one by one sequentially. This is considered to be expensive in the view of total time taken to process.
- Because, if there are 1000 items in the list, and if the element in the list which we are search for is in the last position (or if it does not exists), then before yielding result of search (True or False), we would have done 1000 comparisons.
- In other words, linear search requires *n* number of comparisons for the input size of *n* elements.
- Time complexity of the linear search algorithm is O(*n*).
- The keys in dictionaries of Python are basically ***hashable*** elements.
- The concept of ***hashing*** is applied to store (or maintain) the keys of dictionaries.
- Normally hashing techniques have the time complexity as *O(log n)* for basic operations like insertion, deletion and searching.
- Hence, the ***in*** operator applied on keys of dictionaries works better compared to that on lists.

→ Dictionary as a Set of Counters

- Assume that we need to count the frequency of alphabets in a given string. There are different methods to do it –
  - Create 26 variables to represent each alphabet. Traverse the given string and increment the corresponding counter when an alphabet is found.
  - Create a list with 26 elements (all are zero in the beginning) representing alphabets. Traverse the given string and increment corresponding indexed position in the list when an alphabet is found.
  - Create a dictionary with characters as keys and counters as values. When we find a character for the first time, we add the item to dictionary. Next time onwards, we increment the value of existing item.
- Each of the above methods will perform same task, but the logic of implementation will be different. Here, we will see the implementation using dictionary.

```
s=input("Enter a string:")          #read a string
d=dict()                            #create empty dictionary

for ch in s:                        #traverse through string
    if ch not in d:                 #if new character found
        d[ch]=1                     #initialize counter to 1
    else:                           #otherwise, increment counter
        d[ch]+=1

print(d)                            #display the dictionary
```

The sample output would be –

```
Enter a string:
Hello World
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'W': 1, 'r': 1, 'd': 1}
```

- It can be observed from the output that, a dictionary is created here with characters as keys and frequencies as values. **Note** that, here we have computed ***histogram*** of counters.
- Dictionary in Python has a method called as ***get()*,** which takes key and a default value as two arguments. If key is found in the dictionary, then the *get()* function returns corresponding value, otherwise it returns default value.
- For example,

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
>>> print(tel_dir.get('Jerry',0))
        8135
>>> print(tel_dir.get('Donald',0))
        0
```

- In the above example, when the *get()* function is taking 'Jerry' as argument, it returned corresponding value, as 'Jerry'is found in tel_dir.
- Whereas, when *get()* is used with 'Donald' as key, the default value 0 (which is provided by us) is returned.
- The function ***get()*** can be used effectively for calculating frequency of alphabets in a string.
- Here is the modified version of the program –

```
s=input("Enter a string:")
d=dict()

for ch in s:
    d[ch]=d.get(ch,0)+1

print(d)
```

- In the above program, for every character ch in a given string, we will try to retrieve a value. When the ch is found in d, its value is retrieved, 1 is added to it, and restored.
- If ch is not found, 0 is taken as default and then 1 is added to it.

→ Looping and Dictionaries

- When a *for*-loop is applied on dictionaries, it will iterate over the keys of dictionary.
- If we want to print key and values separately, we need to use the statements as shown

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
for k in tel_dir:
    print(k, tel_dir[k])
```

Output would be –

```
Tom 3491
Jerry 8135
Mickey 1253
```

- Note that, while accessing items from dictionary, the keys may not be in order. If we want to print the keys in alphabetical order, then we need to make a list of the keys, and then sort that list.
- We can do so using ***keys()*** method of dictionary and ***sort()*** method of lists.
- Consider the following code –

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
ls=list(tel_dir.keys())
print("The list of keys:",ls)
ls.sort()
print("Dictionary elements in alphabetical order:")
for k in ls:
    print(k, tel_dir[k])
```

The output would be –

```
The list of keys: ['Tom', 'Jerry', 'Mickey']
Dictionary elements in alphabetical order:
Jerry 8135
Mickey 1253
Tom 3491
```

Note: The key-value pair from dictionary can be together accessed with the help of a method ***items()*** as shown

```
>>> d={'Tom':3412, 'Jerry':6781, 'Mickey':1294}
>>> for k,v in d.items():
                print(k,v)
```

Output:

```
Tom 3412
Jerry 6781
Mickey 1294
```

The usage of comma-separated list k,v here is internally a tuple (another data structure in Python, which will be discussed later).

→ Dictionaries and Files

- A dictionary can be used to count the frequency of words in a file.
- Consider a file *myfile.txt* consisting of following text:

```
hello, how are you?
I am doing fine.
How about you?
```

- Now, we need to count the frequency of each of the word in this file. So, we need to take an outer loop for iterating over entire file, and an inner loop for traversing each line in a file.
- Then in every line, we count the occurrence of a word, as we did before for a character.
- The program is given as below –

```
fname=input("Enter file name:")
try:
    fhand=open(fname)
except:
    print("File cannot be opened")
    exit()

d=dict()
for line in fhand:
    for word in line.split():
        d[word]=d.get(word,0)+1
print(d)
```

The output of this program when the input file is *myfile.txt* would be –

```
Enter file name: myfile.txt
{'hello,':          1,     'how':      1,     'are':        1,     'you?':        2,     'I':        1,     'am':       1,
'doing': 1, 'fine.': 1, 'How': 1, 'about': 1}
```

- Few points to be observed in the above output –
  - The punctuation marks like comma, full point, question mark etc. are also considered as a part of word and stored in the dictionary. This means, when a particular word appears in a file with and without punctuation mark, then there will be multiple entries of that word.
  - The word 'how' and 'How' are treated as separate words in the above example because of uppercase and lowercase letters.

- While solving problems on text analysis, machine learning, data analysis etc. such kinds of treatment of words lead to unexpected results. So, we need to be careful in parsing the text and we should try to eliminate punctuation marks, ignoring the case etc. The procedure is discussed in the next section.

→ Advanced Text Parsing

- As discussed in the previous section, during text parsing, our aim is to eliminate punctuation marks as a part of word.
- The *string* module of Python provides a list of all punctuation marks as shown:

```
>>> import string
>>> string.punctuation
        '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

- The *str* class has a method *maketrans()* which returns a translation table usable for another method *translate()*.
- Consider the following syntax to understand it more clearly:

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

- The above statement replaces the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr.
- The fromstr and tostr can be empty strings and the deletestrparameter can be omitted.
- Using these functions, we will re-write the program for finding frequency of words in a file.

```
import string
fname=input("Enter file name:")
try:
        fhand=open(fname)
 except:
print("File cannot be opened")
 exit()

d=dict()
for line in fhand:
        line=line.rstrip()
        line=line.translate(line.maketrans('','',string.punctuation))
        line=line.lower()
        for word in line.split():
                d[word]=d.get(word,0)+1

print(d)
```

Now, the output would be –

```
Enter file name:myfile.txt
{'hello': 1, 'how': 2, 'are': 1, 'you': 2, 'i': 1, 'am': 1, 'doing': 1, 'fine': 1, 'about': 1}
```

- Comparing the output of this modified program with the previous one, we can make out that all the punctuation marks are not considered for parsing and also the case of the alphabets are ignored.

→ Debugging

- When we are working with big datasets (like file containing thousands of pages), it is difficult to debug by printing and checking the data by hand. So, we can follow any of the following procedures for easy debugging of the large datasets –
- **Scale down the input**: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just first 10 lines or with the smallest example you can find. You can either edit the files themselves, or modify the program so it reads only the first n lines. If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you correct the errors.
- **Check summaries and types**: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
- **Write self-checks**: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a ***sanity check*** because it detects results that are "completely illogical". Another kind of check compares the results of two different computations to see if they are consistent. This is called a ***consistency check***.
- **Pretty print the output**: Formatting debugging output can make it easier to spot an error.

3.3TUPLES

- A tuple is a sequence of items, similar to lists.
- The values stored in the tuple can be of any type and they are indexed using integers.
- Unlike lists, tuples are immutable. That is, values within tuples cannot be modified/reassigned. Tuples are *comparable* and *hashable* objects.
- Hence, they can be made as keys in dictionaries.
- A tuple can be created in Python as a comma separated list of items – may or may not be enclosed within parentheses.

```
>>> t='Mango', 'Banana', 'Apple'                    #without parentheses
>>> print(t)
        ('Mango', 'Banana', 'Apple')
>>> t1=('Tom', 341, 'Jerry')                        #with parentheses
>>> print(t1)
        ('Tom', 341, 'Jerry')
```

- Observe that tuple values can be of mixed types.
- If we would like to create a tuple with single value, then just a parenthesis will not suffice.
- For example,

```
>>> x=(3)                  #trying to have a tuple with single item
>>> print(x)
        3                  #observe, no parenthesis found
>>> type(x)
<class 'int'>              #not a tuple, it is integer!!
```

- Thus, to have a tuple with single item, we must include a comma after the item. That is,

```
>>> t=3,                    #or use the statement t=(3,)
>>> type(t)                 #now this is a tuple
<class 'tuple'>
```

- An empty tuple can be created either using a pair of parenthesis or using a function ***tuple()*** as below

```
>>> t1=()
>>> type(t1)
        <class 'tuple'>

>>> t2=tuple()
>>> type(t2)
        <class 'tuple'>
```

- If we provide an argument of type sequence (a list, a string or tuple) to the method ***tuple()***, then a tuple with the elements in a given sequence will be created:

    - Create tuple using string:

```
>>> t=tuple('Hello')
>>> print(t)
        ('H', 'e', 'l', 'l', 'o')
```

    - Create tuple using list:

```
>>> t=tuple([3,[12,5],'Hi'])
>>> print(t)
        (3, [12, 5], 'Hi')
```

    - Create tuple using another tuple:

```
>>> t=('Mango', 34, 'hi')
>>> t1=tuple(t)
>>> print(t1)
        ('Mango', 34, 'hi')
>>> t is t1
        True
```

Note that, in the above example, both t and t1 objects are referring to same memory location. That is, t1 is a reference to t.

- Elements in the tuple can be extracted using square-brackets with the help of indices.
- Similarly, slicing also can be applied to extract required number of items from tuple.

```
>>> t=('Mango', 'Banana', 'Apple')
>>> print(t[1])
        Banana
>>> print(t[1:])
        ('Banana', 'Apple')
>>> print(t[-1])
```

```
        Apple
```

- Modifying the value in a tuple generates error, because tuples are immutable –
```
>>> t[0]='Kiwi'
TypeError: 'tuple' object does not support item assignment
```

- We wanted to replace 'Mango' by 'Kiwi', which did not work using assignment.
- But, a tuple can be replaced with another tuple involving required modifications –

```
>>> t=('Kiwi',)+t[1:]
>>> print(t)
        ('Kiwi', 'Banana', 'Apple')
```

→ Comparing Tuples

- Tuples can be compared using operators like >, <, >=, == etc.
- The comparison happens lexicographically.
- For example, when we need to check equality among two tuple objects, the first item in first tuple is compared with first item in second tuple.
- If they are same, 2nd items are compared.
- The check continues till either a mismatch is found or items get over.
- Consider few examples –
```
>>> (1,2,3)==(1,2,5)
         False
>>> (3,4)==(3,4)
        True
```

- The meaning of < and > in tuples is not exactly *less than* and *greater than*, instead, it means *comes before* and *comes after*.
- Hence in such cases, we will get results different from checking equality (==).

```
>>> (1,2,3)<(1,2,5)
        True
>>> (3,4)<(5,2)
        True
```

- When we use relational operator on tuples containing non-comparable types, then TypeError will be thrown.
```
>>> (1,'hi')<('hello','world')
TypeError: '<' not supported between instances of 'int' and 'str'
```

- The *sort()* function internally works on similar pattern – it sorts primarily by first element, in case of tie, it sorts on second element and so on. This pattern is known as ***DSU*** –
  - **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
  - **Sort** the list of tuples using the Python built-in *sort*(), and
  - **Undecorate** by extracting the sorted elements of the sequence.

- Consider a program of sorting words in a sentence from longest to shortest, which illustrates DSU property.

```
txt = 'Ram and Seeta went to forest with Lakshman'
words = txt.split()

t = list()
for word in words:
        t.append((len(word), word))

print('The list is:',t)
t.sort(reverse=True)
res = list()

for length, word in t:
        res.append(word)
print('The sorted list:',res)
```

The output would be –

```
The list is:
 [(3,  'Ram'), (3,  'and'), (5, 'Seeta'), (4,  'went'), (2, 'to'), (6, 'forest'), (4, 'with'), (8, 'Lakshman')]

The sorted list:['Lakshman', 'forest', 'Seeta', 'went', 'with',
'and', 'Ram', 'to']
```

- In the above program, we have split the sentence into a list of words.
- Then, a tuple containing length of the word and the word itself are created and are appended to a list.
- Observe the output of this list – it is a list of tuples. Then we are sorting this list in descending order.
- Now for sorting, length of the word is considered, because it is a first element in the tuple.
- At the end, we extract length and word in the list, and create another list containing only the words and print it.

→ Tuple Assignment

- Tuple has a unique feature of having it at LHS of assignment operator.
- This allows us to assign values to multiple variables at a time.

```
>>> x,y=10,20
>>> print(x)              #prints 10
>>> print(y)              #prints 20
```

- When we have list of items, they can be extracted and stored into multiple variables as below –

```
>>> ls=["hello", "world"]
>>> x,y=ls
>>> print(x)                    #prints hello
>>> print(y)                    #prints world
```

- This code internally means that –

```
x= ls[0]
 y= ls[1]
```

- The best known example of assignment of tuples is ***swapping two values*** as below –

```
>>> a=10
>>> b=20
>>> a, b = b, a
>>> print(a, b)                 #prints 20 10
```

- In the above example, the statement a, b = b, a is treated by Python as – LHS is a set of variables, and RHS is set of expressions.
- The expressions in RHS are evaluated and assigned to respective variables at LHS.
- Giving more values than variables generates ValueError –

```
>>> a, b=10,20,5
ValueError: too many values to unpack (expected 2)
```

- While doing assignment of multiple variables, the RHS can be any type of sequence like list, string or tuple. Following example extracts user name and domain from an email ID.

```
>>> email='mamathaa@ieee.org'
>>> usrName, domain = email.split('@')
>>> print(usrName)                                  #prints mamathaa
>>> print(domain)                                   #prints ieee.org
```

→ Dictionaries and Tuples

- Dictionaries have a method called ***items()*** that returns a list of tuples, where each tuple is a key-value pair as shown below –

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
        [('b', 1), ('a', 10), ('c', 22)]
```

- As dictionary may not display the contents in an order, we can use ***sort()*** on lists and then print in required order as below –

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
        [('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> print(t)
        [('a', 10), ('b', 1), ('c', 22)]
```

→ Multiple Assignment with Dictionaries

- We can combine the method *items()*, tuple assignment and a for-loop to get a pattern for traversing dictionary:

```
d={'Tom': 1292, 'Jerry': 3501, 'Donald': 8913}
for key, val in list(d.items()):
        print(val,key)
```

The output would be –

```
1292 Tom
3501 Jerry
8913 Donald
```

- This loop has two iteration variables because ***items()*** returns a list of tuples.
- And key, val is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.
- For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary in hash order.
- Once we get a key-value pair, we can create a list of tuples and sort them:

```
d={'Tom': 9291, 'Jerry': 3501, 'Donald': 8913}
ls=list()
for key, val in d.items():
        ls.append((val,key))                #observe inner parentheses

print("List of tuples:",ls)
ls.sort(reverse=True)
print("List of sorted tuples:",ls)
```

The output would be –

```
List of tuples: [(9291, 'Tom'), (3501, 'Jerry'), (8913, 'Donald')]
List of sorted tuples: [(9291, 'Tom'), (8913, 'Donald'), (3501, 'Jerry')]
```

- In the above program, we are extracting key, val pair from the dictionary and appending it to the list ls.
- While appending, we are putting inner parentheses to make sure that each pair is treated as a tuple.
- Then, we are sorting the list in the descending order.
- The sorting would happen based on the telephone number (val), but not on name (key), as first element in tuple is telephone number (val).

→ The Most Common Words

- We will apply the knowledge gained about strings, tuple, list and dictionary till here to solve a problem – write a program to find most commonly used words in a text file.
- The logic of the program is –
    - ➢ Open a file

- ➢ Take a loop to iterate through every line of a file.
- ➢ Remove all punctuation marks and convert alphabets into lower case
- ➢ Take a loop and iterate over every word in a line.
- ➢ If the word is not there in dictionary, treat that word as a key, and initialize its value as 1. If that word already there in dictionary, increment the value.
- ➢ Once all the lines in a file are iterated, you will have a dictionary containing distinct words and their frequency. Now, take a list and append each key-value (word- frequency) pair into it.
- ➢ Sort the list in descending order and display only 10 (or any number of) elements from the list to get most frequent words.

```
import string
fhand = open('test.txt')
counts = dict()
for line in fhand:
        line = line.translate(str.maketrans('', '',string.punctuation))
        line = line.lower()

        for word in line.split():
                if word not in counts:
                        counts[word] = 1
                else:
                        counts[word] += 1

lst = list()
for key, val in list(counts.items()):
        lst.append((val, key))

lst.sort(reverse=True)
for key, val in lst[:10]:
        print(key, val)
```

Run the above program on any text file of your choice and observe the output.

→ Using Tuples as Keys in Dictionaries

- As tuples and dictionaries are hashable, when we want a dictionary containing composite keys, we will use tuples.
- For Example, we may need to create a telephone directory where name of a person is Firstname-last name pair and value is the telephone number.
- Our job is to assign telephone numbers to these keys.
- Consider the program to do this task –

```
names=(('Tom','Cat'),('Jerry','Mouse'), ('Donald', 'Duck'))
number=[3561, 4014, 9813]

telDir={}

for i in range(len(number)):
        telDir[names[i]]=number[i]

    for fn, ln in telDir:
        print(fn, ln, telDir[fn,ln])
```

The output would be –

```
Tom Cat 3561
Jerry Mouse 4014
Donald Duck 9813
```

→ Summary on Sequences: Strings, Lists and Tuples

- Till now, we have discussed different types of sequences viz. strings, lists and tuples.
- In many situations these sequences can be used interchangeably.
- Still, due their difference in behavior and ability, we may need to understand pros and cons of each of them and then to decide which one to use in a program.
- Here are few key points –

  1. Strings are more limited compared to other sequences like lists and Tuples. Because, the elements in strings must be characters only. Moreover, strings are immutable. Hence, if we need to modify the characters in a sequence, it is better to go for a list of characters than a string.
  2. As lists are mutable, they are most common compared to tuples. But, in some situations as given below, tuples are preferable.
     a. When we have a return statement from a function, it is better to use tuples rather than lists.
     b. When a dictionary key must be a sequence of elements, then we must use immutable type like strings and tuples
     c. When a sequence of elements is being passed to a function as arguments, usage of tuples reduces unexpected behavior due to aliasing.
  3. As tuples are immutable, the methods like ***sort()*** and ***reverse()*** cannot be applied on them. But, Python provides built-in functions ***sorted()*** and ***reversed()*** which will take a sequence as an argument and return a new sequence with modified results.

→ Debugging

- Lists, Dictionaries and Tuples are basically data structures.
- In real-time programming, we may require compound data structures like lists of tuples, dictionaries containing tuples and lists etc.
- But, these compound data structures are prone to ***shape errors*** – that is, errors caused when a data structure has the wrong type, size, composition etc.
- For example, when your code is expecting a list containing single integer, but you are giving a plain integer, then there will be an error.
- When debugging a program to fix the bugs, following are the few things a programmer can try –

  - **Reading:** Examine your code, read it again and check that it says what you meant to say.
  - **Running:** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

- **Ruminating:** Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- **Retreating:** At some point, the best thing to do is back off, undoing recent changes, until you get back you can start rebuilding.

3.4 REGULAR EXPRESSIONS

- Searching for required patterns and extracting only the lines/words matching the pattern is a very common task in solving problems programmatically.
- We have done such tasks earlier using string slicing and string methods like *split()*, *find()* etc.
- As the task of searching and extracting is very common, Python provides a powerful library called ***regular expressions*** to handle these tasks elegantly.
- Though they have quite complicated syntax, they provide efficient way of searching the patterns.
- The regular expressions are themselves little programs to search and parse strings.
- To use them in our program, the library/module ***re*** must be imported.
- There is a ***search()*** function in this module, which is used to find particular substring within a string.
- Consider the following example –

```
import re
fhand = open('myfile.txt')
for line in fhand:
        line = line.rstrip()
        if re.search('how', line):
                print(line)
```

- By referring to file *myfile.txt* that has been discussed in previous Chapters, the output would be

```
 hello, how are you?
how about you?
```

- In the above program, the *search()* function is used to search the lines containing a word *how*.
- One can observe that the above program is not much different from a program that uses ***find()*** function of strings. But, regular expressions make use of special characters with specific meaning.
- In the following example, we make use of caret (^) symbol, which indicates beginning of the line.

```
import re
hand = open('myfile.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^how', line):
                print(line)
```

The output would be –

```
how about you?
```

- Here, we have searched for a line which starts with a string *how*.
- Again, this program will not makes use of regular expression fully.
- Because, the above program would have been written using a string function *startswith()*. Hence,

in the next section, we will understand the true usage of regular expressions.

→ Character Matching in Regular Expressions

- Python provides a list of meta-characters to match search strings.
- Table below shows the details of few important metacharacters.
- Some of the examples for quick and easy understanding of regular expressions are given in next Table.

Table : List of Important Meta-Characters

Character	Meaning
^ (caret)	Matches beginning of the line
$	Matches end of the line
. (dot)	Matches any single character except newline. Using option *m*, then newline also can be matched
[…]	Matches any single character in brackets
[^…]	Matches any single character NOT in brackets
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more occurrence of preceding expression.
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
\d	Matches digits. Equivalent to [0-9].
\D	Matches non-digits.
\w	Matches word characters.
\W	Matches non-word characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].
\S	Matches non-whitespace.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\b	Matches the empty string, but only at the start or end of a word.
\B	Matches the empty string, but not at the start or end of a word.
( )	When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using findall()

Table : Examples for Regular Expressions

Expression	Description
[Pp]ython	Match "Python" or "python"

rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of uppercase, lowercase alphabets and digits
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

- Most commonly used metacharacter is dot, which matches any character.
- Consider the following example, where the regular expression is for searching lines which starts with I and has any two characters (any character represented by two dots) and then has a character m.

```
import re
fhand = open('myfile.txt')
for line in fhand:
        line = line.rstrip()
        if re.search('^I..m', line):
                    print(line)
```

The output would be –

```
I am doing fine.
```

- Note that, the regular expression ^I..m not only matches 'I am', but it can match 'Isdm', 'I*3m' and so on.
- That is, between Iand m, there can be any two characters.
- In the previous program, we knew that there are exactly two characters between I and m. Hence, we could able to give two dots.
- But, when we don't know the exact number of characters between two characters (or strings), we can make use of dot and + symbols together.
- Consider the below given program –

```
import re
hand = open('myfile.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^h.+u', line):
                print(line)
```

The output would be –

```
hello, how are you?
how about you?
```

- Observe the regular expression ^h.+u here.
- It indicates that, the string should be starting with h and ending with u and there may by any number of (dot and +) characters in- between.

Few examples:

- To understand the behavior of few basic meta characters, we will see some examples.
- The file used for these examples is *mbox-short.txt* which can be downloaded from – https://www.py4e.com/code3/mbox-short.txt
- Use this as input and try following examples –

- **Pattern to extract lines starting with the word *From* (or *from*) and ending with *edu*:**

```
import re
fhand = open('mbox-short.txt')
 for line in fhand:
        line =  line.rstrip()
        pattern = '^[Ff]rom.*edu$'
        if re.search(pattern,  line):
                print(line)
```

Here the pattern given for regular expression indicates that the line should start with either *From* or *from*. Then there may be 0 or more characters, and later the line should end with *edu*.

- **Pattern to extract lines ending with any digit:**

Replace the pattern by following string, rest of the program will remain the same.

```
pattern = '[0-9]$'
```

- **Using *Not* :**

```
pattern = '^[^a-z0-9]+'
```

Here, the first ^ indicates we want something to match in the beginning of a line. Then, the ^ inside square-brackets indicate *do not match any single character within bracket*. Hence, the whole meaning would be – line must be started with anything other than a lower-case alphabets and digits. In other words, the line should not be started with lowercase alphabet and digits.

- **Start with upper case letters and end with digits:**

```
pattern = '^[A-Z].*[0-9]$'
```

Here, the line should start with capital letters, followed by 0 or more characters, but must end with any digit.

→ Extracting Data using Regular Expressions

- Python provides a method ***findall()*** to extract all of the substrings matching a regular expression.
- This function returns a list of all non-overlapping matches in the string.
- If there is no match found, the function returns an empty list.
- Consider an example of extracting anything that looks like an email address from any line.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
```

```
lst = re.findall('\S+@\S+', s)
print(lst)
```

The output would be –

```
['csev@umich.edu', 'cwen@iupui.edu']
```

- Here, the pattern indicates at least one non-white space characters (\S) before @ and at least one non-white space after @.
- Hence, it will not match with @2pm, because of a white- space before @.
- Now, we can write a complete program to extract all email-ids from the file.

```
import re
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)
```

- Here, the condition len(x) > 0 is checked because, we want to print only the line which contain an email-ID. If any line do not find the match for a pattern given, the *findall()* function will return an empty list. The length of empty list will be zero, and hence we would like to print the lines only with length greater than 0.

The output of above program will be something as below –

```
['stephen.marquard@uct.ac.za'] ['<postmaster@collab.sakaiproject.org>']
['<200801051412.m05ECIaH010327@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;'] ['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;'] ['apache@localhost)']
……………………………….
………………………………..
```

- Note that, apart from just email-ID's, the output contains additional characters (<, >, ; etc) attached to the extracted pattern. To remove all that, refine the pattern. That is, we want email-ID to be started with any alphabets or digits, and ending with only alphabets. Hence, the statement would be –

```
x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
```

→ Combining Searching and Extracting

- Assume that we need to extract the data in a particular syntax.
- For example, we need to extract the lines containing following format –

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

- The line should start with X-, followed by 0 or more characters. Then, we need a colon and white-space. They are written as it is.
- Then there must be a number containing one or more digits with or without a decimal point. Note that, we want dot as a part of our pattern string, but not as meta character here. The pattern for regular expression would be –

```
^X-.*: [0-9.]+
```

The complete program is –

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)
```

The output lines will as below –

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6961
X-DSPAM-Probability: 0.0000
……………………………………………………
……………………………………………………
```

- Assume that, we want only the numbers (representing confidence, probability etc) in the above output.
- We can use *split()* function on extracted string. But, it is better to refine regular expression. To do so, we need the help of parentheses.
- When we add parentheses to a regular expression, they are ignored when matching the string. But when we are using ***findall()***, parentheses indicate that while we want the whole expression to match, we only are interested in extracting a portion of the substring that matches the regular expression.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X-\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)
```

- Because of the parentheses enclosing the pattern above, it will match the pattern starting with X- and extracts only digit portion. Now, the output would be –

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
```

```
['0.6961']
…………………
………………..
```

- Another example of similar form: The file *mbox-short.txt* contains lines like –

  Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
- We may be interested in extracting only the revision numbers mentioned at the end of these lines. Then, we can write the statement –
  ```
  x = re.findall('^Details:.*rev=([0-9.]+)', line)
  ```
- The regex here indicates that the line must start with Details:, and has something with rev= and then digits.
- As we want only those digits, we will put parenthesis for that portion of expression.
- Note that, the expression [0-9] is greedy, because, it can display very large number. It keeps grabbing digits until it finds any other character than the digit.
- The output of above regular expression is a set of revision numbers as given below –
  ```
  ['39772']
  ['39771']
  ['39770']
  ['39769']
  ………………………
  ………………………
  ```
- Consider another example – we may be interested in knowing time of a day of each email. The file *mbox-short.txt* has lines like –

  From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
- Here, we would like to extract only the hour 09. That is, we would like only two digits representing hour. Hence, we need to modify our expression as –
  ```
  x = re.findall('^From .* ([0-9][0-9]):', line)
  ```
- Here, [0-9][0-9] indicates that a digit should appear only two times.
- The alternative way of writing this would be -
  ```
  x = re.findall('^From .* ([0-9]{2}):', line)
  ```
- The number 2 within flower-brackets indicates that the preceding match should appear exactly two times.
- Hence [0-9]{2} indicates there can be exactly two digits.
- Now, the output would be –
  ```
  ['09']
  ['18']
  ['16']
  ['15']
  …………………
  …………………
  ```

→ Escape Character

- As we have discussed till now, the character like dot, plus, question mark, asterisk, dollar etc. are meta characters in regular expressions.

- Sometimes, we need these characters themselves as a part of matching string.
- Then, we need to escape them using a back- slash.
- For example,

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Output:

```
['$10.00']
```

- Here, we want to extract only the price $10.00. As, $ symbol is a metacharacter, we need to use \ before it.
- So that, now $ is treated as a part of matching string, but not as metacharacter.

→ Bonus Section for Unix/Linux Users

- Support for searching files using regular expressions was built into the Unix OS.
- There is a command-line program built into Unix called ***grep*** (Generalized Regular Expression Parser) that behaves similar to ***search()*** function.

```
$ grep '^From:'            mbox-short.txt
```

Output:

```
From: stephen.marquard@uct.ac.za From:
louis@media.berkeley.edu From:
zqian@umich.edu
From: rjlowe@iupui.edu
```

- Note that, *grep* command does not support the non-blank character \S, hence we need to use [^]indicating not a white-space.

MODULE IV

4.1 CLASSES AND OBJECTS

Programmer defined types, Attributes, Rectangles, Copying, Debugging

4.2 CLASSES AND FUNCTIONS

Time, Pure Functions, Modifiers, Prototyping vs Planning, Debugging

4.3 CLASSES AND METHODS

Object Oriented Features, The init Method and str Method, Operator Overloading, Type-based dispatch, Polymorphism, Interface and Implementation, Debugging

MODULE IV

4.1 CLASSES AND OBJECTS

- Python is an object-oriented programming language, and *class* is a basis for any object oriented programming language.
- Class is a user-defined data type which binds data and functions together into single entity.
- Class is just a prototype (or a logical entity/blue print) which will not consume any memory.
- An object is an instance of a class and it has physical existence.
- One can create any number of objects for a class.
- A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

→ Programmer-defined Types

- A class in Python can be created using a keyword class.
- Here, we are creating an empty class without any members by just using the keyword passwithin it.

```
class Point:
        pass

print(Point)
```

The output would be –

```
<class '_main__.Point'>
```

- The term_main_ indicates that the class Point is in the main scope of the current module.
- In other words, this class is at the top level while executing the program.
- Now, a user-defined data type Point got created, and this can be used to create any number of objects of this class.
- Observe the following statements:

```
p=Point()
```

- Now, a reference (for easy understanding, treat reference as a pointer) to Point object is created and is returned. This returned reference is assigned to the object p.
- The process of creating a new object is called as ***instantiation*** and the object is ***instance*** of a class.
- When we print an object, Python tells which class it belongs to and where it is stored in the memory.

```
print(p)
```

The output would be –

```
<_main_.Point object at 0x003C1BF0>
```

- The output displays the address (in hexadecimal format) of the object in the memory.
- It is now clear that, the object occupies the physical space, whereas the class does not.

→ Attributes

- An object can contain named elements known as ***attributes***.
- One can assign values to these attributes using dot operator.
- For example, keeping coordinate points in mind, we can assign two attributes x and y for the object of a class Point as below

```
p.x =10.0
p.y =20.0
```

- A state diagram that shows an object and its attributes is called as ***object diagram.***
- For the object p, the object diagram is shown in Figure below.

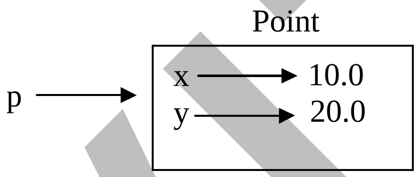


Figure : Object Diagram

- ***The diagram indicates that a variable (i.e. object) p refers to a Point object***, which contains two attributes.
- Each attributes refers to a floating point number.
- One can access attributes of an object as shown –

```
>>> print(p.x)
        10.0
>>> print(p.y)
        20.0
```

- Here, p.x means *"Go to the object p refers to and get the value of x".*
- Attributes of an object can be assigned to other variables

```
>>> x= p.x
>>> print(x)
         10.0
```

- Here, the variable x is nothing to do with attribute x.
- There will not be any name conflict between normal program variable and attributes of an object.
- **A complete program:** Write a class Point representing a point on coordinate system. Implement following functions –

➢ A function read_point() to receive x and y attributes of a Point object as user input.

- A function distance() which takes two objects of Point class as arguments and computes the Euclidean distance between them.
    - A function print_point()to display one point in the form of ordered-pair.

Program:

```
import math

class Point:
  """ This is a class Point representing a coordinate point"""

def read_point(p):
  p.x=float(input("x coordinate:"))
  p.y=float(input("y coordinate:"))

def print_point(p):
  print("(%g,%g)"%(p.x, p.y))

def distance(p1,p2):
  d=math.sqrt((p1.x-p2.x)**2+(p1.y-p2.y)**2)
  return d

p1=Point()                               #create first object
print("Enter First point:")
read_point(p1)                           #read x and y for p1

p2=Point()                               #create second object
print("Enter Second point:")
read_point(p2)                           #read x and y for p2

dist=distance(p1,p2)                     #compute distance
print("First point is:")
print_point(p1)                          #print p1
print("Second point is:")
print_point(p2)                          #print p2

print("Distance is: %g" %(distance(p1,p2)))  #print d
```

The sample output of above program would be –

```
Enter First point:
 x coordinate:10
y coordinate:20
Enter Second point:
 x coordinate:3
y coordinate:5
First point is: (10,20)
```

```
Second point is:(3,5)
Distance is: 16.5529
```

Let us discuss the working of above program thoroughly –

- The class Point contains a string enclosed within 3 double-quotes. This is known as ***docstring***. Usually, a string literal is written within 3 consecutive double-quotes inside a class, module or function definition. It is an important part of documentation and is to help someone to understand the purpose of the said class/module/function. The docstring becomes a value for the special attribute viz._ doc_ available for any class (and objects of that class). To get the value of docstring associated with a class, one can use the statements like –

```
>>> print(Point.__doc__)
  This is a class Point representing a coordinate point

>>> print(p1.__doc__)
  This is a class Point representing a coordinate point
```

  Note that, you need to type two underscores, then the word doc and again two underscores.In the above program, there is no need of docstring and we would have just used pass to indicate an empty class. But, it is better to understand the professional way of writing user-defined types and hence, introduced docstring.

- The function read_point() take one argument of type Point object. When we use the statements like,

```
read_point(p1)
```

  the parameter p of this function will act as an alias for the argument p1. Hence, the modification done to the alias p reflects the original argument p1. With the help of this function, we are instructing Python that the object p1 has two attributes x and y.

- The function print_point() also takes one argument and with the help of format- strings, we are printing the attributes x and y of the Point object as an ordered-pair (x,y).

- As we know, the Euclidean distance between two points (x1,y1) and (x2,y2) is

$$\sqrt{(x1-x2)^2+(y1-y2)^2}$$

  In this program, we have Point objects as (p1.x, p1.y) and (p2.x, p2.y). Apply the formula on these points by passing objects p1 and p2 as parameters to the function distance(). And then return the result.

Thus, the above program gives an idea of defining a class, instantiating objects, creating attributes, defining functions that takes objects as arguments and finally, calling (or invoking) such functions whenever and wherever necessary.

NOTE: User-defined classes in Python have two types of attributes viz. ***class attributes*** and ***instance attributes***. Class attributes are defined inside the class (usually, immediately after class header). They are common to all the objects of that class. That is, they are shared by all the objects created from that class. But, instance attributes defined for individual objects. They are available only for that instance (or object). Attributes of one instance are not available for another instance of the same class.
For example, consider the class Point as discussed earlier –

```
class Point:
        pass

p1= Point()                              #first object of the class
p1.x=10.0                                #attributes for p1
p1.y=20.0
print(p1.x, p1.y)                        #prints 10.0 20.0

p2= Point()                              #second object of the class
print(p2.x)                              #displays error as below

AttributeError: 'Point' object has no attribute 'x'
```

This clearly indicates that the attributes x and y created are available only for the object p1, but not for p2. Thus, x and y are instance attributes but not class attributes.

We will discuss class attributes late in-detail. But, for the understanding purpose, observe the following example –

```
class Point:
        x=2
        y=3

p1=Point()                               #first object of the class
print(p1.x, p1.y)                        # prints 2 3

p2=Point()                               #second object of the class
print(p2.x, p2.y)                        # prints 2 3
```

Here, the attributes x and y are defined inside the definition of the class Point itself. Hence, they are available to all the objects of that class.

→ Rectangles

- It is possible to make an object of one class as an attribute to other class.
- To illustrate this, consider an example of creating a class called as Rectangle.
- A rectangle can be created using any of the following data –
  - ➢ By knowing width and height of a rectangle and one corner point (ideally, a bottom- left corner) in a coordinate system
  - ➢ By knowing two opposite corner points

➢ Let us consider the first technique and implement the task: Write a class Rectangle containing numeric attributes width and height.

- This class should contain another attribute *corner* which is an instance of another class Point. Implement following functions –
  - A function to print corner point as an ordered-pair
  - A function *find_center()* to compute center point of the rectangle
  - A function *resize()* to modify the size of rectangle

The program is as given below –

```
class Point:
        """ This is a class Point representing coordinate point"""

class Rectangle:
        """ This is a class Rectangle. Attributes: width, height and Corner Point """

def find_center(rect):
        p=Point()
        p.x = rect.corner.x + rect.width/2
        p.y = rect.corner.y + rect.height/2
        return p

def resize(rect, w, h):
        rect.width +=w
        rect.height+=h

def print_point(p):
        print("(%g,%g)"%(p.x, p.y))


box=Rectangle()                     #create Rectangle object
box.corner=Point()                  #define an attribute corner for box
box.width=100                       #set attribute width to box
box.height=200                      #set attribute height to box
box.corner.x=0                      #corner itself has two attributes x and y
box.corner.y=0                      #initialize x and y to 0


print("Original Rectangle is:")
print("width=%g, height=%g"%(box.width, box.height))

center=find_center(box)
print("The center of rectangle is:")
print_point(center)

resize(box,50,70)
print("Rectangle after resize:")
print("width=%g, height=%g"%(box.width, box.height))
```

```
center=find_center(box)
print("The center of resized rectangle is:")
print_point(center)
```

A sample output would be:

```
Original Rectangle is: width=100, height=200
The center of rectangle is: (50,100)
 Rectangle after resize: width=150, height=270
 The center of resized rectangle is: (75,135)
```

The working of above program is explained in detail here –

- Two classes Pointand Rectanglehave been created with suitable docstrings. As of now, they do not contain any class-level attributes.
- The following statement instantiates an object of Rectangleclass.

  ```
  box=Rectangle()
  ```

  The statement

  ```
  box.corner=Point()
  ```

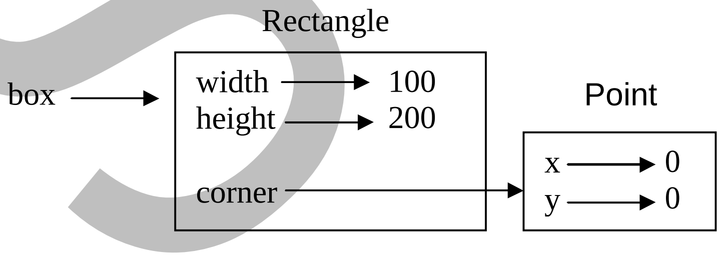
  indicates that corner is an attribute for the object box and this attribute itself is an object of the class Point. The following statements indicate that the object box has two more attributes

  ```
  box.width=100           #give any numeric value
  box.height=200          #give any numeric value
  ```

  In this program, we are treating the corner point as the origin in coordinate system and hence the following assignments –

  ```
  box.corner.x=0 box.corner.y=0
  ```

  (Note that, instead of origin, any other location in the coordinate system can be given as corner point.) Based on all above statements, an object diagram can be drawn as –



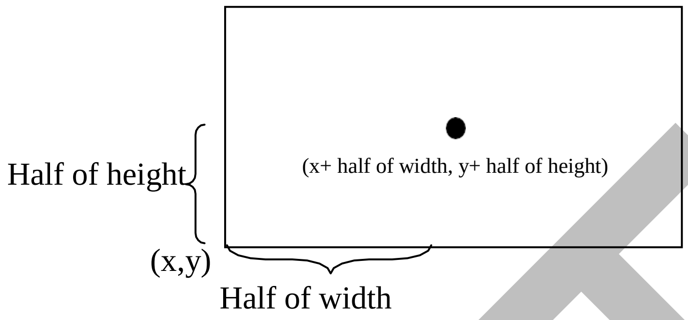
  The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."

- The function find_center() takes an object rect as an argument. So, when a call is made using the statement –

  ```
  center=find_center(box)
  ```

the object rect acts as an alias for the argument box.

A local object p of type Point has been created inside this function. The attributes of p are x and y, which takes the values as the coordinates of center point of rectangle. Center of a rectangle can be computed with the help of following diagram.



The function find_center() returns the computed center point. Note that, the return value of a function here is an instance of some class. That is, one can have an ***instance as return values*** from a function.

- The function resize() takes three arguments: rect – an instance of Rectangle class and two numeric variables w and h. The values w and h are added to existing attributes width and height. This clearly shows that ***objects are mutable***. State of an object can be changed by modifying any of its attributes. When this function is called with a statement –
  resize(box,50,70)
  the rect acts as an alias for box. Hence, width and height modified within the function will reflect the original object box.

Thus, the above program illustrates the concepts: ***Object of one class is made as attribute for object of another class, returning objects from functions*** and ***objects are mutable.***

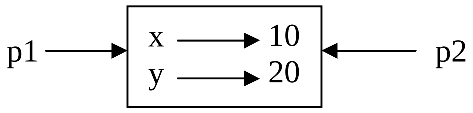
→ **Copying**

- An object will be aliased whenever there an object is assigned to another object of same class. This may happen in following situations –
  - Direct object assignment (like p2=p1)
  - When an object is passed as an argument to a function
  - When an object is returned from a function

- The last two cases have been understood from the two programs in previous sections.
- Let us understand the concept of aliasing more in detail using the following program

```
>>> class Point:
            pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20
>>> p2=p1
>>> print(p1)
        <_main_.Point object at 0x01581BF0>
>>> print(p2)
        <_main_.Point object at 0x01581BF0>
```

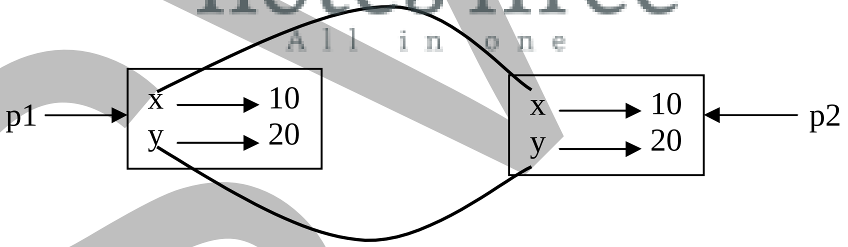
- Observe that both p1 and p2 objects have same physical memory. It is clear now that the object p2 is an alias for p1.
- So, we can draw the object diagram as below –



- Hence, if we check for equality and identity of these two objects, we will get following result.

```
>>> p1 is p2
        True
>>> p1==p2
        True
```

- But, the aliasing is not good always. For example, we may need to create a new object using an existing object such that – the new object should have a different physical memory, but it must have same attribute (and their values) as that of existing object. Diagrammatically, we need something as below –



- In short, ***we need a copy of an object, but not an alias.***
- To do this, Python provides a module called ***copy*** and a method called ***copy()***. Consider the below given program to understand the concept.

```
>>> class Point:
            pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20

>>> import copy                          #import module copy
>>> p3=copy.copy(p1)                     #use the method copy()
>>> print(p1)
```

```
            <_main_.Point object at 0x01581BF0>
    >>> print(p3)
            <_main_.Point object at 0x02344A50>
    >>> print(p3.x,p3.y)
           10 20
```

- Observe that the physical address of the objects p1 and p3 are now different.
- But, values of attributes xand y are same. Now, use the following statements –

```
    >>> p1 is p3
            False
    >>> p1 == p3
            False
```

- Here, the is operator gives the result as False for the obvious reason of p1 and p3 are being two different entities on the memory.
- But, why == operator is generating False as the result, though the contents of two objects are same? The reason is p1 and p3 are the objects of user-defined type.
- And, Python cannot understand the meaning of equality on the new data type. The default behavior of equality (==) is identity (is operator) itself. Hence, Python applies this default behavior on p1 == p3and results in False.

NOTE: If we need to define the meaning of equality (==) operator explicitly on user-defined data types (i.e. on class objects), then we need to override the method_eq_() inside the class. This will be discussed later in detail.

- The ***copy()*** method of ***copy*** module duplicates the object.
- The content (i.e. attributes) of one object is copied into another object as we have discussed till now.
- But, when an object itself is an attribute inside another object, the duplication will result in a strange manner.
- To understand this concept, try to copy Rectangle object (created in previous section) as given below

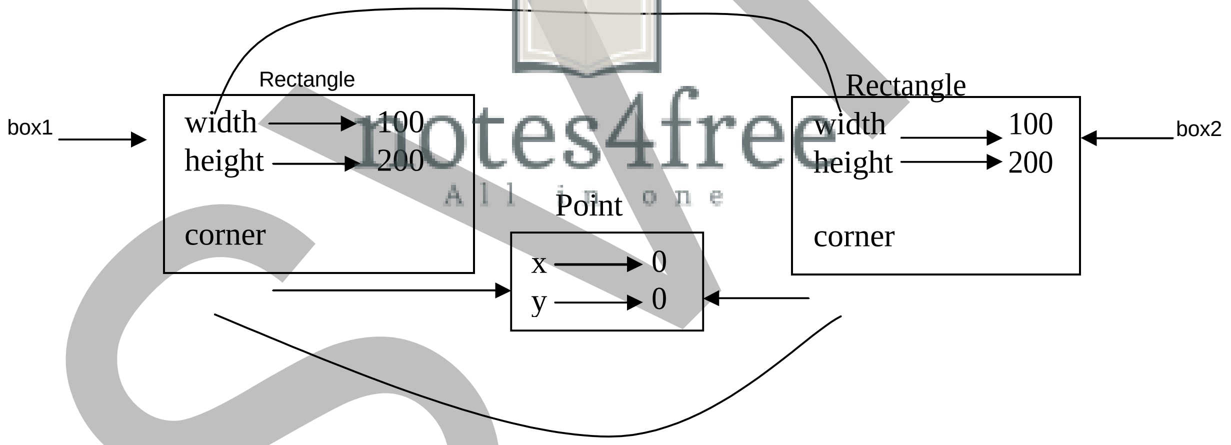
```
    import copy class
    Point:
          """ This is a class Point representing coordinate point"""

    class Rectangle:
          """ This is a class Rectangle.Attributes: width, height and Corner Point """

    box1=Rectangle()
    box1.corner=Point()
    box1.width=100
    box1.height=200
    box1.corner.x=0
    box1.corner.y=0
```

```
box2=copy.copy(box1)
print(box1 is box2)                                  #prints False
print(box1.corner is box2.corner)                    #prints True
```

- Now, the question is – why **box1.corner** and **box2.corner** are same objects, when **box1** and **box2** are different? Whenever the statement is executed,

```
box2=copy.copy(box1)
```

- The contents of all the attributes of box1 object are copied into the respective attributes of box2 object.
- That is, box1.width is copied into box2.width, box1.height is copied into box2.height.
- Similarly, box1.corner is copied into box2.corner.
- Now, recollect the fact that corner is not exactly the object itself, but it is a reference to the object of type Point (Read the discussion done for Figure at the beginning of this Chapter).
- Hence, the value of reference (that is, the physical address) stored in box1.corner is copied into box2.corner.
- Thus, the physical object to which box1.corner and box2.corner are pointing is only one.
- This type of copying the objects is known as ***shallow copy.***
- To understand this behavior, observe the following diagram



- Now, the attributes width and height for two objects box1 and box2 are independent.
- Whereas, the attribute corner is shared by both the objects.
- Thus, any modification done to box1.corner will reflect box2.corner as well.
- Obviously, we don't want this to happen, whenever we create duplicate objects. That is, we want two independent physical objects.
- Python provides a method ***deepcopy()*** for doing this task.
- This method copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on.

```
box3=copy.deepcopy(box1)
print(box1 is box3)                                  #prints False
print(box1.corner is box3.corner)                    #prints False
```

Thus, the objects box1 and box3 are now completely independent.

→ **Debugging**

- While dealing with classes and objects, we may encounter different types of errors.
- For example, if we try to access an attribute which is not there for the object, we will get *AttributeError*. For example –

```
>>> p= Point()
>>> p.x = 10
>>> p.y = 20
>>> print(p.z)
        AttributeError: 'Point' object has no attribute 'z'
```

- To avoid such error, it is better to enclose such codes within try/except as given below –

```
try:
        z = p.x
except AttributeError: z = 0
```

- When we are not sure, which type of object it is, then we can use ***type()*** as –

```
>>> type(box1)
        <class '_main__.Rectangle'>
```

- Another method ***isinstance()*** helps to check whether an object is an instance of a particular class

```
>>> isinstance(box1,Rectangle)
        True
```

- When we are not sure whether an object has a particular attribute or not, use a function hasattr() –

```
>>> hasattr(box1, 'width')
       True
```

- Observe the string notation for second argument of the function ***hasattr()***. Though the attribute width is basically numeric, while giving it as an argument to function ***hasattr()***, it must be enclosed within quotes.

4.2 CLASSES AND FUNCTIONS

- Though Python is object oriented programming languages, it is possible to use it as functional programming. There are two types of functions viz. ***pure functions*** and ***modifiers***.
- A pure function takes objects as arguments and does some work without modifying any of the original argument.
- On the other hand, as the name suggests, modifier function modifies the original argument.
- In practical applications, the development of a program will follow a technique called as ***prototype*** and ***patch***.
- That is, solution to a complex problem starts with simple prototype and incrementally dealing with the complications.

→ **Pure Functions**

- To understand the concept of pure functions, let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes.
- Write a function to print time in HH:MM:SS format and another function to add two time objects.
- Note that, adding two time objects should yield proper result and hence we need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.

```
class Time:
    """Represents the time of a day Attributes: hour, minute, second """

def printTime(t):
    print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))

def add_time(t1,t2):
    sum=Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
     if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum

t1=Time()
t1.hour=10
t1.minute=34
t1.second=25
print("Time1 is:")
printTime(t1)

t2=Time()
t2.hour=2
t2.minute=12
t2.second=41
print("Time2 is :")
printTime(t2)

t3=add_time(t1,t2)
print("After adding two time objects:")
printTime(t3)
```

The output of this program would be :

```
Time1 is: 10:34:25
Time2 is : 02:12:41
After adding two time objects: 12:47:06
```

- Here, the function add_time() takes two arguments of type Time, and returns a Time object, whereas, it is not modifying contents of its arguments t1 and t2.
- Such functions are called as *pure functions.*

→ Modifiers

- Sometimes, it is necessary to modify the underlying argument so as to reflect the caller.
- That is, arguments have to be modified inside a function and these modifications should be available to the caller.
- The functions that perform such modifications are known as ***modifier function.***
- Assume that, we need to add few seconds to a time object, and get a new time.
- Then, we can write a function as below

```
def increment(t, seconds):
        t.second += seconds

        while t.second >= 60:
                t.second -= 60
                t.minute += 1

        while t.minute >= 60:
                t.minute -= 60
                t.hour += 1
```

- In this function, we will initially add the argument seconds to t.second.
- Now, there is a chance that t.second is exceeding 60.
- So, we will increment minute counter till t.second becomes lesser than 60.
- Similarly, till the t.minute becomes lesser than 60, we will decrement minute counter.
- Note that, the modification is done on the argument t itself. Thus, the above function is a ***modifier.***

→ Prototyping v/s Planning

- Whenever we do not know the complete problem statement, we may write the program initially, and then keep of modifying it as and when requirement (problem definition) changes. This methodology is known as *prototype and patch.*
- That is, first design the prototype based on the information available and then perform patch-work as and when extra information is gathered.
- But, this type of incremental development may end-up in unnecessary code, with many special cases and it may be unreliable too.
- An alternative is *designed development*, in which high-level insight into the problem can make the programming much easier.
- For example, if we consider the problem of adding two time objects, adding seconds to time object

etc. as a problem involving numbers with base 60 (as every hour is 60 minutes and every minute is 60 seconds), then our code can be improved.
- Such improved versions are discussed later in this chapter.

→ **Debugging**
- In the program written inabove, we have treated time objects as valid values.
- But, what if the attributes (second, minute, hour) of time object are given as wrong values like negative number, or hours with value more than 24, minutes/seconds with more than 60 etc? So, it is better to write error-conditions in such situations to verify the input.
- We can write a function similar to as given below –

```
def valid_time(time):
        if time.hour < 0 or time.minute < 0 or time.second < 0:
                return False

        if time.minute >= 60 or time.second >= 60:
                return False

        return True
```

- Now, at the beginning of add_time()function, we can put a condition as –

```
def add_time(t1, t2):
        if not valid_time(t1) or not valid_time(t2):
                raise ValueError('invalid Time object in add_time')

                #remaining statements of add_time() functions
```

- Python provides another debugging statement ***assert***.
- When this keyword is used, Python evaluates the statement following it.
- If the statement is True, further statements will be evaluated sequentially. But, if the statement is False, then ***AssertionError*** exception is raised.
- The usage of *assert* is shown here –

```
def add_time(t1, t2):
        assert valid_time(t1) and valid_time(t2)
        #remaining statements of add_time() functions
```

- The *assert* statement clearly distinguishes the normal conditional statements as a part of the logic of the program and the code that checks for errors.

4.3 CLASSES AND METHODS
- The classes that have been considered till now were just empty classes without having any definition.
- But, in a true object oriented programming, a class contains class-level attributes, instance-level attributes, methods etc.

- There will be a tight relationship between the object of the class and the function that operate on those objects. Hence, the object oriented nature of Python classes will be discussed here.

→ Object-Oriented Features

As an object oriented programming language, Python possess following characteristics:

- ➢ Programs include class and method definitions.
- ➢ Most of the computation is expressed in terms of operations on objects.
- ➢ Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact.

- To establish relationship between the object of the class and a function, we must define a function as a member of the class. \
- function which is associated with a particular class is known as a ***method***.
- Methods are semantically the same as functions, but there are two syntactic differences:
  - ➢ Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
  - ➢ The syntax for invoking a method is different from the syntax for calling a function.
- Now onwards, we will discuss about classes and methods.

→ The __init__() Method

- A method init () has to be written with two underscores before and after the word *init*
- Python provides a special method called as init () which is similar to constructor method in other programming languages like C++/Java.
- The term *init* indicates initialization.
- As the name suggests, this method is invoked automatically when the object of a class is created. Consider the example given here –

```
import math

class Point:
        def init (self,a,b):
                self.x=a
                self.y=b

        def dist(self,p2):
                d=math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)
                 return d

        def str__(self):
                return "(%d,%d)"%(self.x, self.y)

p1=Point(10,20)                    # init () is called automatically
p2=Point(4,5)                      # init () is called automatically

print("P1 is:",p1)                 # str () is called automatically
 print("P2 is:",p2)                # str () is called automatically
```

```
d=p1.dist(p2)                    #explicit call for dist()

print("The distance is:",d)
```

The sample output is –

```
P1 is: (10,20)
P2 is: (4,5)
Distance is: 16.15549442140351
```

- Let us understand the working of this program and the concepts involved:
  - Keep in mind that every method of any class must have the first argument as ***self***. The argument ***self*** is a reference to the current object. That is, it is reference to the object which invoked the method. (Those who know C++, can relate ***self*** with ***this*** pointer). The object which invokes a method is also known as ***subject***.
  - The method init () inside the class is an initialization method, which will be invoked automatically when the object gets created. When the statement like –

    ```
    p1=Point(10,20)
    ```

    is used, the_init_() method will be called automatically. The internal meaning of the above line is –

    ```
    p1._init__(10,20)
    ```

    Here, p1 is the object which is invoking a method. Hence, reference to this object is created and passed to_init_() as *self*. The values 10 and 20 are passed to formal parameters a and b of init_() method. Now, inside_init_() method, we have statements

    ```
    self.x=10
    self.y=20
    ```

    This indicates, x and y are instance attributes. The value of x for the object p1 is 10 and, the value of y for the object p1is 20.

    When we create another object p2, it will have its own set of x and y. That is, memory locations of instance attributes are different for every object.

    Thus, state of the object can be understood by instance attributes.
  - The method dist() is an ordinary member method of the class Point. As mentioned earlier, its first argument must be self. Thus, when we make a call as –

    ```
    d=p1.dist(p2)
    ```

    a reference to the object p1 is passed as self to dist() method and p2 is passed explicitly as a second argument. Now, inside the dist()method, we are calculating distance between two point (Euclidian distance formula is used) objects. Note that, in this method, we cannot use the name p1, instead we will use self which is a reference (alias) to p1.

- The next method inside the class is_ ***str_ (). It is a special method used for string representation of user-defined object.*** Usually, print() is used for printing basic types in Python. But, user-defined types (class objects) have their own meaning and a way of representation. To display such types, we can write functions or methods like print_point() as we did in previous section But, more polymorphic way is to use_ str_ () so that, when we write just print() in the main part of the program, the str_ () method will be invoked automatically. Thus, when we use the statement like –

  ```
  print("P1 is:",p1)
  ```

  the ordinary print() method will print the portion "P1 is:" and the remaining portion is taken care by_str_() method. In fact,_str_() method will return the string format what we have given inside it, and that string will be printed by print() method.

→ Operator Overloading

- ***Ability of an existing operator to work on user-defined data type (class)*** is known as operator overloading.
- It is a polymorphic nature of any object oriented programming.
- Basic operators like +, -, * etc. can be overloaded.
- To overload an operator, one needs to write a method within user-defined class.
- Python provides a special set of methods which have to be used for overloading required operator.
- The method should consist of the code what the programmer is willing to do with the operator. Following table shows gives a list of operators and their respective Python methods for overloading.

Operator	Special Function in Python	Operator	Special Function in Python
+	__add__()	<=	__le__()
-	__sub__()	>=	__ge__()
*	__mul__()	==	__eq__()
/	__truediv__()	!=	__ne__()
%	__mod__()	in	__contains_()
<	__lt__()	len	__len__()
>	__gt__()	str	__str__()

- Let us consider an example of Point class considered earlier.
- Using operator overloading, we can try to add two point objects. Consider the program given below –

```
class Point:
        def _init_(self,a=0,b=0):
                self.x=a
                self.y=b

        def __add__(self, p2):
                p3=Point()
                p3.x=self.x+p2.x
                p3.y=self.y+p2.y
                 return p3

        def __str__(self):
                return "(%d,%d)"%(self.x, self.y)


p1=Point(10,20)
p2=Point(4,5)

print("P1 is:",p1)
print("P2 is:",p2)
p4=p1+p2                                    #call for add () method
print("Sum is:",p4)
```

The output would be –

```
P1 is: (10,20)
P2 is: (4,5)
Sum is: (14,25)
```

- In the above program, when the statement p4 = p1+p2 is used, it invokes a special method__add__() written inside the class. Because, internal meaning of this statement is–

  p4 = p1.__add__(p4)

Here, p1 is the object invoking the method. Hence, self inside_add_() is the reference (alias) of p1. And, p4 is passed as argument explicitly.

In the definition of__add__(), we are creating an object p3with the statement –

p3=Point()

The object p3 is created without initialization. Whenever we need to create an object with and without initialization in the same program, we must set arguments of_init_() for some default values. Hence, in the above program arguments a and b of_init_() are made as default arguments with values as zero. Thus, x and y attributes of p3will be now zero. In the add () method, we are adding respective attributes of self and p2 and storing in p3.x and p3.y. Then the object p3 is returned. This returned object is received as p4and is printed.

NOTE that, in a program containing operator overloading, the overloaded operator behaves in a normal way when basic types are given. That is, in the above program, if we use the statements

```
m= 3+4
print(m)
```

it will be usual addition and gives the result as 7. But, when user-defined types are used as operands, then the overloaded method is invoked.

- Let us consider a more complicated program involving overloading. Consider a problem of creating a class called Time, adding two Time objects, adding a number to Time object etc. that we had considered in previous section. Here is a complete program with more of OOP concepts.

```
class Time:
    def init (self, h=0,m=0,s=0):
        self.hour=h
        self.min=m
        self.sec=s

    def time_to_int(self):
        minute=self.hour*60+self.min
        seconds=minute*60+self.sec
        return seconds

    def int_to_time(self, seconds):
        t=Time()
        minutes, t.sec=divmod(seconds,60)
         t.hour, t.min=divmod(minutes,60)
        return t

    def __str__(self):
        return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)

    def __eq__(self,t):
        return self.hour==t.hour and self.min==t.min and self.sec==t.sec

    def __add__(self,t):
        if isinstance(t, Time):
            return self.addTime(t)
        else:
            return self.increment(t)

    def addTime(self, t):
          seconds=self.time_to_int()+t.time_to_int()
          return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int() return
```

```
            self.int_to_time(seconds)

        def_radd_(self,t):
            return self.add__(t)

 T1=Time(3,40)
 T2=Time(5,45)
 print("T1 is:",T1)
 print("T2 is:",T2)
 print("Whether T1 is same as T2?",T1==T2)  #call for_eq_()

 T3=T1+T2                              #call for_add__()

 print("T1+T2 is:",T3)

 T4=T1+75                              #call for_add_()
 print("T1+75=",T4)

 T5=130+T1                             #call for_radd_()
 print("130+T1=",T5)

 T6=sum([T1,T2,T3,T4])
 print("Using sum([T1,T2,T3,T4]):",T6)
```

The output would be –

```
T1 is: 03:40:00
T2 is: 05:45:00
Whether T1 is same as T2? False
 T1+T2 is: 09:25:00
T1+75= 03:41:15
130+T1= 03:42:10
Using sum([T1,T2,T3,T4]): 22:31:15
```

- Working of above program is explained hereunder –
  - The class Time has______init () method for initialization of instance attributes hour, min and sec. The default values of all these are being zero.
  - The method time_to_int() is used convert a Time object (hours, min and sec) into single integer representing time in number of seconds.
  - The method int_to_time() is written to convert the argument seconds into time object in the form of hours, min and sec. The built-in method ***divmod()*** gives the quotient as well as remainder after dividing first argument by second argument given to it.
  - Special method eq () is for overloading equality (==) operator. We can say one Time object is equal to the other Time object if underlying hours, minutes and seconds are equal respectively. Thus, we are comparing these instance attributes individually and returning either True of False.
  - When we try to perform addition, there are 3 cases –
    - Adding two time objects like T3=T1+T2.

  - Adding integer to Time object like T4=T1+75
  - Adding Time object to an integer like T5=130+T1

- Each of these cases requires different logic. When first two cases are considered, the first argument will be T1 and hence self will be created and passed to _add_() method.
- Inside this method, we will check the type of second argument using isinstance() method.
- If the second argument is Time object, then we call addTime() method. In this method, we will first convert both Time objects to integer (seconds) and then the resulting sum into Time object again
- So, we make use time_to_int() and int_to_time() here. When the 2nd argument is an integer it is obvious that it is number of seconds. Hence, we need to call increment() method.
- Thus, based on the type of argument received in a method, we take appropriate action. This is known as ***type-based dispatch.***
- In the 3rd case like T5=130+T1, Python tries to convert first argument 130 into self, which is not possible. Hence, there will be an error. This indicates that for Python, T1+5 is not same as 5+T1 (Commutative law doesn't hold good!!).
- To avoid the possible error, we need to implement ***right-side addition*** method_radd__(). Inside this method, we can call overloaded method_add__().
- The beauty of Python lies in surprising the programmer with more facilities!! As we have implemented_ add_ () method (that is, overloading of + operator), the built- in sum() will is capable of adding multiple objects given in a sequence. This is due to ***Polymorphism*** in Python**.**
- Consider a list containing Time objects, and then call sum()on that list as –

```
T6=sum([T1,T2,T3,T4])
```

- The sum() internally calls_add_() method multiple times and hence gives the appropriate result. Note down the square-brackets used to combine Time objects as a list and then passing it to sum().
- Thus, the program given here depicts many features of OOP concepts.

→ Debugging

- We have seen earlier that ***hasattr()*** method can be used to check whether an object has particular attribute.
- There is one more way of doing it using a method ***vars()***. This method maps attribute names and their values as a dictionary.
- For example, for the Point class defined earlier, use the statements

```
>>> p = Point(3, 4)
>>> vars(p)                    #output is {'y': 4, 'x': 3}
```

- For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

- Here, print_attributes() traverses the dictionary and prints each attribute name and its corresponding value.
- The built-in function getattr() takes an object and an attribute name (as a string) and returns the attribute values

MODULE V

5.1NETWORKED PROGRAMS

In this era of internet, it is a requirement in many situations to retrieve the data from web and to process it. In this section, we will discuss basics of network protocols and Python libraries available to extract data from web.

→ HyperText Transfer Protocol (HTTP)

- HTTP (HyperText Transfer Protocol) is the media through which we can retrieve web- based data.
- The **HTTP** is an application protocol for distributed and hypermedia information systems.
- HTTP is the foundation of data communication for the World Wide Web.
- Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text. HTTP is the protocol to exchange or transfer hypertext.
- Consider a situation:
  - ❖ you try to read a socket, but the program on the other end of the socket has not sent any data, then you need to wait.
  - ❖ If the programs on both ends of the socket simply wait for some data without sending anything, they will wait for a very long time.
- So an important part of programs that communicate over the Internet is to have some sort of protocol. A protocol is a set of precise rules that determine
  - ❖ Who will send request for what purpose
  - ❖ What action to be taken
  - ❖ What response to be given
- To send request and to receive response, HTTP uses GET and POST methods.

NOTE: To test all the programs in this section, you must be connected to internet.

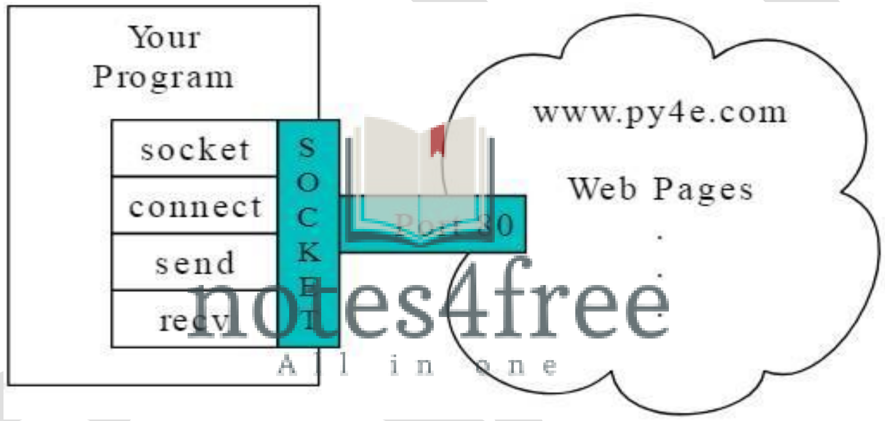
→ The World's Simplest Web Browser

- The built-in module ***socket*** of Python facilitates the programmer to make network connections and to retrieve data over those sockets in a Python program.
- ***Socket*** is bidirectional data path to a remote system.
- **A *socket is much like a file, except that a single socket provides a two-way connection* between two programs.**
- You can both read from and write to the same socket.
- If you write something to a socket, it is sent to the application at the other end of the socket.
- If you read from the socket, you are given the data which the other application has sent.
- Consider a simple program to retrieve the data from a web page. To understand the program given below, one should know the meaning of terminologies used there.

- ❖ **AF_INET** is an address family (IP) that is used to designate the type of addresses that your socket can communicate with.When you create a socket, you have to specify its address family, and then you can use only addresses of that type with the socket.
- ❖ **SOCK_STREAM** is a constant indicating the type of socket (TCP). It works as a file stream and is most reliable over the network.
- ❖ **Port** is a logical end-point. Port 80 is one of the most commonly used **port** numbers in the Transmission Control Protocol (TCP) suite.
- ❖ The command to retrieve the data must use CRLF(Carriage Return Line Feed) line endings, and it must end in \r\n\r\n (line break in protocol specification).
- ❖ ***encode()*** method applied on strings will return bytes-representation of the string. Instead of *encode()* method, one can attach a character *b* at the beginning of the string for the same effect.
- ❖ ***decode()*** method returns a string decoded from the given bytes.

Figure : A Socket Connection

- A socket connection between the user program and the webpage is shown in Figure below



- Now, observe the following program –

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 mysock.connect(('data.pr4e.org', 80))
cmd='GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if (len(data) < 1):
        break
    print(data.decode(),end='')
mysock.close()
```

- When we run above program, we will get some information related to web-server of the website which we are trying to scrape.
- Then, we will get the data written in that web-page. In this program, we are extracting 512 bytes of data at a time. (One can use one's convenient number here). The extracted data is decoded and printed. When the length of data becomes less than one (that is, no more data left out on the web page), the loop is terminated.

→ Retrieving an Image over HTTP

- In the previous section, we retrieved the text data from the webpage. Similar logic can be used to extract images on the webpage using HTTP.
- In the following program, we extract the image data in the chunks of 5120 bytes at a time, store that data in a string, trim off the headers and then store the image file on the disk.

```
import socket
import time

HOST = 'data.pr4e.org'                        #host name
PORT = 80                                     #port number

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))

mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')

count = 0
picture = b""                                 #empty string in binary format

while True:
        data = mysock.recv(5120)              #retrieve 5120 bytes at a time
        if (len(data) < 1):
                break

        time.sleep(0.25)              #programmer can see data retrieval easily
        count = count + len(data)
        print(len(data), count)               #display cumulative data retrieved
        picture = picture + data

mysock.close()

pos = picture.find(b"\r\n\r\n") #find end of the header (2 CRLF)
```

```
print('Header length', pos)
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]

fhand = open("stuff.jpg", "wb") #image is stored as stuff.jpg

fhand.write(picture) fhand.close()
```

- When we run the above program, the amount of data (in bytes) retrieved from the internet is displayed in a cumulative format.
- At the end, the image file ‘stuff.jpg’ will be stored in the current working directory. (One has to verify it by looking at current working directory of the program).

→ Retrieving Web Pages with urllib

- Python provides simpler way of webpage retrieval using the library ***urllib.***
- Here, webpage is treated like a file. ***urllib*** handles all of the HTTP protocol and header details.
- Following is the code equivalent to the program given above.

```
import urllib.request
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
        print(line.decode().strip())
```

- Once the web page has been opened with urllib.urlopen, we can treat it like a file and read through it using a for-loop.
- When the program runs, we only see the output of the contents of the file.
- The headers are still sent, but the urllib code consumes the headers and only returns the data to us.
- Following is the program to retrieve the data from the file romeo.txt which is residing at www.data.pr4e.org, and then to count number of words in it.

```
import urllib.request
fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
counts = dict()

for line in fhand:
        words = line.decode().split()
        for word in words:
            counts[word] = counts.get(word, 0) + 1
print(counts)
```

→ Reading Binary Files using urllib

- Sometimes you want to retrieve a non-text (or binary) file such as an image or video file.
- The data in these files is generally not useful to print out, but you can easily make a copy of a URL to a local file on your hard disk using ***urllib***.
- Above, we have seen how to retrieve image file from the web using sockets.
- Now, here is an equivalent program using ***urllib***.

```
import urllib.request img=urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
 fhand.close()
```

- Once we execute the above program, we can see a file cover3.jpg in the current working directory in our computer.
- The program reads all of the data in at once across the network and stores it in the variable ***img*** in the main memory of your computer, then opens the file cover.jpg and writes the data out to your disk.
- This will work if the size of the file is less than the size of the memory (RAM) of your computer.
- However, if this is a large audio or video file, this program may crash or at least run extremely slowly when your computer runs out of memory.
- In order to avoid memory overflow, we retrieve the data in blocks (or buffers) and then write each block to your disk before retrieving the next block.
- This way the program can read any size file without using up all of the memory you have in your computer.
- Following is another version of above program, where data is read in chunks and then stored onto the disk.

```
import urllib.request

img=urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0

while True:
        info = img.read(100000) if
        len(info) < 1:
                break
        size = size + len(info)
        fhand.write(info)
```

```
print(size, 'characters copied.') fhand.close()
```

- Once we run the above program, an image file cover3.jpg will be stored on to the current working directory.

→ Parsing HTML and Scraping the Web

- One of the common uses of the urllib capability in Python is to *scrape the web.*
- Web scraping is when we write a program that pretends to be a web browser and retrieves pages, then examines the data in those pages looking for patterns.
- Example: a search engine such as Google will look at the source of one web page and extract the links to other pages and retrieve those pages, extracting links, and so on.
- Using this technique, **Google *spiders its way through nearly all of the* pages on the web.**
- Google also uses the frequency of links from pages it finds to a particular page as one measure of how "important" a page is and how high the page should appear in its search results.

→ Parsing HTML using Regular Expressions

- Sometimes, we may need to parse the data on the web which matches a particular pattern.
- For this purpose, we can use regular expressions. Now, we will consider a program that extracts all the hyperlinks given in a particular webpage.
- To understand the Python program for this purpose, one has to know the pattern of an HTML file.
- Here is a simple HTML file –

```
<h1>The First Page</h1>
<p>
        If you like, you can switch to the
        <a href="http://www.dr-chuck.com/page2.htm"> Second Page</a>.

</p>
```

- Here,

<h1> and </h1>are the beginning and end of header tags
<p>and </p>are the beginning and end of paragraph tags
<a>and </a>are the beginning and end of anchor tag which is used for giving links
href is the attribute for anchor tag which takes the value as the link for another page.

- The above information clearly indicates that if we want to extract all the hyperlinks in a webpage, we need a regular expression which matches the href attribute. Thus, we can create a regular expression as –

```
href="http://.+?"
```

- Here, the question mark in .+? indicate that the match should find smallest possible matching string.
- Now, consider a Python program that uses the above regular expression to extract all hyperlinks

from the webpage given as input.

```
    import urllib.request import re
    url = input('Enter - ')                    #give URL of any website
    html = urllib.request.urlopen(url).read()
    links = re.findall(b'href="(http://.*?)"', html)

    for link in links:
          print(link.decode())
          ctx.check_hostname = False
          ctx.verify_mode = ssl.CERT_NONE

   url = input('Enter - ')

   html = urlopen(url, context=ctx).read()
   soup = BeautifulSoup(html, "html.parser")
    tags = soup('a')
   for tag in tags:
          print('TAG:', tag)
          print('URL:', tag.get('href', None))
          print('Contents:', tag.contents[0])
          print('Attrs:', tag.attrs)
```

The sample output would be –

```
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm"> Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Contents: Second Page
Attrs: {'href': 'http://www.dr-chuck.com/page2.htm'}
```

5.2 USING WEB SERVICES

- There are two common formats that are used while exchanging data across the web.
- One is HTML and the other is XML (eXtensible Markup Language).
- In the previous section we have seen how to retrieve the data from a web-page which is in the form of HTML.
- Now, we will discuss the retrieval of data from web-page designed using XML.
- XML is best suited for exchanging document-style data.
- When programs just want to exchange dictionaries, lists, or other internal information with each other, they use JavaScript Object Notation or JSON (refer www.json.org).
- We will look at both formats.

→ eXtensible Markup Language (XML)

- XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
        <name>Chuck</name>
        <phone type="intl"> +1 734 303 4456
        </phone>
        <email hide="yes"/>
</person>
```

- Often it is helpful to think of an XML document as a tree structure where there is a top tag person and other tags such as phone are drawn as children of their parent nodes.
- Figure is the tree structure for above given XML code.

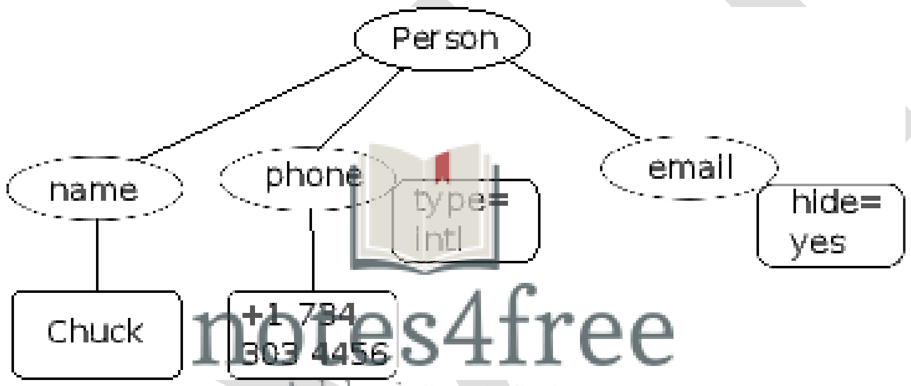


Figure : Tree Representation of XML

→ Parsing XML

- Python provides library xml.etree.ElementTree to parse the data from XML files.
- One has to provide XML code as a string to built-in method fromstring() of ElementTree class.
- ElementTree acts as a parser and provides a set of relevant methods to extract the data.
- Hence, the programmer need not know the rules and the format of XML document syntax.
- The fromstring() method will convert XML code into a tree-structure of XML nodes.
- When the XML is in a tree format, Python provides several methods to extract data from XML.
- Consider the following program.

```
import xml.etree.ElementTree as ET

#XML code embedded in a string format
data = '''
<person>
        <name>Chuck</name>
        <phone type="intl"> +1 734 303 4456
        </phone>
        <email hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print('Attribute for tag email:', tree.find('email').get('hide'))
print('Attribute for tag phone:', tree.find('phone').get('type'))
```

The output would be –

```
Name: Chuck
Attribute for the tag email: yes Attribute for the
tag phone: intl
```

- When we run this program, it prompts for user input.
- We need to give a valid URL of any website. Then all the hyperlinks on that website will be displayed.

→ Parsing HTML using BeautifulSoup

- There are a number of Python libraries which can help you parse HTML and extract data from the pages.
- Each of the libraries has its strengths and weaknesses and you can pick one based on your needs.
- ***BeautifulSoup** library* is one of the simplest libraries available for parsing.
- To use this, *download and install the BeautifulSoup code* from:

  http://www.crummy.com/software/

- Consider the following program which uses urllib to read the page and uses BeautifulSoup to extract href attribute from the anchor tag.

```
import urllib.request from bs4
import BeautifulSoup
import ssl                                    #Secure Socket Layer

ctx = ssl.create_default_context()
ctx.check_hostname = False
 ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url,context=ctx).read()
 soup = BeautifulSoup(html, 'html.parser')
```

```
tags = soup('a')

for tag in tags:
        print(tag.get('href', None))
```

A sample output would be –

```
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
```

- The above program prompts for a web address, then opens the web page, reads the data and passes the data to the BeautifulSoup parser, and then retrieves all of the anchor tags and prints out the href attribute for each tag.
- The BeautifulSoup can be used to extract various parts of each tag as shown below –

```
from urllib.request import urlopen from bs4
import BeautifulSoup import ssl

ctx = ssl.create_default_context()
```

- In the above example, fromstring() is used to convert XML code into a tree.
- The find() method searches XML tree and retrieves a node that matches the specified tag.
- The get() method retrieves the value associated with the specified attribute of that tag. Each node can have some text, some attributes (like hide), and some "child" nodes. Each node can be the parent for a tree of nodes.

→ Looping Through Nodes

- Most of the times, XML documents are hierarchical and contain multiple nodes.
- To process all the nodes, we need to loop through all those nodes.
- Consider following example as an illustration.

```
import xml.etree.ElementTree as ET
input = '''
<stuff>
        <users>
            <user x="2">
                <id>001</id>
                <name>Chuck</name>
            </user>
            <user x="7">
                <id>009</id>
                <name>Brent</name>
            </user>
    </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
        print('Name', item.find('name').text)
        print('Id', item.find('id').text)
        print('Attribute', item.get("x"))
```

The output would be –

```
User count: 2
Name Chuck
 Id 001
Attribute 2
Name Brent
 Id 009
Attribute 7
```

- The findall() method retrieves a Python list of subtrees that represent the user structures in the XML tree.
- Then we can write a for-loop that extracts each of the user nodes, and prints the name and id, which are text elements as well as the attribute x from the usernode.

→ JavaScript Object Notation (JSON)

- The JSON format was inspired by the object and array format used in the JavaScript language.
- But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON.
- So the format of JSON is a combination of Python lists and dictionaries.
- Following is the JSON encoding that is roughly equivalent to the XML code (the string data) given in the program of previous.

```
{
    "name" : "Chuck",
    "phone": {"type" : "intl", "number" : "+1 734 303 4456"}, "email": {"hide" : "yes"}
}
```

- Observe the differences between XML code and JSON code:
  - ❖ In XML, we can add attributes like "intl" to the "phone" tag. In JSON, we simply have key-value pairs.

❖ XML uses tag “person”, which is replaced by a set of outer curly braces in JSON.

- In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML.
- But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python’s dictionaries and lists
- JSON is a very natural format to have two compatible programs exchange data. JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

→ Parsing JSON

- Python provides a module json to parse the data in JSON pages.
- Consider the following program which uses JSON equivalent of XML string written in previous Section.
- Note that, the JSON string has to embed a list of dictionaries.

```
import json

data = ''' [
            { "id" : "001",
              "x" : "2",
          "name" : "Chuck" } ,
            { "id" : "009",
              "x" : "7",
            "name" : "Chuck"
      }
      ]'''

info = json.loads(data) print('User count:',
len(info))

for item in info:
        print('Name', item['name'])
        print('Id', item['id'])
        print('Attribute', item['x'])
```

The output would be –

```
User count: 2
Name Chuck
```

```
Id 001
Attribute 2
Name Chuck Id
009
Attribute 7
```

- Here, the string data contains a list of users, where each user is a key-value pair. The method loads() in the json module converts the string into a list of dictionaries.
- Now onwards, we don't need anything from json, because the parsed data is available in Python native structures.
- Using a for-loop, we can iterate through the list of dictionaries and extract every element (in the form of key-value pair) as if it is a dictionary object. That is, we use index operator (a pair of square brackets) to extract value for a particular key.

NOTE: Current IT industry trend is to use JSON for web services rather than XML. Because, JSON is simpler than XML and it directly maps to native data structures we already have in the programming languages. This makes parsing and data extraction simpler compared to XML. But XML is more self descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

→ Application Programming Interface (API)

- Till now, we have discussed how to exchange data between applications using HTTP, XML and JSON.
- The next step is to understand API. Application Programming Interface defines and documents the contracts between the applications.
- When we use an API, generally one program makes a set of services available for use by other applications and publishes the APIs (i.e., the "rules") that must be followed to access the services provided by the program.
- When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a ***Service-Oriented Architecture***(SOA).
- A SOA approach is one where our overall application makes use of the services of other applications.
- A non-SOA approach is where the application is a single stand-alone application which contains all of the code necessary to implement the application.
- Consider an example of SOA: Through a single website, we can book flight tickets and hotels. The data related to hotels is not stored in the airline servers. Instead, airline servers contact the services on hotel servers and retrieve the data from there and present it to the user.
- When the user agrees to make a hotel reservation using the airline site, the airline site uses another

web service on the hotel systems to actually make the reservation.

- Similarly, to reach airport, we may book a cab through a cab rental service.
- And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process. This process is depicted in Figure.

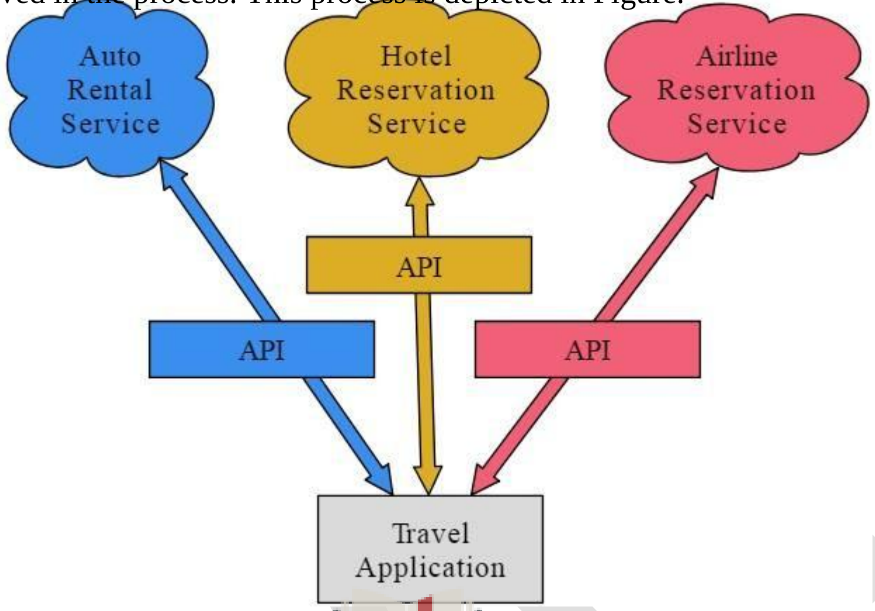


Figure : Server Oriented Architecture

- SOA has following major advantages:
  - ❖ we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit)
  - ❖ the owners of the data can set the rules about the use of their data.

- With these advantages, an SOA system must be carefully designed to have good performance and meet the user's needs. When an application makes a set of services in its API available over the web, then it is called as ***web services.***

→ Google Geocoding Web Service

- Google has a very good web service which allows anybody to use their large database of geographic information.
- We can submit a geographic search string like "Rajarajeshwari Nagar" to their geocoding API.
- Then Google returns the location details of the string submitted.
- The following program asks the user to provide the name of a location to be searched for.
- Then, it will call Google geocoding API and extracts the information from the returned JSON.

```
import urllib.request, urllib.parse, urllib.error
import json
```

```
serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'
 address = input('Enter location: ')
if len(address) < 1:
        exit()

url = serviceurl + urllib.parse.urlencode({'address': address})
print('Retrieving', url)
uh = urllib.request.urlopen(url)
data = uh.read().decode()
print('Retrieved', len(data), 'characters')


try:
        js = json.loads(data)
except:
        js = None

if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)


print(json.dumps(js, indent=4))
lat = js["results"][0]["geometry"]["location"]["lat"]
lng = js["results"][0]["geometry"]["location"]["lng"]
print('lat', lat, 'lng', lng)
location = js['results'][0]['formatted_address']
print(location)
```

(Students are advised to run the above program and check the output, which will contain several lines of Google geographical data).

- The above program retrieves the search string and then encodes it. This encoded string along with Google API link is treated as a URL to fetch the data from the internet. The data retrieved from the internet will be now passed to JSON to put it in JSON object format.
- If the input string (which must be an existing geographical location like Channasandra, Malleshwaram etc!!) cannot be located by Google API either due to bad internet or due to unknown location, we just display the message as ‘Failure to Retrieve’.
- If Google successfully identifies the location, then we will dump that data in JSON object.
- Then, using indexing on JSON (as JSON will be in the form of dictionary), we can retrieve the location address, longitude, latitude etc.

→ Security and API Usage

- Public APIs can be used by anyone without any problem.
- But, if the API is set up by some private vendor, then one must have ***API key*** to use that API.
- If API key is available, then it can be included as a part of POST method or as a parameter on the URL while calling API.
- Sometimes, vendor wants more security and expects the user to provide cryptographically signed messages using shared keys and secrets.
- The most common protocol used in the internet for signing requests is ***OAuth.***
- As the Twitter API became increasingly valuable, Twitter went from an open and public API to an API that required the use of OAuth signatures on each API request.
- But, there are still a number of convenient and free OAuth libraries so you can avoid writing an OAuth implementation from scratch by reading the specification.
- These libraries are of varying complexity and have varying degrees of richness.
- The OAuth web site has information about various OAuth libraries.

5.3 USING DATABASES AND SQL

- A structured set of data stored in a permanent storage is called as ***database***.
- Most of the databases are organized like a dictionary – that is, they map keys to values.
- Unlike dictionaries, databases can store huge set of data as they reside on permanent storage like hard disk of the computer.
- There are many database management softwares like Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite etc.
- They are designed to insert and retrieve data very fast, however big the dataset is.
- Database software builds *indexes* as data is added to the database so as to provider quicker access to particular entry.
- In this course of study, SQLite is used because it is already built into Python. SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a non-standard variant of the SQL query language.
- SQLite is designed to be *embedded* into other applications to provide database support within the application.
- For example, the Firefox browser also uses the SQLite database internally.
- SQLite is well suited to some of the data manipulation problems in Informatics such as the Twitter spidering application etc.

→ Database Concepts

- For the first look, database seems to be a spreadsheet consisting of multiple sheets.
- The primary data structures in a database are ***tables, rows*** and ***columns.***
- In a relational database terminology, tables, rows and columns are referred as ***relation, tuple*** and ***attribute*** respectively.

- Typical structure of a database table is as shown below.
- Each table may consist of n number of attributes and m number of tuples (or records).
- Every tuple gives the information about one individual.
- Every cell(i, j) in the table indicates value of j th attribute for i th tuple.

	Attribute1	**Attribute2**	**………………**	**Attribute_n**
Tuple1	V11	V12	………………	V1n
Tuple2	V21	V22	………………	V2n
…………..	………	…….	………………	……….
………….	…………	……….	……………..	………..
Tuple_m	Vm1	Vm2	……………..	Vmn

- Consider the problem of storing details of students in a database table. The format may look like –

	RollNo	**Name**	**DoB**	**Marks**
Student1	1	Ram	22/10/2001	82.5
Student2	2	Shyam	20/12/2000	81.3
…………..	………	…….	………………	……….
………….	…………	……….	……………..	………..
Student_m	…………..	……….	…………….	…………

- Thus, table columns indicate the type of information to be stored, and table rows gives record pertaining to every student.
- We can create one more table say *addressTable* consisting of attributes like DoorNo, StreetName, Locality, City, PinCode. To relate this table with a respective student stored in *studentTable,* we need to store RollNo also in *addressTable* (Note that, RollNo will be unique for every student, and hence there won't be any confusion).
- Thus, there is a relationship between two tables in a single database. There are softwares that can maintain proper relationships between multiple tables in a single database and are known as Relational Database Management Systems (RDBMS).

→ Structured Query Language (SQL) Summary

- To perform operations on databases, one should use structured query language.
- SQL is a standard language for storing, manipulating and retrieving data in databases.
- Irrespective of RDBMS software (like Oracle, MySQL, MS Access, SQLite etc) being used, the syntax of SQL remains the same.
- The usage of SQL commands may vary from one RDBMS to the other and there may be little syntactical difference.
- Also, when we are using some programming language like Python as a front-end to perform database applications, the way we embed SQL commands inside the program source-code is as

per the syntax of respective programming language.

- Still, the underlying SQL commands remain the same. Hence, it is essential to understand basic commands of SQL.
- There are some *clauses* like FROM, WHERE, ORDER BY, INNER JOIN etc. that are used with SQL commands, which we will study in a due course.
- The following table gives few of the SQL commands.

Command	Meaning
CREATE DATABASE	creates a new database
ALTER DATABASE	modifies a database
CREATE TABLE	creates a new table
ALTER TABLE	modifies a table
DROP TABLE	deletes a table
SELECT	extracts data from a database
INSERT INTO	inserts new data into a database
UPDATE	updates data in a database
DELETE	deletes data from a database

- As mentioned earlier, every RDBMS has its own way of storing the data in tables. Each of RDBMS uses its own set of data types for the attribute values to be used. SQLite uses the data types as mentioned in the following table –

Data Type	Description
NULL	The value is a NULL value.
INTEGER	The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
REAL	The value is a floating point value, stored as an 8-byte floating point number
TEXT	The value is a text string, stored using the database encoding (UTF- 8, UTF-16BE or UTF-16LE)
BLOB	The value is a blob (Binary Large Object) of data, stored exactly as it was input

- Note that, SQL commands are case-insensitive. But, it is a common practice to write commands and clauses in uppercase alphabets just to differentiate them from table name and attribute names.
- Now, let us see some of the examples to understand the usage of SQL statements –
  - ❖ CREATE TABLE Tracks (title TEXT, plays INTEGER)

    This command creates a table called as Tracks with the attributes title and plays where title can store data of type TEXT and playscan store data of type INTEGER.

  - ❖ INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)

    This command inserts one record into the table Tracks where values for the attributes title and plays are 'My Way' and 15 respectively.

  - ❖ SELECT * FROM Tracks

    Retrieves all the records from the table Tracks

  - ❖ SELECT * FROM Tracks WHERE title = 'My Way'

    Retrieves the records from the table Tracks having the value of attribute title as 'My Way'

  - ❖ SELECT title, plays FROM Tracks ORDER BY title

    The values of attributes title and plays are retrieved from the table Tracks with the records ordered in ascending order of title.

  - ❖ UPDATE Tracks SET plays = 16 WHERE title = 'My Way'

    Whenever we would like to modify the value of any particular attribute in the table, we can use UPDATE command. Here, the value of attribute plays is assigned to a new value for the record having value of title as 'My Way'.

  - ❖ DELETE FROM Tracks WHERE title = 'My Way'

    A particular record can be deleted from the table using DELETE command. Here, the record with value of attribute title as 'My Way' is deleted from the table Tracks.

→ Database Browser for SQLite

- Many of the operations on SQLite database files can be easily done with the help of software called *Database Browser for SQLite* which is freely available from:

  http://sqlitebrowser.org/

- Using this browser, one can easily create tables, insert data, edit data, or run simple SQL queries on the data in the database.
- This database browser is similar to a text editor when working with text files.

- When you want to do one or very few operations on a text file, you can just open it in a text editor and make the changes you want.
- When you have many changes that you need to do to a text file, often you will write a simple Python program.
- You will find the same pattern when working with databases. You will do simple operations in the database manager and more complex operations will be most conveniently done in Python.

→ Creating a Database Table

- When we try to create a database table, we must specify the names of table columns and the type of data to be stored in those columns.
- When the database software knows the type of data in each column, it can choose the most efficient way to store and look up the data based on the type of data.
- Here is the simple code to create a database file and a table named Tracks with two columns in the database:

Ex1.

```
import sqlite3
conn = sqlite3.connect('music.sqlite')
 cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)') conn.close()
```

- The connect() method of sqlite3 makes a "connection" to the database stored in the file music.sqlite3 in the current directory.
- If the file does not exist, it will be created.
- Sometimes, the database is stored on a different database server from the server on which we are running our program.
- But, all the examples that we consider here will be local file in the current working directory of Python code.
- A cursor() is like a file handle that we can use to perform operations on the data stored in the database. Calling cursor() is very similar conceptually to calling open() when dealing with text files.
- Hence, once we get a cursor, we can execute the commands on the contents of database using execute()method.

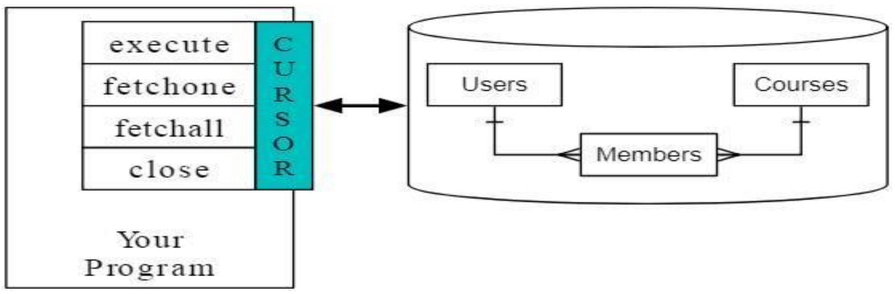


Figure : A Database Cursor

- In the above program, we are trying to remove the database table Tracks, if at all it existed in the current working directory.
- The DROP TABLE command deletes the table along with all its columns and rows.
- This procedure will help to avoid a possible error of trying to create a table with same name.
- Then, we are creating a table with name Tracks which has two columns viz. title, which can take TEXT type data and plays, which can take INTEGER type data.
- Once our job with the database is over, we need to close the connection using close()method.
- In the previous example, we have just created a table, but not inserted any records into it
- So, consider below given program, which will create a table and then inserts two rows and finally delete records based on some condition.

Ex2.

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

cur.execute("INSERT INTO Tracks (title, plays) VALUES ('Thunderstruck', 20)")
cur.execute("INSERT INTO Tracks (title, plays) VALUES (?, ?)", ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
        print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')
 cur.close()
```

- In the above program, we are inserting first record with the SQL command –
  "INSERT INTO Tracks (title, plays) VALUES('Thunderstruck', 20)"
- Note that, execute() requires SQL command to be in string format. But, if the value to be store in the table is also a string (TEXT type), then there may be a conflict of string representation using quotes.
- Hence, in this example, the entire SQL is mentioned within double-quotes and the value to be inserted in single quotes. If we would like to use either single quote or double quote everywhere, then we need to use escape-sequences like \' or \".
- While inserting second row in a table, SQL statement is used with a little different syntax –
  "INSERT INTO Tracks (title, plays) VALUES (?, ?)",('My Way', 15)
- Here, the question mark acts as a place-holder for particular value.
- This type of syntax is useful when we would like to pass user-input values into database table.
- After inserting two rows, we must use commit() method to store the inserted records permanently on the database table.
- If this method is not applied, then the insertion (or any other statement execution) will be temporary and will affect only the current run of the program.
- Later, we use SELECT command to retrieve the data from the table and then use for-loop to display all records.
- When data is retrieved from database using SELECT command, the cursor object gets those data as a list of records.
- Hence, we can use for-loop on the cursor object. Finally, we have used a DELETE command to delete all the records WHERE plays is less than 100.

Let us consider few more examples –

Ex3.

```
import sqlite3
from sqlite3 import Error

def create_connection():
    """ create a database connection to a database that resides in the memory"""
    try:

        conn = sqlite3.connect(':memory:')
        print("SQLite Version:",sqlite3.version)
        except Error as e:
                print(e)
        finally:
                conn.close()
create_connection()
```

Few points about above program:

- ❖ Whenever we try to establish a connection with database, there is a possibility of error due to non-existing database, authentication issues etc. So, it is always better to put the code for connection inside try-except block.
- ❖ While developing real time projects, we may need to create database connection and close it every now-and-then. Instead of writing the code for it repeatedly, it is better to write a separate function for establishing connection and call that function whenever and wherever required.
- ❖ If we give the term :memory: as an argument to connect() method, then the further operations (like table creation, insertion into tables etc) will be on memory (RAM) of the computer, but not on the hard disk.

Ex4. Write a program to create a Student database with a table consisting of student name and age. Read n records from the user and insert them into database. Write queries to display all records and to display the students whose age is 20.

```
import sqlite3 conn=sqlite3.connect('StudentDB.db') c=conn.cursor()
c.execute('CREATE TABLE tblStudent(name text, age Integer)')

n=int(input("Enter number of records:")) for i in range(n):
        nm=input("Enter Name:")
        ag=int(input("Enter age:"))
        c.execute("INSERT INTO tblStudent VALUES(?,?)",(nm,ag))

conn.commit()
c.execute("select * from tblStudent ") print(c.fetchall())

c.execute("select * from tblStudent where age=20") print(c.fetchall())

conn.close()
```

In the above program we take a for-loop to get user-input for student's name and age. These data are inserted into the table. Observe the question mark acting as a placeholder for user-input variables. Later we use a method fetchall() that is used to display all the records form the table in the form of a list of tuples. Here, each tuple is one record from the table.

→ Three Kinds of Keys

Sometimes, we need to build a data model by putting our data into multiple linked tables and linking the rows of those tables using some ***keys***. There are three types of keys used in database model:

- ❖ A ***logical key*** is a key that the "real world" might use to look up a row. It defines the relationship between primary keys and foreign keys. Most of the times, a UNIQUE constraint is added to a logical key. Since the logical key is how we look up a row from the outside world, it makes little sense to allow multiple rows with the same value in the table.
- ❖ A ***primary key*** is usually a number that is assigned automatically by the database. It generally has no meaning outside the program and is only used to link rows from different tables together. When we want to look up a row in a table, usually searching for the row using the primary key is the fastest way to find the row. Since primary keys are integer numbers, they take up very little storage and can be compared or sorted very quickly.
- ❖ A ***foreign key*** is usually a number that points to the primary key of an associated row in a different table.

- Consider a table consisting of student details like RollNo, name, age, semester and address as shown below –

RollNo	Name	Age	Sem	Address
1	Ram	29	6	Bangalore
2	Shyam	21	8	Mysore
3	Vanita	19	4	Sirsi
4	Kriti	20	6	Tumkur

- In this table, RollNo can be considered as a primary key because it is unique for every student in that table. Consider another table that is used for storing marks of students in all the three tests as below

RollNo	Sem	M1	M2	M3
1	6	34	45	42.5
2	6	42.3	44	25
3	4	38	44	41.5
4	6	39.4	43	40
2	8	37	42	41

- To save the memory, this table can have just RollNo and marks in all the tests. There is no need to store the information like name, age etc of the students as these information can be retrieved from first table. Now, RollNo is treated as a foreign key in the second table.

→ Basic Data Modeling

- The relational database management system (RDBMS) has the power of linking multiple tables. The act of deciding how to break up your application data into multiple tables and establishing the relationships between the tables is called ***data modeling***.
- The design document that shows the tables and their relationships is called a ***data model***. Data modeling is a relatively sophisticated skill.
- The data modeling is based on the concept of ***database normalization*** which has certain set of rules.
- In a raw-sense, we can mention one of the basic rules as never put the same string data in the database more than once. If we need the data more than once, we create a numeric ***key*** (primary key) for the data and reference the actual data using this key.
- This is because string requires more space on the disk compared to integer, and data retrieval (by comparing) using strings is difficult compared to that with integer.
- Consider the example of Student database discussed above.
- We can create a table using following SQL command –

```
CREATE TABLE tblStudent
        (RollNo INTEGER PRIMARY KEY, Name TEXT, age INTEGER, sem INTEGER, address
        TEXT)
```

Here, RollNo is a primary key and by default it will be unique in one table. Now, another take can be created as –

```
CREATE TABLE tblMarks
        (RollNo INTEGER, sem INTEGER, m1 REAL, m2 REAL, m3 REAL,
        UNIQUE(RollNo,sem))
```

- Now, in the tblMarks consisting of marks of 3 tests of all the students, RollNo and sem are together unique. Because, in one semester, only one student can be there having a particular RollNo. Whereas in another semester, same RollNo may be there.
- Such types of relationships are established between various tables in RDBMS and that will help better management of time and space.

→ Using JOIN to Retrieve Data

- When we follow the rules of database normalization and have data separated into multiple tables, linked together using primary and foreign keys, we need to be able to build a SELECT that reassembles the data across the tables.
- SQL uses the JOIN clause to reconnect these tables. In the JOIN clause you specify the fields that are used to reconnect the rows between the tables.

- Consider the following program which creates two tables tblStudent and tblMarks as discussed in the previous section.
- Few records are inserted into both the tables. Then we extract the marks of students who are studying in 6th semester.

```
import sqlite3

conn=sqlite3.connect('StudentDB.db')
c=conn.cursor()

c.execute('CREATE TABLE tblStudent
        (RollNo  INTEGER PRIMARY KEY, Name TEXT, age INTEGER, sem
        INTEGER, address TEXT)')

c.execute('CREATE TABLE tblMarks
        (RollNo  INTEGER, sem INTEGER, m1 REAL, m2 REAL, m3 REAL,
        UNIQUE(RollNo,sem))')

c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (1,'Ram',20,6,'Bangalore'))
c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (2,'Shyam',21,8,'Mysore'))
c.execute("INSERT INTO tblstudent VALUES(?,?,?,?,?)",
                                        (3,'Vanita',19,4,'Sirsi')) c.execute("INSERT
INTO tblstudent VALUES(?,?,?,?,?)",
                                        (4,'Kriti',20,6,'Tumkur'))


c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(1,6,34,45,42.5))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(2,6,42.3,44,25))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(3,4,38,44,41.5))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(4,6,39.4,43,40))
c.execute("INSERT INTO tblMarks VALUES(?,?,?,?,?)",(2,8,37,42,41))

conn.commit()

query="SELECT tblStudent.RollNo, tblStudent.Name, tblMarks.sem, tblMarks.m1,
        tblMarks.m2, tblMarks.m3 FROM tblStudent JOIN tblMarks ON tblStudent.sem =
        tblMarks.sem AND tblStudent.RollNo = tblMarks.RollNo WHERE tblStudent.sem=6"

c.execute(query)
```

```
for row in c:
        print(row)
conn.close()
```

The output would be –

```
(1, 'Ram', 6, 34.0, 45.0, 42.5)
(4, 'Kriti', 6, 39.4, 43.0, 40.0)
```

The query joins two tables and extracts the records where RollNo and sem matches in both the tables, and sem must be 6.