110+20=130

MODULE-1: Chapters 1, 2, 3 from T1

Contents:

Python Basics: Entering Expressions into the Interactive Shell, the Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program

Flow control: Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys. exit()

Functions: Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number

Chapter 1

Python Basics

➢ Python – Introduction

- ***Beginner's Language***
- Developed by **Guido van Rossum** in the late eighties and early nineties at the ***National Research Institute for Mathematics and Computer Science*** in the Netherlands.
- Named after a British comedy **"Monty Python's Flying Circus"**
- Derived from many other languages, including Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other Scripting languages like Perl, PHP.
- ***High-level, Interpreted, Interactive, Object-Oriented and Scripting language.***
- Designed to be highly ***Readable*** **and** ***Reliable***
- **Free and Open Source,** Like Perl and PHP, Python source code is now available under the GNU General Public License (GPL).
- **Most *preferred language*** in companies like **Google, YouTube, and NASA**
- Official website **http://www.python.org**
- **Current version: 3.8.3,** you can look at latest version also.

➢ Python - Features

- *Easy-to-learn, Easy-to-read, Easy-to-maintain*

- ***A broad standard library***
- *Portable, Extendable, Scalable*
- ***Databases, GUI Programming***
- Supports ***functional and structured programming*** methods as well as ***Object Oriented Programming.***
- Can be used as a ***scripting language*** or can be ***compiled to byte-code*** for building large applications.
- Provides *high-level dynamic data types* and supports *dynamic type checking.*
- Supports ***automatic garbage collection.***
- Can be easily *integrated with* ***C, C++, CORBA, and Java.***

➢ Where to use Python

- System Management (scripting)
- Graphical User Interface (GUI)
- Networking and Internet programming
- Database (DB) programming
- Text data processing
- Distributed processing
- Numerical operations
- Graphics
- Image Processing
- AI and Machine Learning

  And so on…

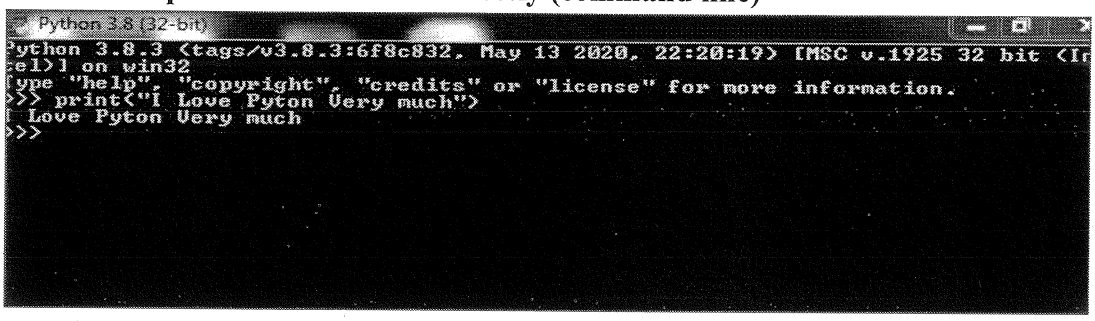
➢ Python Programming Basics

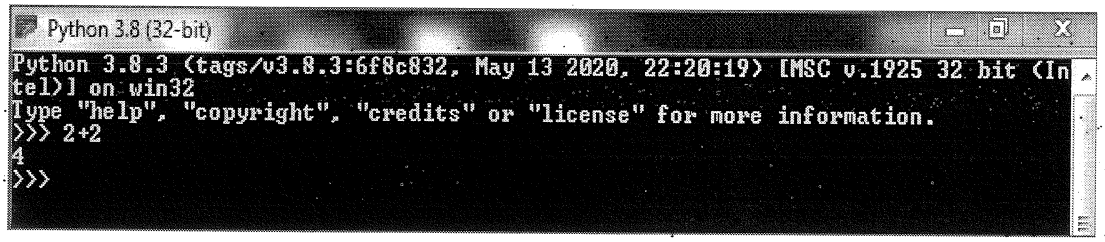
✓ Entering Expressions into the Interactive Shell

- **Download Python from www.python.org and install.**
- Access python console in two ways:
  1. Using **command prompt >>>,** you can run python interpreter directly (command line)
  2. Using **GUI** called Python's **Integrated Development Learning Environment (IDLE)**
- **Python files have extension .py**
- Call python program via the python interpreter

  % python fact.py
- **Run the interactive shell by launching IDLE** - installed with Python
- **On Windows**, open the **Start menu**, select **All Programs → Python 3.8 → IDLE (3.8, 32/64-bit)**

- A window with >>> prompt should appear - that is the **interactive shell** as below- with **print statement** and **directly (command line)**





Running Programs on UNIX

- An example of a python program run as a script
- Adding an appropriate path to your python interpreter as the first line of your file

#!/usr/bin/python
print ("Hello, world")

If this is saved to a file **hello.py**, then set execute permissions on this file **(chmod u+x hello.py in Unix/Linux), and run it with Unix command line**
% ./hello.py
When run, this program prints
Hello, world

Running Python

- Once you're **inside the Python interpreter**, type in commands
- Example:

>>> print ('Hello world')
Hello world
Relevant output is displayed on subsequent lines without the >>> **symbol**
>>> x = [0,1,2]
Quantities stored in memory are not displayed by default **(comment line with #)**
If a quantity is stored in memory, typing its name will display it
>>> x
[0,1,2]
>>> 2+3
5

```
>>> # Type ctrl-D to exit the interpreter
%
```

➢ Enough to Understand the code

- **Indentation** matters to **code meaning**
  - **Block structure** indicated by **indentation**
- **First assignment** to a variable **creates it**
  - Variable types **don't need to be declared.**
  - Python figures out the **variable types** on its own.
- Assignment is = and comparison is ==
- For numbers + - * / % // are as expected
  - **Special use of +** for **string concatenation** and **% for string formatting** (as in C's printf)
- Logical operators **are words (and, or, not)** not symbols
- The basic printing command is **print**

➢ Character Set, Words, Sentences

- **Characters:**
  - digits(0-9), letters([a-z],[A-Z]), special symbols(+, -,…, %, #, :, $, !, <, >,… etc.,
- **Python Identifier:**
  - A name used to identify a variable, function, class, module, or other object. Ex: a, student, subject, std_cs12
  - Starts with a letter or an underscore (_) followed by zero or more letters, underscores and digits.
  - No punctuation characters such as @, $, and % within identifiers.
- **Python is a case sensitive programming language.**
  - **Manpower and manpower** are two different identifiers
- **Python Sentence / Statement:**

```
print('Sample Sentences !')    # Comment statement
x,y = 3,6                      # Assignment statement / Computation
If (x > y):                    # Conditional statement / looping
    print(' Max:', x)          # Output statement / input
```

- **A *statement* is a unit of code that the Python interpreter can execute**

➢ Python Keywords

- These are reserved words that have very special meaning to Python and you cannot use them as **constant or variable** or any other **identifier names.**
- All the **Python keywords contain lowercase letters** only.

and	continue	except	if	not	return
as	def	from	in	nonlocal	try
assert	del	finally	is	or	while
break	elif	for	import	pass	with
class	else	global	lambda	raise	yield
True	False	None			

➢ Python Program Structure

- Statements in Python typically end with a **new line**
- Statements contained within the **[], {}, or ()** brackets do not need to use the line continuation character.
- Python provides **no braces** to indicate blocks of code for class and **function definitions or flow control.**
  - Blocks of code are denoted by line indentation, which is rigidly enforced.
  - The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.
- Python accepts single ('), double (") and triple (""" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- A hash **sign (#)** that is not inside a string literal begins a **comment.** Python interpreter ignores them
- **Modules, Packages, Classes**

➢ Values and Type

- Primitive / Basic
  - **Integers: 24, 325634L**
  - **Floating Point: 32.3, 3.1E2**
  - **Complex: 3 + 2j, 1j**
  - **String: 'R', "RNSIT", """RNS"""**
  - **Boolean: True, False**
- Non-primitive / Advanced
  - **Lists: l = [1,2,3]**
  - **Tuples: t = (1,2,3)**
  - **Sets: s={'A', 'e','3','RNS'}**
  - **Dictionaries: d = {'hello' : 'there', 2 : 15}**

- Example:

use of **types.py:**

```
pi = 3.141596
message = "Hello, world"
i = 2+4
print type(pi)
print type(message)
print type(i)
```

types of individual is printed

Output:

```
<type 'float'>
<type 'str'>
<type 'int'>
```

- **Summary**
  - **Numbers: int, long, float, complex**
  - **Boolean: True, False**
  - **Strings and Tuples: immutable**
  - **Lists and Dictionaries: mutable**
  - **Sets: no duplicates**

➢ Variables and Storing Values in Variables

- **Variable**: is a name that refers to a value

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.14159265
```

- **Assignment statement**: An assignment statement creates new variables and assigns them values
- Syntax: ***name*** = ***value***
- Examples: **x = 5**

gpa = 3.14

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

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

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
>>> type(3.2)
<class 'float'>
```

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

```
>>> print(x)
5
>>> print(gpa)
3.14
```

- **You can assign to multiple names at the same time**
  `>>> x, y = 2, 3`
- **Assignments can be chained**
  `>>> a = b = x = 2`
- **Actually, "variables" in python are really *object references*.**
  - No need to declare.
  - Can contain letters, numbers, and underscores
  - Must begin with a **letter**
  - **Cannot be one of the Python keywords**
  - Everything is a "variable": Even functions, classes, modules
  - Names starting with **one underscore** (_V) are not imported from the module import * statement
  - Names starting and ending with **2 underscores** are special, system-defined names (e.g., __V__)
  - Names beginning with **2 underscores** (but without trailing underscores) are local to a class (__V)
  - A **single underscore** (_) by itself denotes the result of the last expression

- **Illegal variable name, 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 @.
class is one of Python's *keywords*.

Valid variable names	Invalid variable names
balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
current_balance	4account (can't begin with a number)
_spam	42 (can't begin with a number)
SPAM	total_$um (special characters like $ are not allowed)
account4	'hello' (special characters like ' are not allowed)

➢ Operators

- *Operators* are special symbols that represent **computations/operations**
- The values the operator is applied to are called ***operands***
- **Operations : Arithmetic, Relational, Logical, Bitwise, Assignment**
- **The operators +, -, *, /, %, // and ** perform arithmetic operations such as addition, subtraction, multiplication, division, modulo, integer division and exponentiation** respectively.
- **Similarly**
  - **Relational: <, >, <=, >=, !=, ==**
  - **Logical: and, or, not**
  - **Bitwise: &, |, ~, ^, <<, >>**
  - **Unary: -, !, ~, ++, --, not**
  - **Special: Membership (in, not in), Identity (is, is not)**

✓ Order of Operations (precedence of operators)

- When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*.
- For mathematical operators, Python follows the acronym ***PEMDAS*** as a useful way to remember the rules:
  - ▶ *P*arentheses 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.**

( )	**Parenthesis (inner before outer)**
**	**Exponent**
*, /, %	**Multiplication, division, modulo**
+, -	**Addition, subtraction**

➢ 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 (assuming that the variable x has been assigned a value):

Examples: **42**
x
x + 17

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

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

✓ Relational and Logical Expressions

- Relational expressions use ***relational operators*:**

Operator	Meaning	Example	Result
==	equals	1 + 1 == 2	True
!=	does not equal	3.2 != 2.5	True
<	less than	10 < 5	False
>	greater than	10 > 5	True
<=	less than or equal to	126 <= 100	False
>=	greater than or equal to	5.0 >= 5.0	True

- **Relational expressions** can be **combined with *logical operators*** → **Logical Expression**

Operator	Example	Result
and	9 != 6 and 2 < 3	True
or	2 == 3 or -1 < 5	True
not	not 7 > 0	False
Shift	a<<b, a>>b	

➢ Output Statement: print

- print : Produces text output on the console.
- Syntax:

  print ("Message")

  print(Expression)

  ▶ Prints the given text message or expression value on the console, and moves the cursor down to the next line.

  Print(Item1, Item2, ..., ItemN)

  ▶ Prints several messages and/or expressions on the same line.
- Examples:

```
print ("Hello, world!")
age = 45
print( "You have", 65 - age, "years until retirement")
```

- Output:

```
Hello, world!
You have 20 years until retirement
```

➢ Input Statement: input

- **input: Reads a string from user input.**
  - ▶ You can assign (store) the result of input into a variable.
  - ▶ Input returns what the **user typed as a string**. The **string** can be converted by using the conversion methods int(string), float(string), etc as in **snapshot below**:
- Syntax:

  <variable name> = input()

  OR

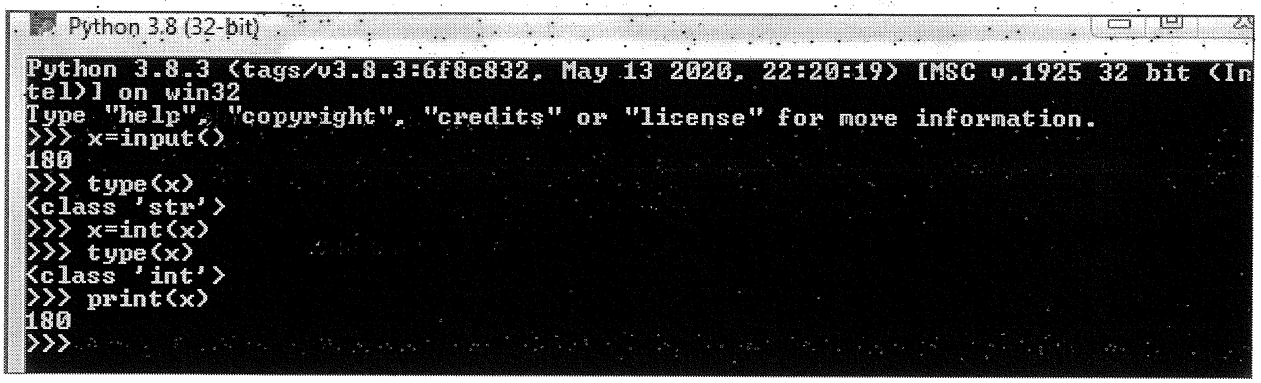
  <variable name> = input("<Prompting message>")

- Example:

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

age = int(input("How old are you? "))
print ("Your age is", age)
print ("You have", 65 - age, "years until retirement")
```

- Output:

```
How old are you? 53
Your age is 53
You have 12 years until retirement
```



➢ Math commands

- **Python has useful commands for performing calculations.**

Command name	Description
abs(*value*)	absolute value
ceil(*value*)	rounds up
cos(*value*)	cosine, in radians
floor(*value*)	rounds down
log(*value*)	logarithm, base *e*
log10(*value*)	logarithm, base 10
max(*value1*, *value2*)	larger of two values
min(*value1*, *value2*)	smaller of two values
round(*value*)	nearest whole number
sin(*value*)	sine, in radians
sqrt(*value*)	square root

Constant	Description
e	2.7182818...
pi	3.1415926...

➢ Programming Examples

- 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)
```

- Program to compute the logarithm base 10 of the signal-to-noise ratio.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
```

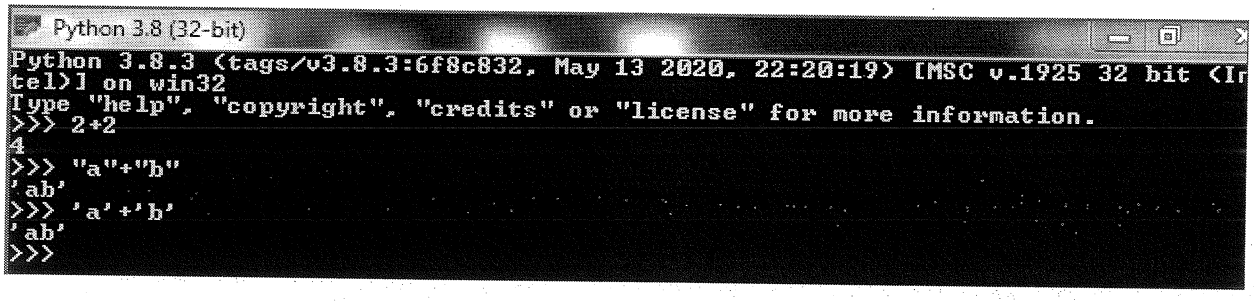
- Program to convert from degrees to radians, divide by 360 and multiply by 2π. (To use many of these commands, you must write the following at the top of your Python program: **from math import *)**

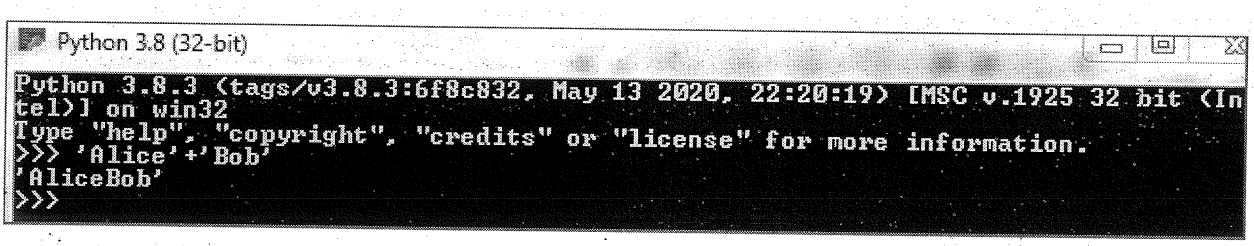
```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

➢ String Concatenation and Replication

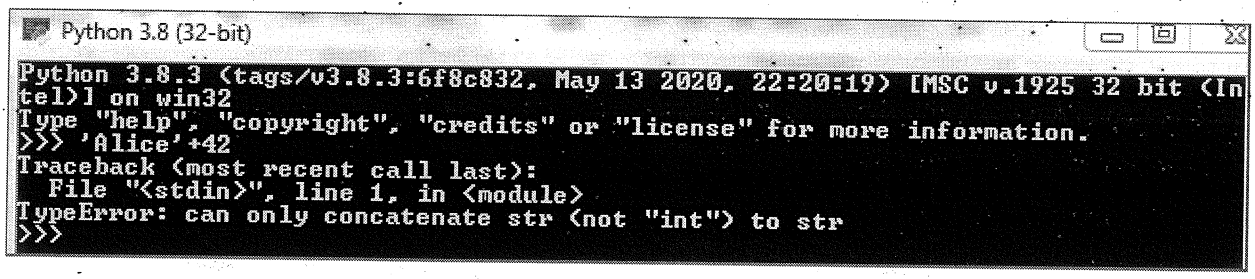
- The **+ operator** works with strings, but it is not addition in the mathematical sense. Instead it performs ***concatenation*****,** which means **joining the strings** by **linking them end to end**. For **example:**

```
>>> first = 10
>>> second = 15
>>> print(first + second)
25

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

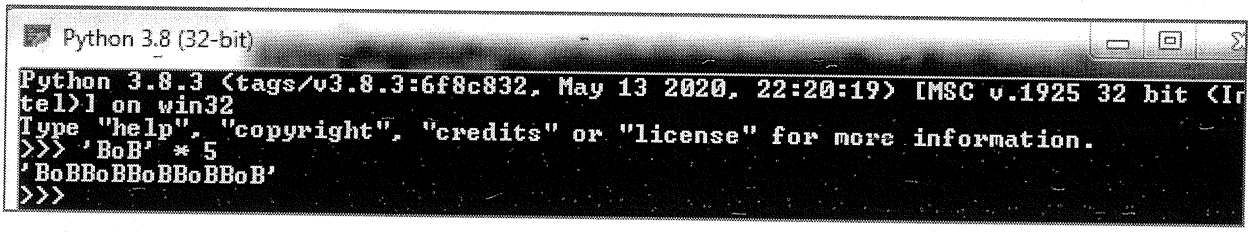




- The + operator on **a string and an integer value**, Python will not know how to handle this, and it will display an **error message** as below:



- Code will have to explicitly convert the integer to a string, because Python cannot do this automatically.
- The * **operator** is used for multiplication when it **operates on two integer or floating-point values.** But when the * **operator** is used on **one string value** and **one integer value,** it becomes the **string replication operator**. This is as shown below:



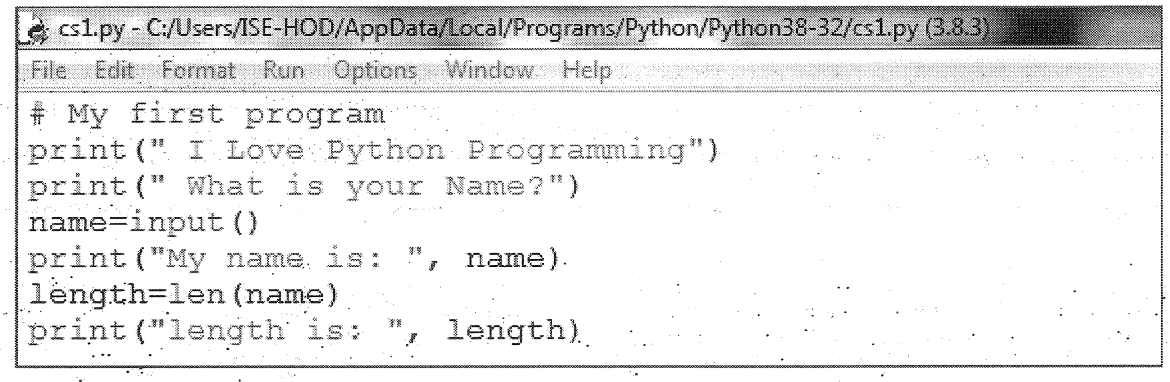
- The expression evaluates down to a single string value that repeats the original a number of times equal to the integer value.

➢ First Python Program

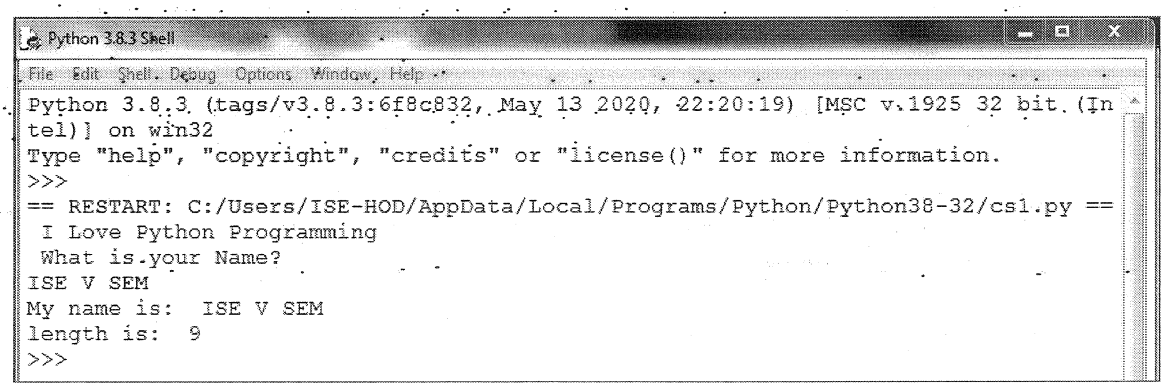
There are two ways of writing the python program.

1. Python interactive shell with >>> **prompt.**
2. **IDLE Text Editor:** The file editor lets you type in many instructions save the file, and run the program **(Select Run module or just press the F5 key.)**

Details in First Program with IDLE text editor:



Output: accepts name and computes **length of a string** using **len** function.

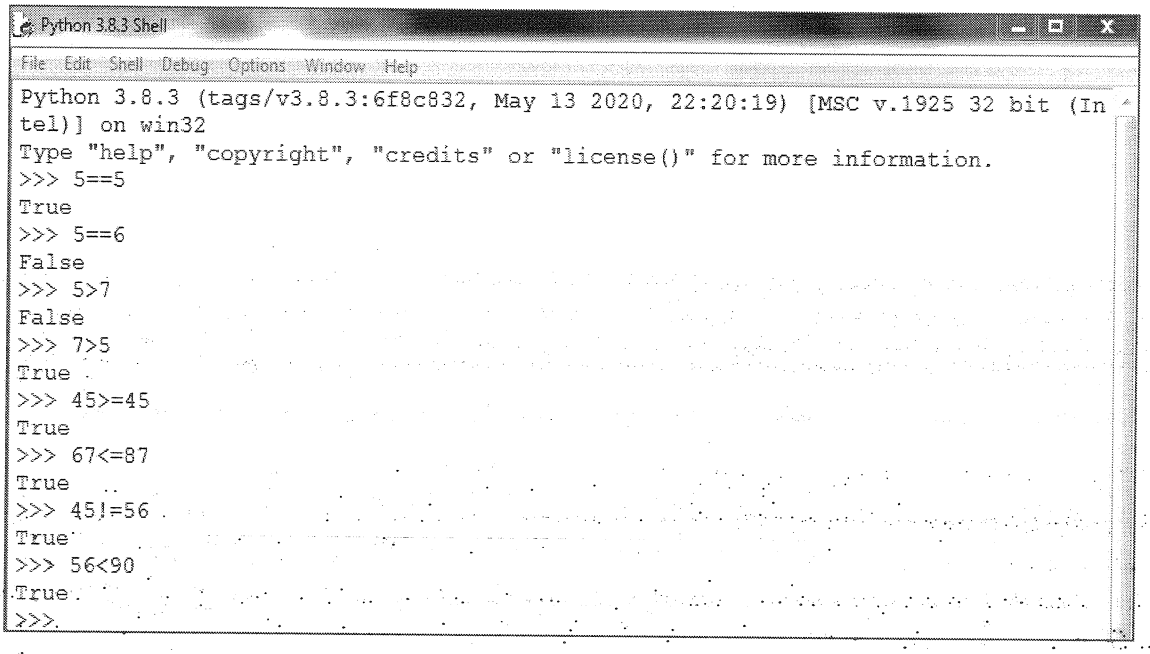


➢ Flow Control

Flow control statements can decide which Python instructions to execute under which conditions. To understand **yes or no**, let us explore Boolean values, comparison operators, and Boolean operators.

• Boolean data type

It has only two values: **True and False**. (Boolean is capitalized because the data type is named after mathematician George Boole.)



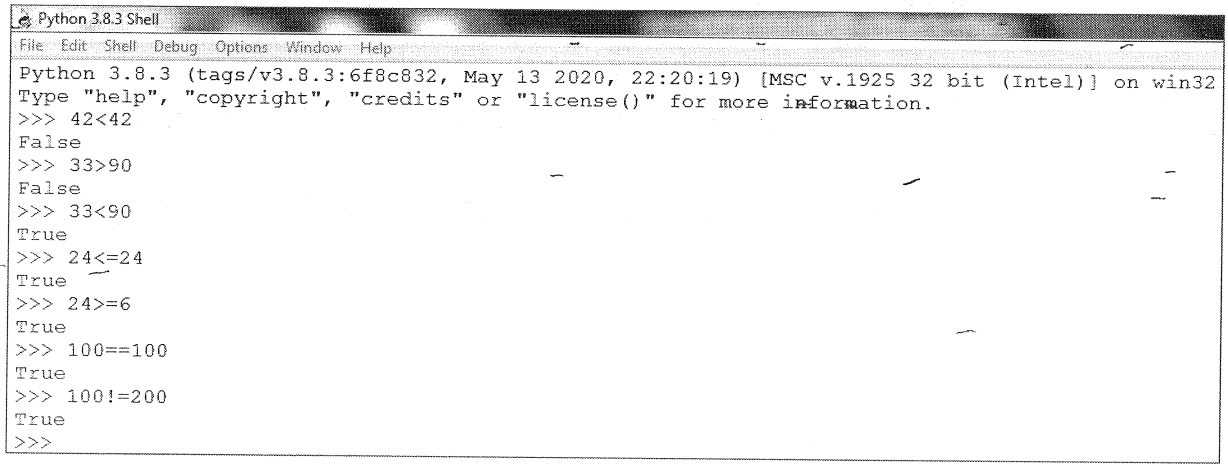
- **Comparison Operators**

Comparison operators compare two values and evaluate down to a single Boolean value as below:

Comparison Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Examples:



- **Binary Boolean Operators**
  - ***and*** **and** ***or*** operators always take **two Boolean values** (or expressions), so considered as **binary operators.**
  - The **and operator** evaluates an expression to **True if both Boolean values are True**; **otherwise,** it evaluates to **False.** A truth table shows every possible result of the **and** Boolean operator.

Expression	Evaluates to.
True and True	True
True and False	False
False and True	False
False and False	False

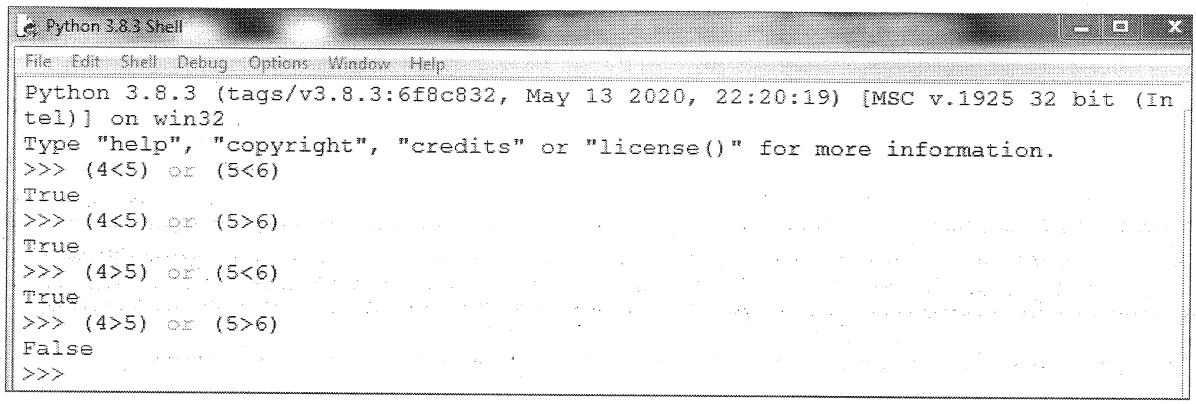
Examples:

```
Python 3.8.3 Shell
File Edit Shell Debug Options Window Help
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> (4<5) and (5<6)
True
>>> (4>5) and (5<6)
False
>>> (4<5) and (5>6)
False
>>> (4>5) and (5>6)
False
>>>
```

- The **or operator** evaluates an expression **to True if either of the two Boolean values is True**. If **both are False**, it evaluates to **False.** The possible outcome of the **or operator** is as follows:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

Examples:

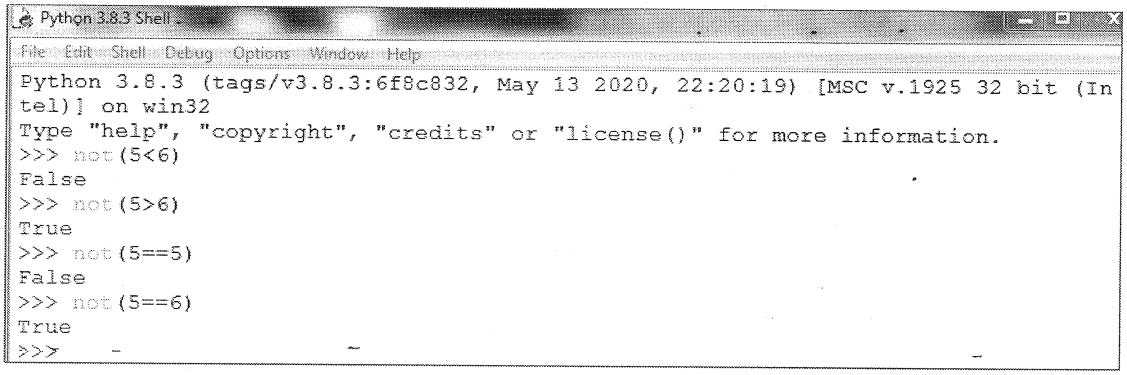


- **The not Operator**

The **not operator** operates on **only one Boolean value** (or **expression**). The not operator simply evaluates to the **opposite Boolean value** as below:

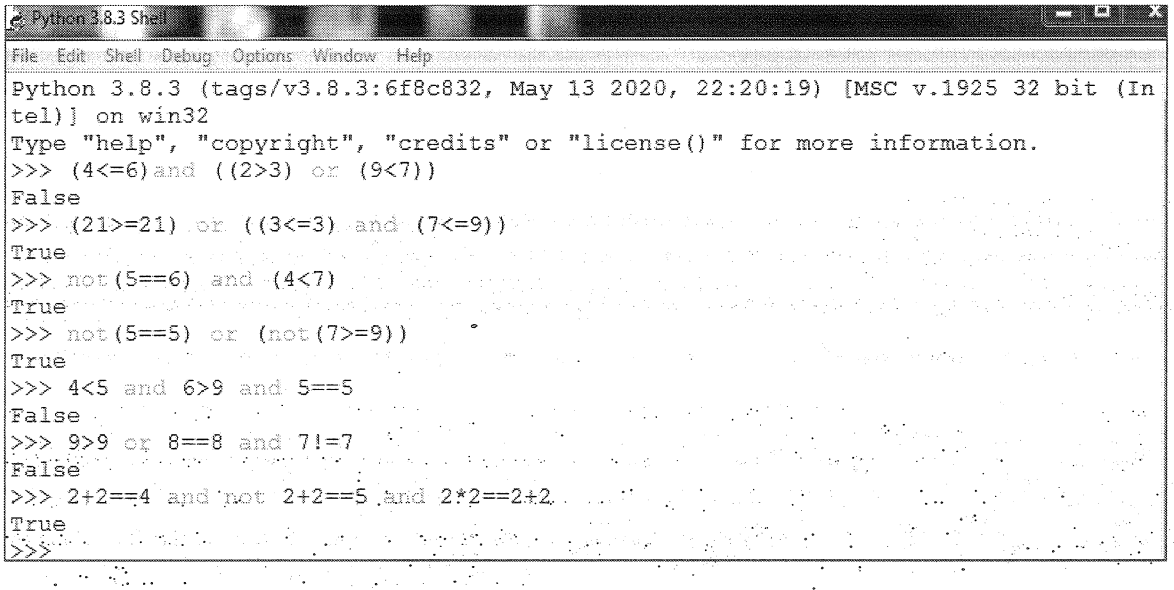
Expression	Evaluates to.
not True	False
not False	True

Examples:



- **Mixing Boolean and Comparison Operators**

The **comparison (Relational) operators** evaluate to Boolean values, they can be used in **expressions with the Boolean operators**. Recall that the ***and, or,*** **and** ***not*** operators are called **Boolean (logical)** operators because they always operate on the Boolean.



➢ Elements of Flow Control

Flow control statements often start with a part called the *condition*, and all are followed by a *block of code* called the clause.

- **Conditions**
  - The Boolean expressions all be considered conditions, which are same as expressions.
  - Conditions always evaluate down to a Boolean value, **True or False**.
  - A flow control statement decides what to do, based on whether its condition is **True or False**, and almost every **flow control statement uses a condition.**
- **Blocks of Code**

  Lines of Python code can be grouped together in blocks. When a block begins and ends from the indentation of the lines of code is done. There are three rules for blocks.

  - Blocks begin when the indentation increases.
  - Blocks can contain other blocks.
  - Blocks end when the indentation decreases to zero or to a containing block's indentation.
- **Program Execution**

  The program execution is a term for the current instruction being executed. Not all programs execute by simply going straight down. Jumping around the source code based on conditions, and you'll probably skip entire clauses.

Chapter 2

Flow Control Statements

Flow Control Statements are the actual decisions programs will make. There are many flow control statements in python.

✓ **if** statement

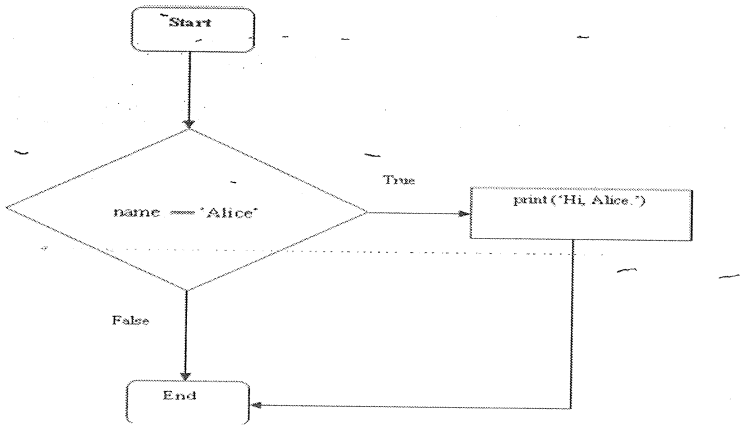
An **if** statement's clause (the block following the if statement) will execute if the statement's **condition is True**. The clause is skipped if the **condition is False.**

- **if** statement consists of the following:
  - **if** keyword
  - A **condition** (that is, an **expression** that evaluates to **True or False**)
  - A **colon**
  - Starting on the next line, **an indented block of code** (called the **if clause)**

Examples:

```
if (a==100):
    print("the value of a:", a)

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

if (name=='Alice'):
    print('Hi, Alice')
```

All flow control statements end with a colon and are followed by a new block of code. Figure below is a **flowchart.**



✓ else Statement

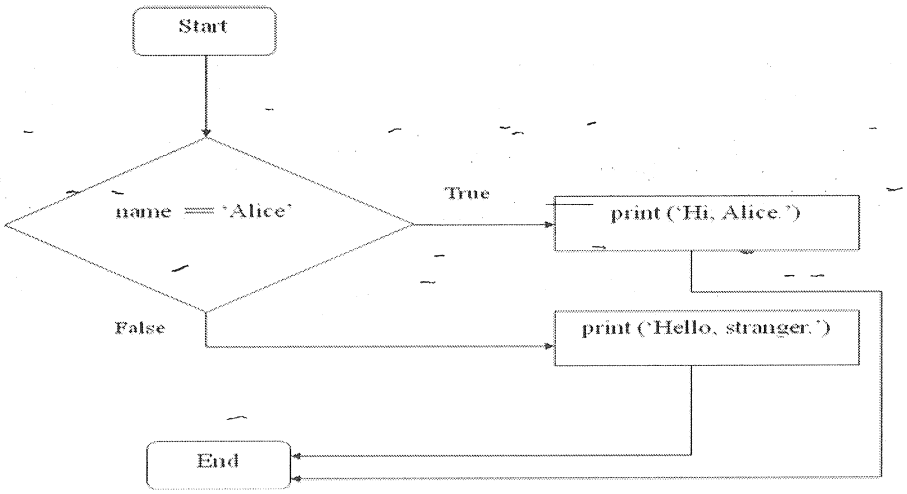
An if clause can **optionally** be followed by an **else statement.** The else clause is executed only when the **if statement's condition is False**. An else statement **doesn't have a condition. else** statement always consists of the following:

- **The else keyword**
- **A colon**
- **Starting on the next line, an indented block of code (called the else clause)**

Examples:

```
if (a==100):
   print("the value of a:",  a) #or use + in place of comma
else:
   print(" Not equal")

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

if (name=='Alice'):
      print('Hi, Alice')
else:
      print('Hello, stranger)
```

Figure below is a **flowchart:**



✓ elif statements

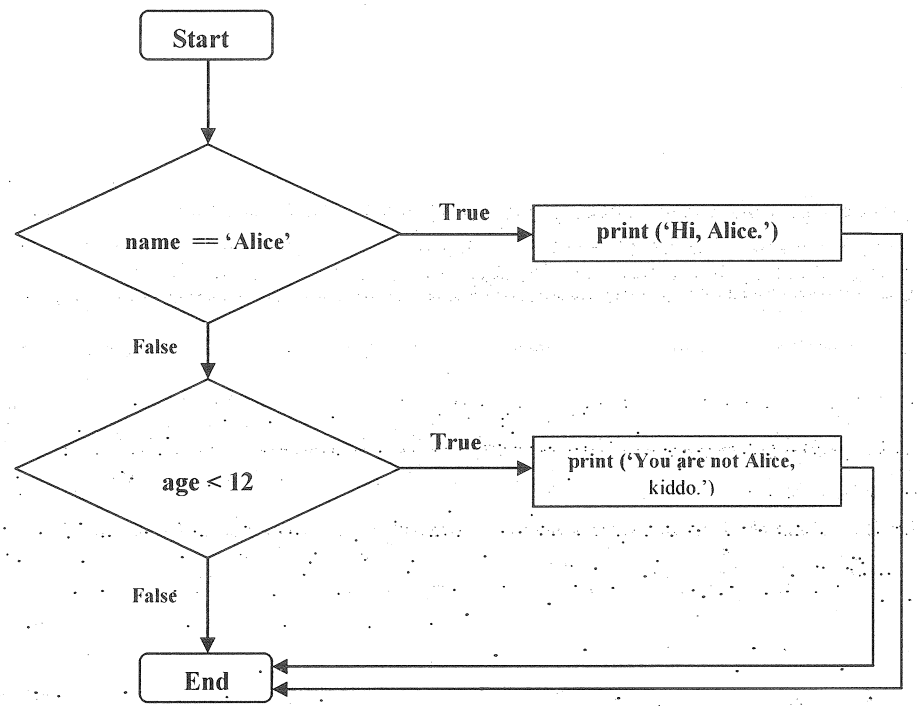
While only one of the **if or else** clauses will execute, when many of possible clauses to execute. The **elif** statement is an **"else if"** statement that always follows an **if** or another **elif** statement. It provides another condition that is checked only if any of the previous conditions were False. The order of the **elif** statements does matter. An **elif** statement always consists of the following:

- **The elif keyword**
- **A condition (that is, an expression that evaluates to True or False)**
- **A colon**
- **Starting on the next line, an indented block of code (called the elif clause)**

Examples:

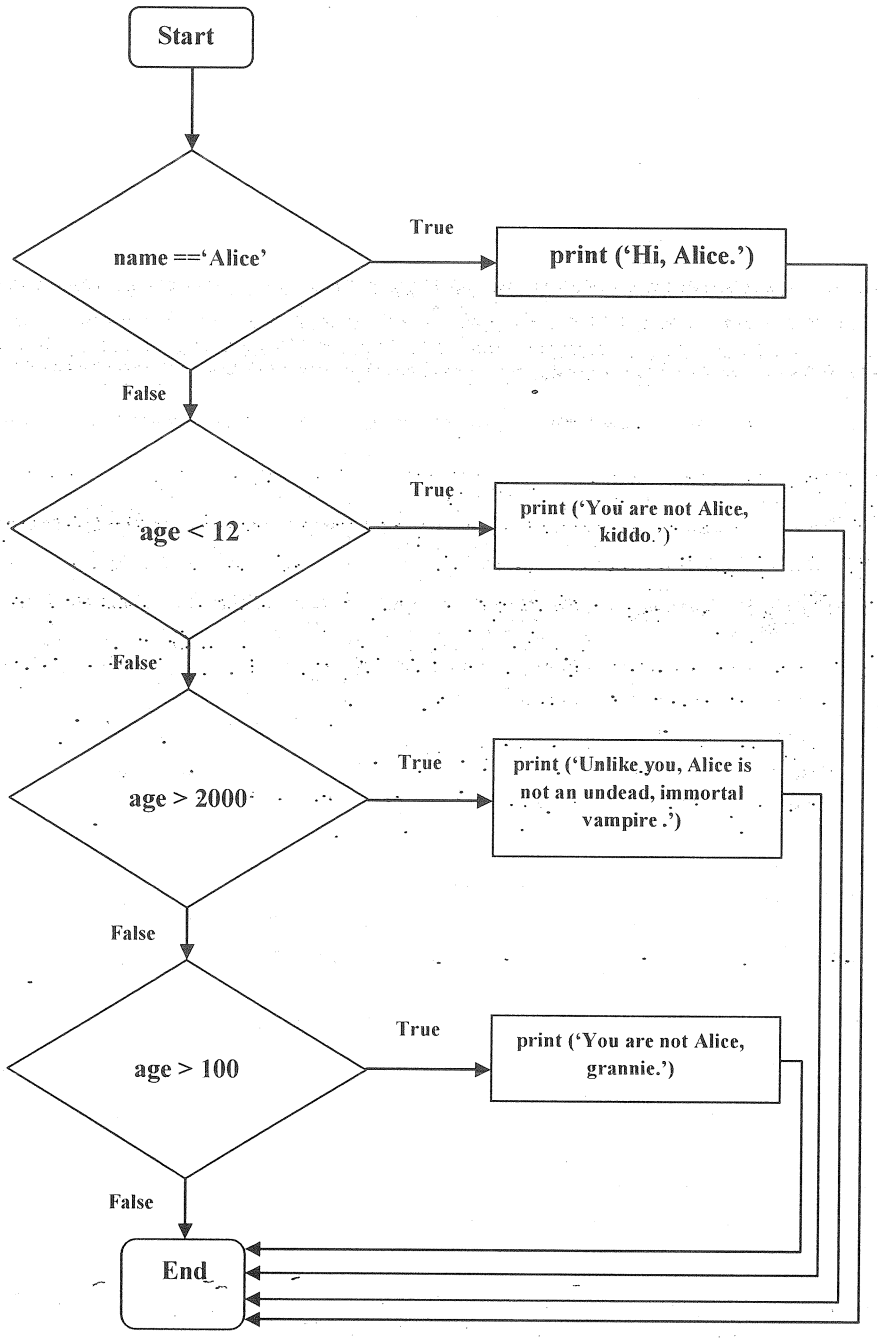
```
if x < y:
        print('x is less than y')
elif x > y:
        print('x is greater than y')
else:
        print('x and y are equal')

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

if (name=='Alice'):
        print('Hi, Alice')
elif  age-< 12
        print('you are not Alice, kiddo)
```

Below figure shows **flowchart**




```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Figure below shows the flowchart



If the conditions in every **if and elif statement are False**, then the **else clause** is executed.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

➢ while Loop Statements

The code in a **while clause** will be executed as long as the **while statement's condition is True.** A while statement always consists of the following:

- **while** keyword
- A condition (an expression that evaluates to **True or False**)
- A colon
- Starting on the next line, an indented block of code (called the **while clause**)

✓ While statement V/S if statement

The difference is in how they behave.

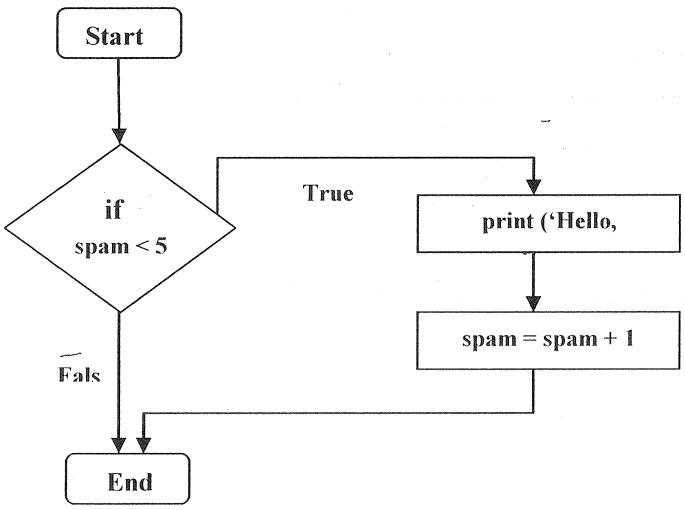
- At the end of an **if clause**, the program **execution continues** after the **if statement.** But at the end of a while clause, the program **execution jumps back** to the start of the **while statement.**
- The while clause is often called the **while loop or loop.** Consider an if statement and a while loop that use the same condition and take the same actions based on that condition.

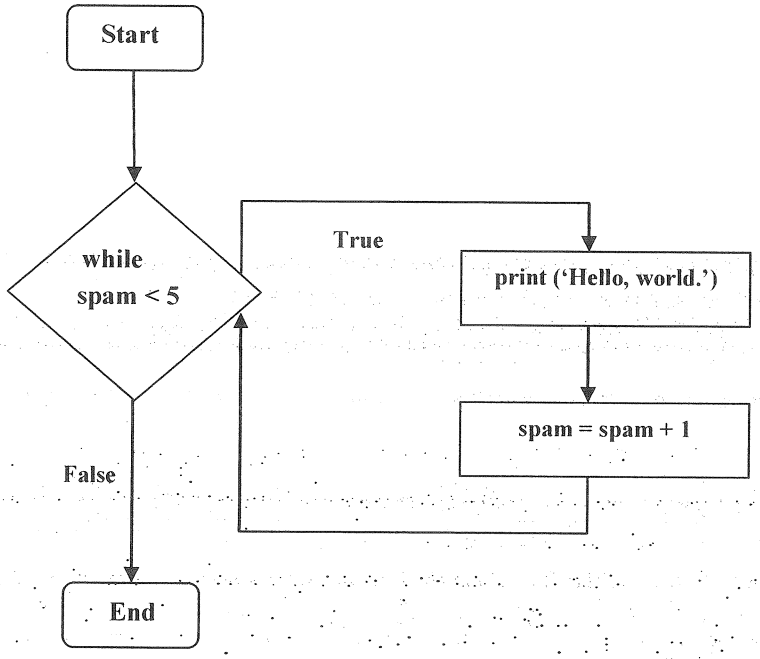
Here is the code with an if statement and while statement both.

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Flowcharts for these two pieces of code for **if and while** are as below:





In the while loop, the condition is always checked at the **start of each iteration**. If the condition is **True**, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be **False**, the **while clause is skipped**.

Example program:

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input()
print('Thank you!')
```

NOTE: While loops can be **infinite,** because the while statement's condition is **always True.** Press **ctrl-C** or select Shell's **Restart Shell** from the menu to terminate.

```
while True:
    print('Hello world!')
```

✓ **Break Statements**

Break statement makes the program **execution out of a while loop's clause**. If the execution reaches a **break statement**, it **immediately exits** the while loop's clause.

In code, a break statement simply contains the **break** keyword.

```
while True:
    print('Please type your name.')
    name = input()
    if name == 'your name':
        break
print('Thank you!')
```

```
while True:
    print('please type your name')
    name=input ()
    if name=='ise rnsit':
        break
print('Thank you')
```

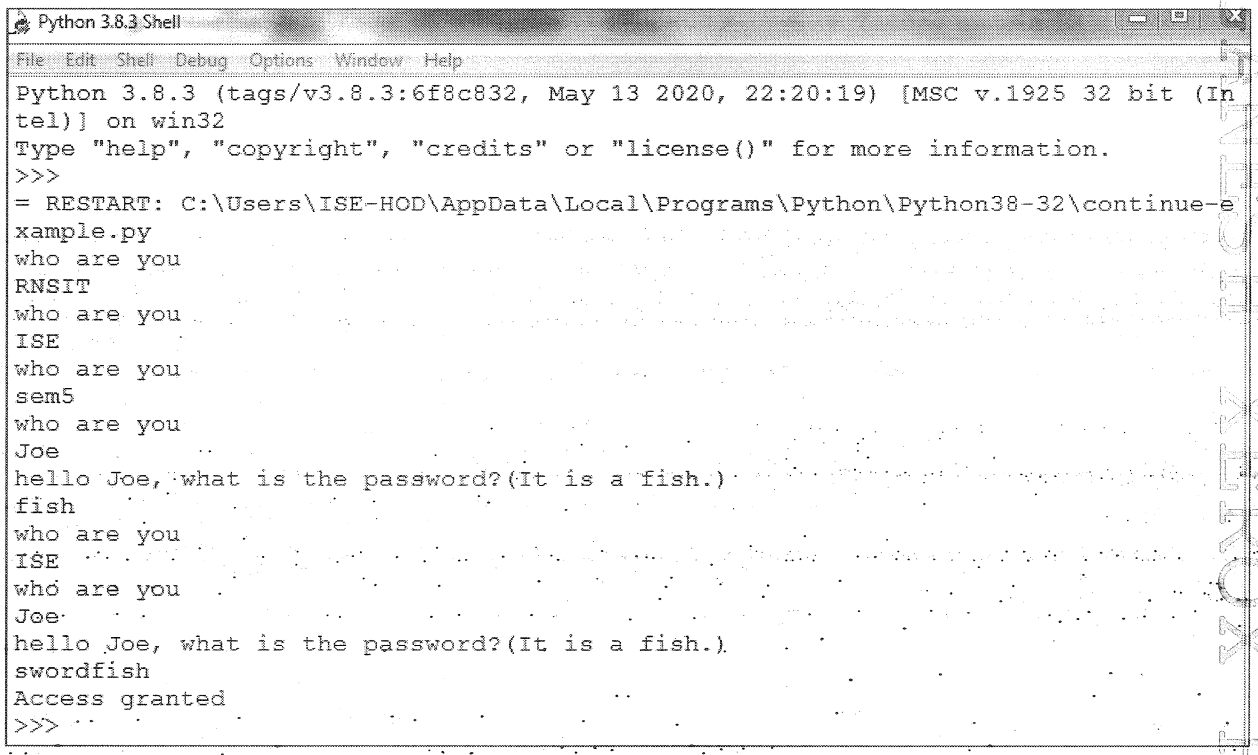
✓ **Continue Statement**

Continue statements are used **inside loops**. When the program **execution** reaches a **continue statement**, the program execution **immediately jumps back to the start of the loop** and reevaluates the loop's condition.

Example:

```
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted.')
```

```
while True:
    print('who are you')
    name=input()
    if name != 'Joe':
        continue
    print('hello Joe, what is the password?(It is a fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted')
```



- If the user enters any name besides Joe, the continue statement causes the program execution to jump back to the start of the loop.
- When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value **True.**
- Once make it pass that if statement, the user is asked for a password.
- If the password entered is **swordfish**, then the **break statement** is run, and the execution **jumps out of the while loop** to print Access granted.
- Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop. (See Figure 2-13 in Text Book for this program's flowchart).

➢ for Loops and the range() Function

If a block of code has to execute only a **certain number of times**, it can done with a **for loop statement and range () function.**

A **for** statement looks something like **for i in range(5):** and always includes the following:

- **for** keyword
- A **variable name**
- **in** keyword
- **A call to the range() method with up to three integers passed to it**

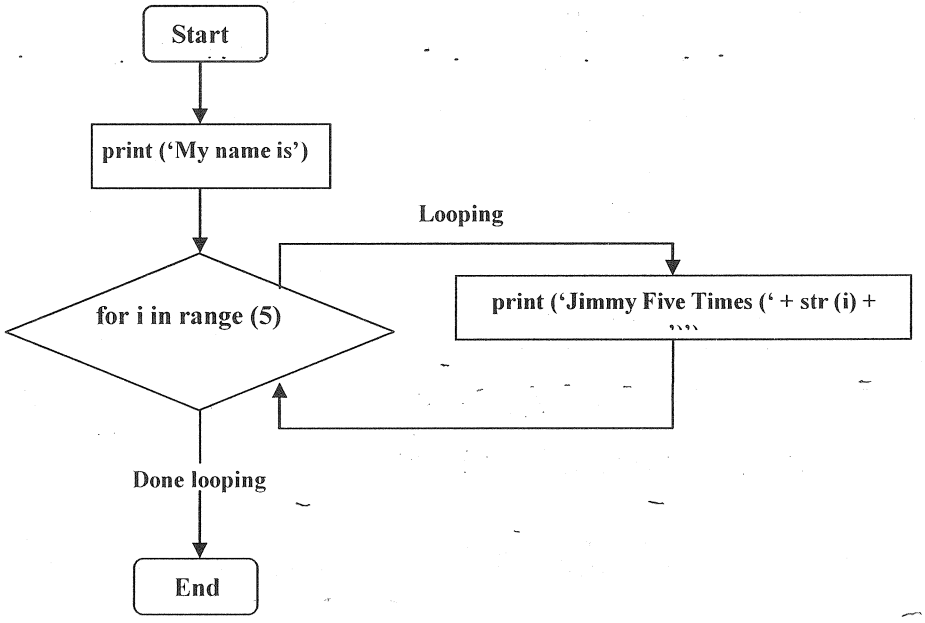
- **A colon**
- **Starting on the next line, an indented block of code (called the for clause)**

Example:

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

- The code in the **for loop's clause** will run **five times**
- range (5) results in **five iterations** through the clause, with i being set to 0, then 1, then 2, then 3, and then 4.
- The variable i will go **up to**, but will **not include,** the **integer passed** to range() (i.e. 0 to n-1). So it is from 0 to 4.
- Below shows the **output and flowchart.**

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```



✓ **The Starting, Stopping, and Stepping Arguments to range ()**

The range () function can be called with multiple arguments separated by a comma to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

The first argument indicates where **for loop's variable starts**, and the **second argument will be up to.** So the output will be

```
12
13
14
15
```

The range() function can also be called with **three arguments**. The **first two** arguments will be the **start and stop** values, and the **third** will be the **step argument.** The step is the amount that the variable is incremented with iterations.

```
for i in range(0, 10, 2):
    print(i)
```

So calling range(0, 10, 2) will count from zero to eight by intervals of two. Here the output will be

Even, we can use a **negative number** for the step argument to make the for loop **countdown** instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

Above code should print from five down to zero **5 4 3 2 1 0**

➢ Importing Modules

All Python programs can call a **basic set of functions** called **built-in-functions**, including the print(), input(), and len() functions. Python also contains a set of modules called the standard library. Each **module is a Python program** that contains a **related group of functions** that can be embedded in your programs.

For example, the **math module** has mathematics- related functions, the **random module** has random number–related functions, and so on. Before the use of functions in a module, module must be imported with an import statement.

In code, an **import statement** consists of the following:

- **import** keyword

- **name** of the module
- **Optionally, more module names, as long as they are separated by commas**

Example:

Use of **random.ranint()** function.

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

The **random.randint()** function call evaluates to a random integer value between the two integers that are passed, since **randint()** is in the random module. Random integers between 1 and 10 are printed as an output.

Here is an example of an import statement that imports **four** different modules:

```
import random, sys, os, math
```

From import Statements, an alternative form of the import statement is

from random import *.

➢ Ending a Program Early with sys.exit()

This deals with how to **terminate** the program. This always happens if the program execution reaches the bottom of the instructions. However, program can be **terminated**, or **exited**, by calling the **sys.exit()** function.

Since this function is in the sys module, **import sys**.

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

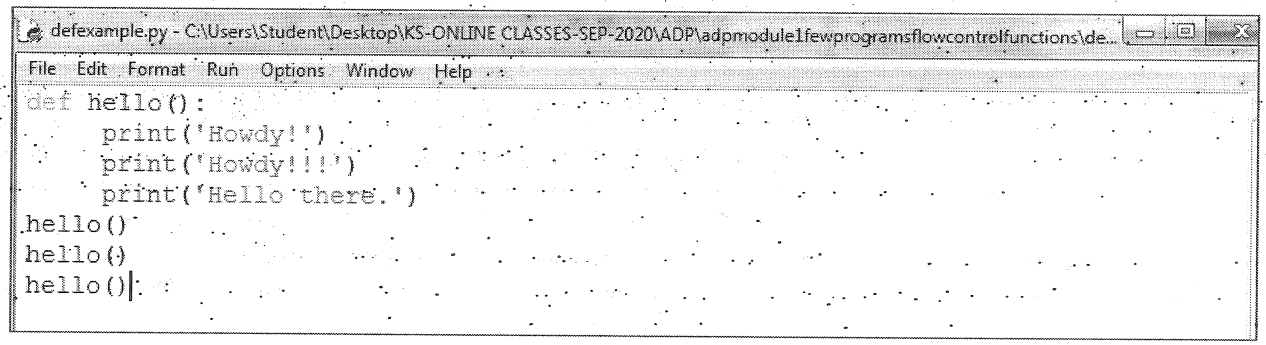
This program has an infinite loop with **no break statement inside**. The only way this program will end is if the user enters **exit,** causing **sys.exit() to be called**. When response is equal to exit, the program ends.

Chapter 3

Functions

Python provides several built- in functions like len( ), str( ) etc. User defined functions can also be created. Functions always help to avoid duplicating code. A major purpose of functions is to group code that gets executed **multiple times.**

Example:



- The first line is a **def** statement, which defines a function named **hello()**
- The **code in the block** that follows the **def** statement is the **body of the function**. This code is **executed** when the **function is called**, not when the function is first defined.

➢ Function call

A function call is just the **function's name followed by parentheses** can be with or without some number of **arguments** in between the parentheses.

The **hello()** lines after the function are **function calls in the above code.**

When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there.

When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\defexample.py
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

➢ def Statements with Parameters

User defined functions can be defined which accept arguments.

```
defexample.py - C:/Users/Student/Desktop/defexample.py (3.7.3)
File Edit Format Run Options Window Help
def hello(name):
    print('Hello'+name)

hello('Arjun')
hello('Priya')
hello('Yuktha')
```

The output:

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============= RESTART: C:/Users/Student/Desktop/defexample.py =============
Hello Arjun
Hello Priya
Hello Yuktha
>>>
```

➢ Return Values and return Statements

When len() function is called with an argument such as 'Hello', the function call evaluates to the integer **value 5**, which is the length of the string you passed it.

In general, the value that a **function call evaluates** to is called the **return value** of the function. When creating a function using the **def** statement, you can specify what the **return value** should be with a **return statement.**

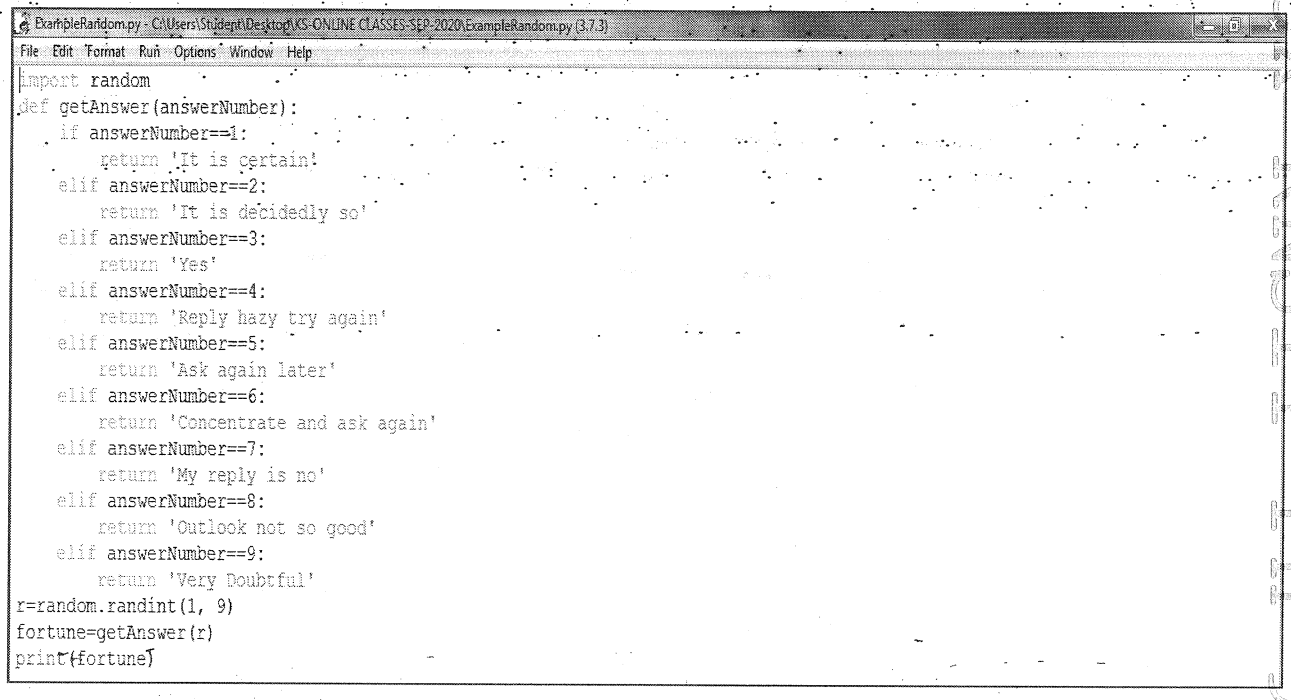
A **return** statement consists of the following:

- **return keyword**
- **value or expression** that the function should return

When an expression is used with a return statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument.

Note: since you can pass return values as an argument to another function call, you could shorten these three lines to this **single equivalent line.**

Print(getAnswer(random.randInt(1,9))

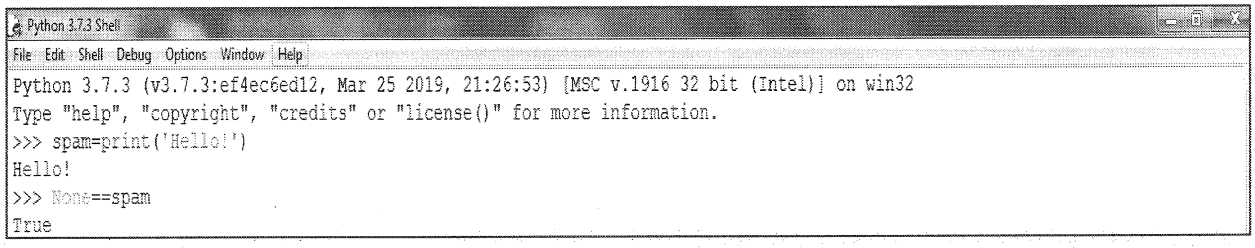


➢ The None Value

- In Python, there is a value called **None**, which represents the **absence of a value**.
- **None** is the only value of the **None Type** data type.
- None must be typed with a **capital N**.
- This **value-without-a-value** can be helpful when you need to store something that will not be confused for a real value in a variable.
- One place where **None is used** is as the **return value of print().**

The print () function displays text on the screen, but no return value.

But since all function calls need to evaluate to a return value, print() returns None.



Python adds return None to the end of any function definition with no return statement.

This is similar to how a while or for loop implicitly ends with a continue statement.

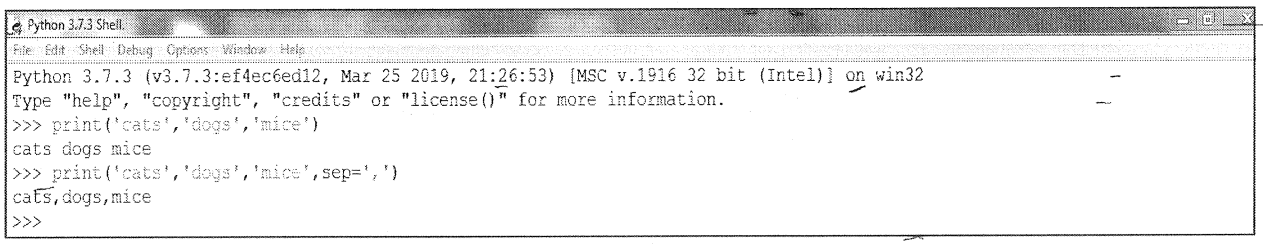
Function return statement without a value will be None always.

- **Keyword arguments**: Key word arguments are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters.

For example, the print () function has the optional parameters end and **Sep** to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

The two strings appear on separate lines because the print () function automatically adds a newline character to the end of the string it is passed. However, you can set the **end** keyword argument to change this to a different string.

To disable the newline that gets added to the end of every print() function call. Similarly, when you pass multiple string values to print(), the function will automatically separate them with a single space. Replace the default separating string by passing the **sep** keyword argument.



You can add keyword arguments to the functions you write as well

➢ Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope.

A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable.

A variable must be one or the other; it cannot be both local and global.

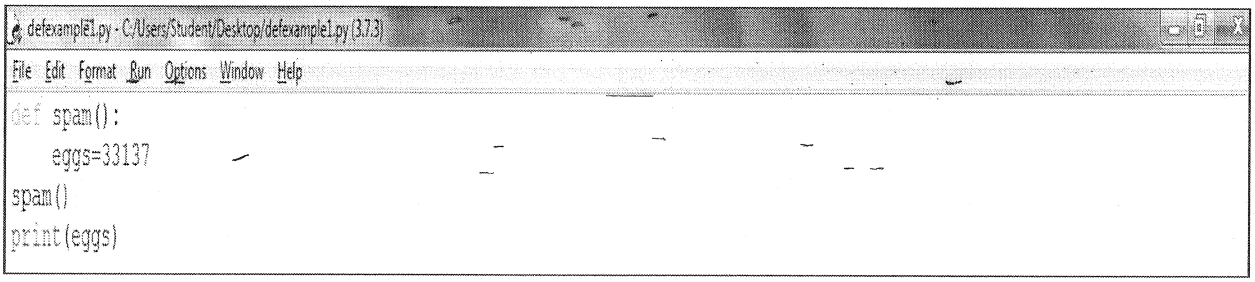
A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- •Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in dif ferent scopes.
- While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:



Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs. This makes sense if you think about it; when the program

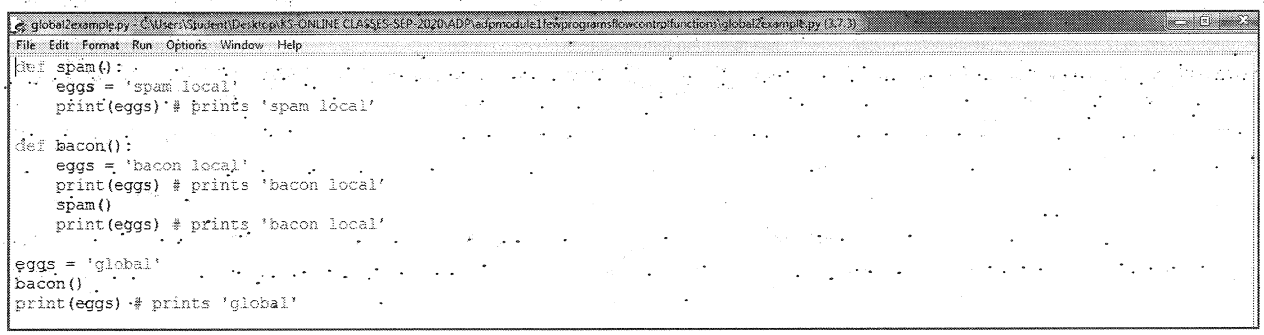
execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

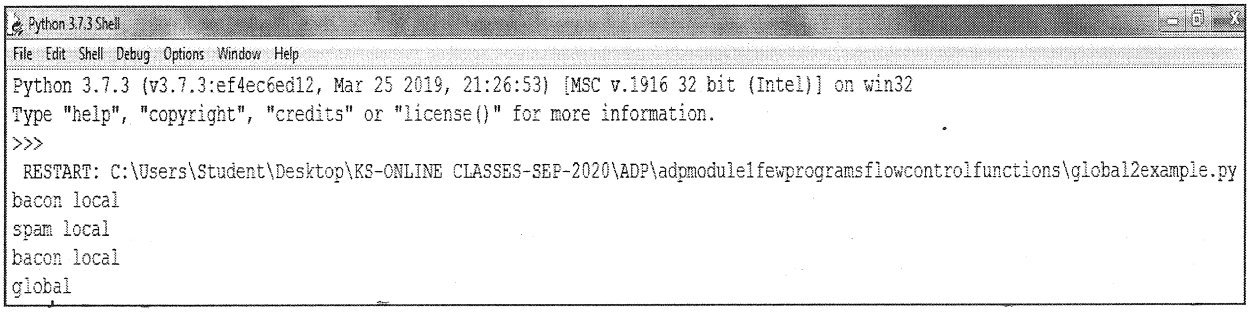
A new local scope is created whenever a function is called, including when a function is called from another function.

Global Variables Can Be Read from a Local Scope

Local and Global Variables with the Same Name, it's perfectly legal to do so in Python. To see what happens, type the following code into the file editor and save it as sameName.py:



Output:



There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

- A variable named eggs that exists in a local scope when spam() is called.
- A variable named eggs that exists in a local scope when bacon() is called.
- A variable named eggs that exist in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

We can modify a global variable from within a function, use the global statement. If you have a line such as global eggs at the top of a function, it tells Python, "In this function, eggs refers to the global variable, so don't create a local variable with this name."

```
def spam():
    global eggs
    eggs='spam'
eggs='global'
spam()
print(eggs)
```

Output will be spam.

```
def spam():
    global eggs
    eggs='spam' #This is global
def bacon():
    eggs='bacon'#This is a local
def ham():
    print(eggs) #this is global
eggs=42
spam()
print(eggs)
```

Output will be spam.

➢ Exception Handling

An error, or exception, in your Python program crashes the program. Program execution will be terminated abruptly. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a "divide-by- zero" error. Open a new file editor window and enter the following code, saving it as zeroDivide.py:

```
ExceptionExample.py - C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\ExceptionExample.py (3.7.3)
File Edit Format Run Options Window Help
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what hap- pens.

This is the output you get when you run the previous code:

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\ExceptionE
py
21.0
3.5
Traceback (most recent call last):
  File "C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\ExceptionExa
", line 6, in <module>
    print(spam(0))
  File "C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\ExceptionExa
", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

Errors can be handled with try and except statements.

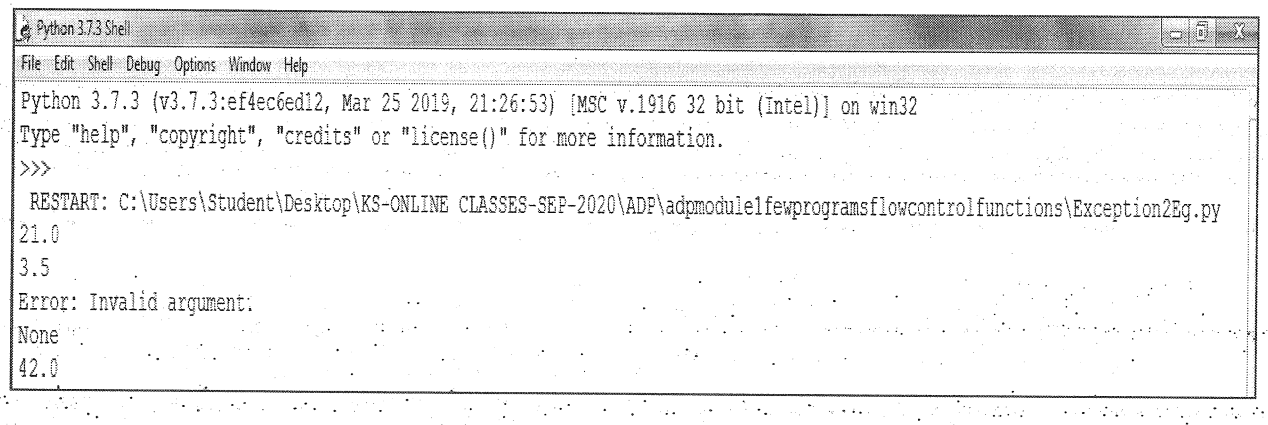
The code that could potentially have an error is put in a try clause.

The program execution moves to the start of a following except clause if an error happens.

You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
Exception2Eg.py - C:\Users\Student\Desktop\KS-ONLINE CLASSES-SEP-2020\ADP\adpmodule1fewprogramsflowcontrolfunctions\Exception2Eg.py (3.7.3)
File Edit Format Run Options Window Help
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

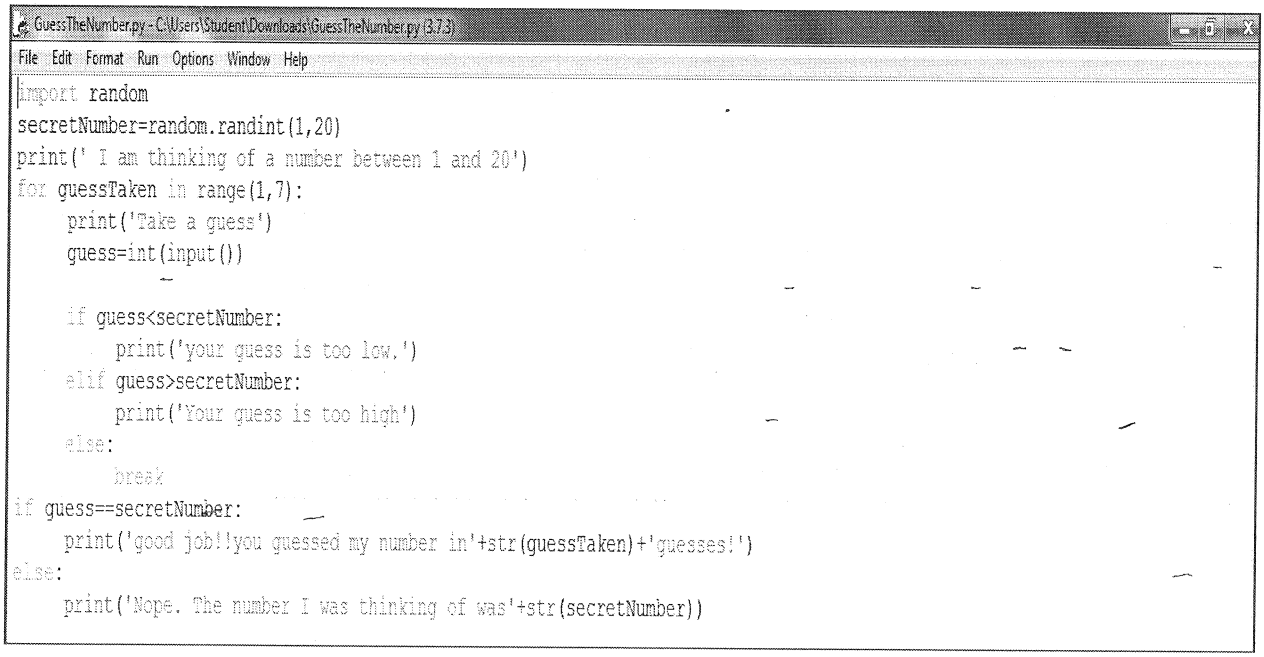
When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous pro- gram is as follows:



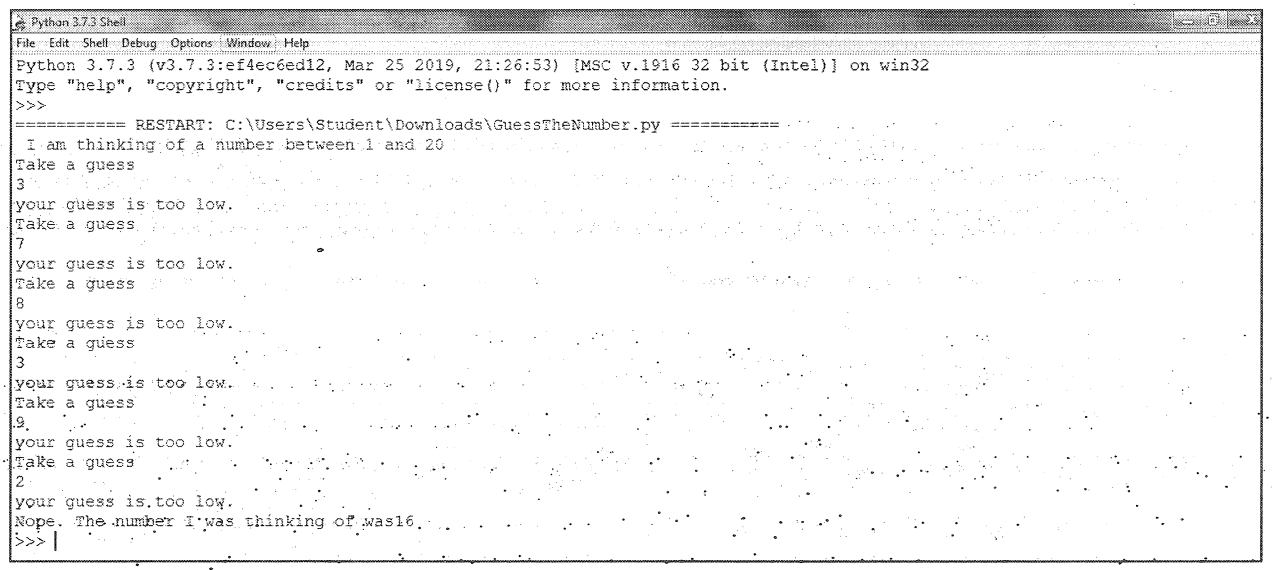
Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

❖ A Short Program: Guess the Number

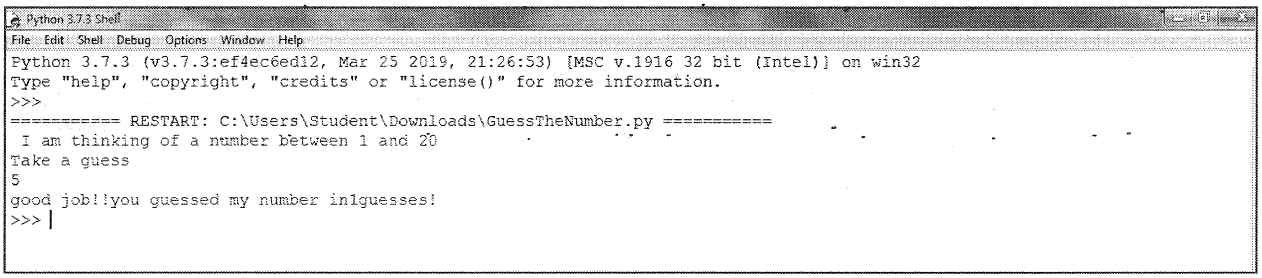
In this section, a simple "guess the number" game is given as an example to use all the basic concepts understood so far.



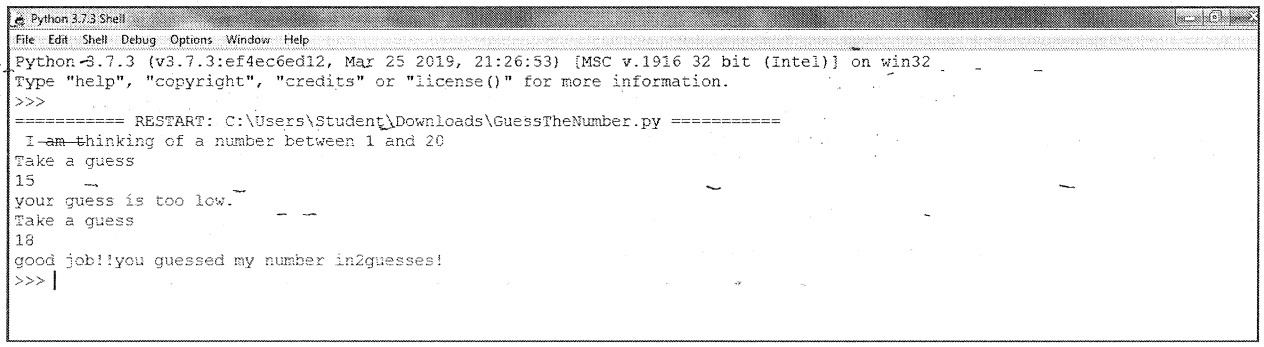
Output-1:



Output-2:



Output-3:



MODULE-2

Contents:

Lists: The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, 08 Example Program: Magic 8 Ball with a List, List-like Types: Strings and Tuples, References.

Dictionaries and Structuring Data: The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things.

Manipulating Strings: Working with Strings, Useful String Methods, Project: Password Locker, Project: Adding Bullets to Wiki Markup.

1: Chapters 4 – 6 RBT: L1, L2, L3

Textbook 1: Chapters 1 – 3 RBT: L1, L2

❖ LISTS

The list Data Type

A list is a sequence of values. A list is a value that contains multiple values in an ordered sequence. The elements of a list don't have to be the same type. A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and])

Example: A list that contains no elements is called an empty list; you can create one with empty brackets, [].

First list is a list of four integers [10, 20, 30, 40]

The second is a list of three strings [‘Crunchy frog', 'ram bladder']

A list within another list is nested. [“hi”, 2.0, 5, [10, 20]]

We can assign list values to variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
>>> empty = []
```

```
>>> print(cheeses, numbers, empty)
'Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Getting Individual Values in a List with Indexes

Consider List spam= ['cat', 'bat', 'rat', 'elephant']

Here, spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on.

The integer inside the square brackets spam that follows the list is called an **index.**

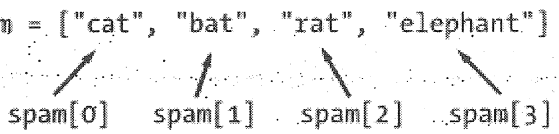


Figure 4-1: A list value stored in the variable spam, showing which value each index refers to

Indexes can be only integer values, not floats. The following example will cause a TypeError error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

Negative Indexes

Negative integers can be used for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Getting Sub lists with Slices

Just as an index can get a single value from a list, a slice can get several values from a list, in the form of a new list

A slice is typed between square brackets, like an index, but it has two integers separated by a colon

- spam[2] is a list with an index (one integer)
- spam[1:4] is a list with a slice (two integers)

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

In slice, we can leave out the first index, second index or both the index. These are the examples

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
```

```
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len() : The len() function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

List Methods

The list data type has some more methods. Here are all of the methods of list objects:

list.**append**(*x*)

Add an item to the end of the list. Equivalent to a[len(a):] = [x].

list.**extend**(*iterable*)

Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

list.**insert**(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.**remove**(*x*)

Remove the first item from the list whose value is equal to *x*. It raises a ValueError if there is no such item.

list.**pop**([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.**clear**()

Remove all items from the list. Equivalent to del a[:].

list.**index**(*x*[, *start*[, *end*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a ValueError if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

list.**count**(*x*)

Return the number of times *x* appears in the list.

list.**sort**(*key=None*, *reverse=False*)

Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

list.**reverse**()

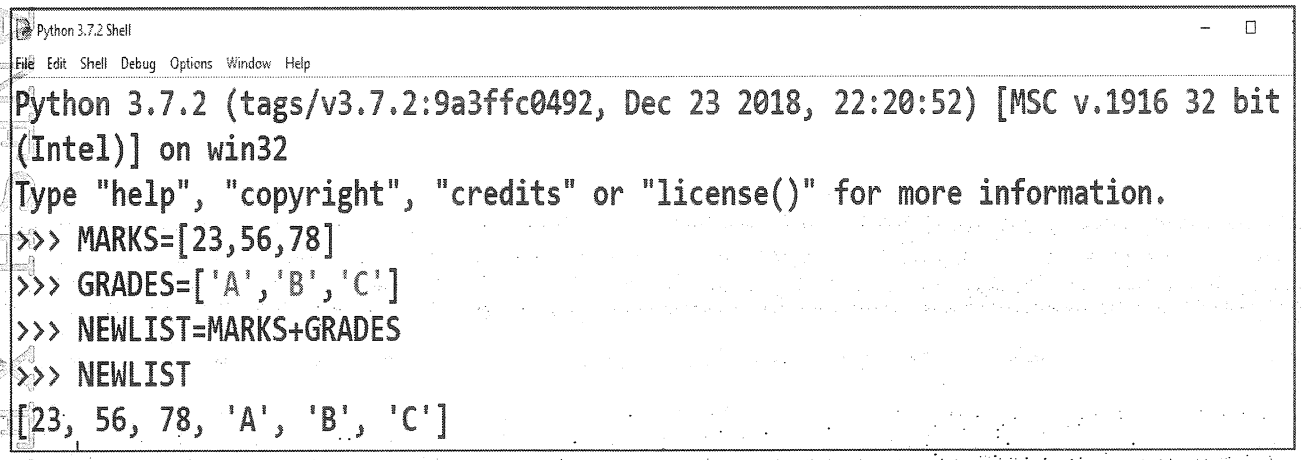
Reverse the elements of the list in place.

list.**copy**()

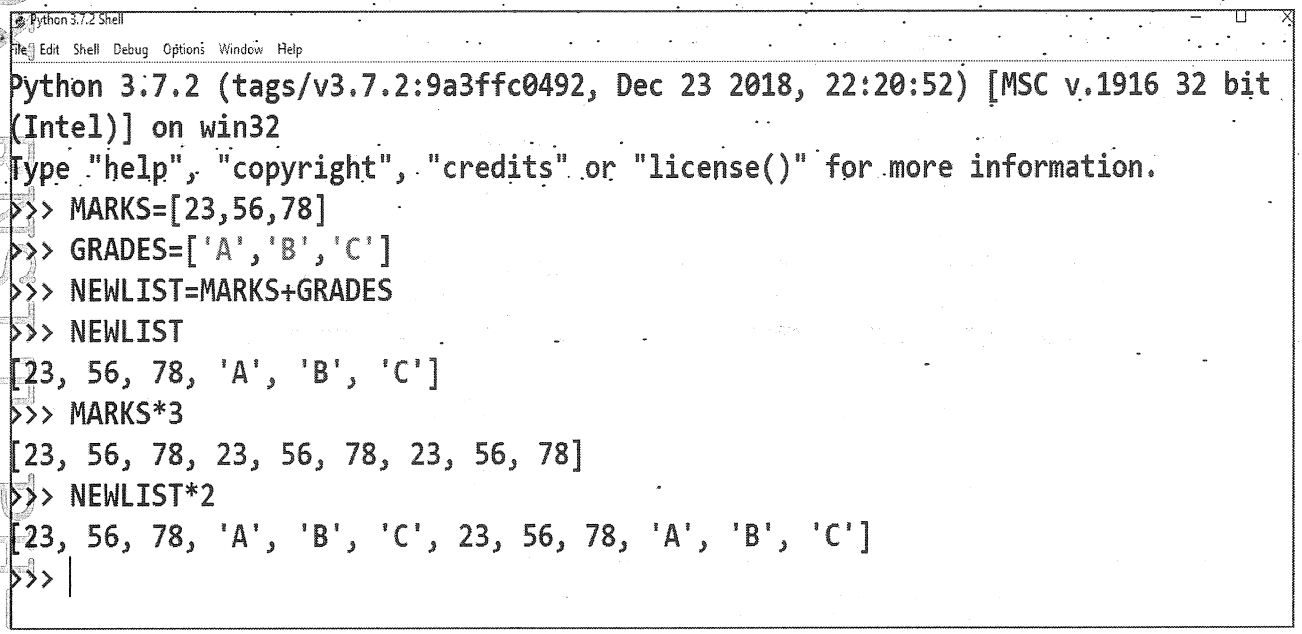
Return a shallow copy of the list. Equivalent to a[:].

List Concatenation and List Replication

The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The * operator can also be used with a list and an integer value to replicate the list.



List replication:



Removing Values from Lists with del Statements

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index.

Working with Lists

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. This new version uses a single list and can store any number of cats that the user types in

cat.py - C:/Users/KUSUMA/Desktop/cat.py (3.7.2)
File Edit Format Run Options Window Help

```
print('enter the name of cat 1:')
catName1=input()
print('enter the name of cat 2:')
catName2=input()
print('enter the name of cat 3:')
catName3=input()
print('enter the name of cat 4:')
catName4=input()
print('enter the name of cat 5:')
catName5=input()
print('enter the name of cat 6:')
catName6=input()
print('the cat names are:')
print(catName1+''+catName2+''+catName3+''+catName4+''+catName5+''+catName6)
```

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]  # list concatenation
print('The cat names are:')
for name in catNames:
    print('  ' + name)
```

Using for Loops with Lists

A for loop repeats the code block once for each value in a list or list-like value.

Syntax:

```
for i in range(4):

        print(i)
```

This is because the return value from range (4) is a list-like value that Python considers similar to [0, 1, 2, 3].

```
for  i in [0,1 ,2,3]:
        Print(i)
```

Above loop through its clause with the variable i set to a successive value in the [0, 1, 2, 3] list in each iteration. A common Python technique is to use range(len(someList)) with a for loop to iterate over the indexes of a list

The in and not in Operators

You can determine whether a value is or isn't in a list with the in and not in operators. Like other operators, in and not in are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value.

Example:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam True
```

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code

>>> cat = ['fat', 'black', 'loud'] >>> size, color, disposition = cat

The number of variables and the length of the list must be exactly equal, or Python will give you a ValueError.

Removing Values from Lists

With remove() The remove() method is passed the value to be removed from the list it is called on.

>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam.remove('bat')

>>> spam ['cat', 'rat', 'elephant']

Attempting to delete a value that does not exist in the list will result in a ValueError error.

- If the value appears multiple times in the list, only the first instance of the value will be removed.
- The del statement is good to use when you know the index of the value you want to remove from the list. The remove() method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the sort() method.

>>> spam = [2, 5, 3.14, 1, -7]

>>> spam.sort()

>>> spam [-7, 1, 2, 3.14, 5]

>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']

>>> spam.sort()

>>> spam ['ants', 'badgers', 'cats', 'dogs', 'elephants']

Pass True for the reverse keyword argument to have sort () sort the values in reverse order

>>> spam.sort(reverse=True)

```
>>> spam ['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

The sort() method sorts the list in place; don't try to capture the return value by writing code like

```
>>>spam = spam.sort().
```

We cannot sort lists that have both number values and string values in them, since Python doesn't know how to compare these values.

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last): File "", line 1, in spam.sort()
TypeError: unorderable types: str() < int()
```

To sort the values in regular alphabetical order, pass str. lower for the key keyword argument in the sort() method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam ['a', 'A', 'z', 'Z']
```

Example Program: Magic 8 Ball with a List

```
import random
messages = ['It is certain', 'It is decidedly so', 'Yes definitely', 'Reply hazy try again', 'Ask again later',
'Concentrate and ask again', 'My reply is no', 'Outlook not so good', 'Very doubtful']
print(messages[random.randint(0, len(messages) - 1)]).
```

The expression you use as the index into messages: random .randint(0, len(messages) - 1). This produces a random number to use for the index, regardless of the size of messages.

List-like Types: Strings and Tuples

Strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Indexing; slicing; and using them with for loops, with len(), and with the in and not in operators can be used in strings also.

```
>>> name = 'Zophie'
>>> name[0] 'Z'
>>> name[-2] 'i'
 >>> name[0:4] 'Zoph'
>>> 'Zo' in name True
>>> 'z' in name False
 >>> 'p' not in name False
 >>> for i in name:
                print('* * * ' + i + ' * * *')
```

Output :

```
* * * Z * * *
 * * * o * * *
* * * p * * *
 * * * h * * *
 * * * i * * *
 * * * e * * *
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a mutable data type: It can have values added, removed, or changed. However, a string is immutable: It cannot be changed.

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
(most recent call last): File "", line 1, in name[7] = 'the'
TypeError: 'str' object does not support item assignment.
```

The proper way to "mutate" a string is to use slicing and concatenation to build a new string by copying from parts of the old string.

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
```

```
>>> name 'Zophie a cat' Lists 95
>>> newName 'Zophie the cat'
```

The proper way to "mutate" a string is to use slicing and concatenation to build a new string by copying from parts of the old string.

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name 'Zophie a cat' Lists 95
>>> newName 'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable. Although a list value is mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs [4, 5, 6]
```

The Tuple Data Type

The tuple data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and].

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0] 'hello'
>>> eggs[1:3] (42, 0.5)
>>> len(eggs)
3
```

Converting Types with the list() and tuple()

Functions Just like how str(42) will return '42', the string representation of the integer 42, the functions list() and tuple() will return list and tuple versions of the values passed to them.

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

References

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

Spam and cheese are different variables that store different values when you assign a list to a variable; you are actually assigning a list *reference* to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list.

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
```

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value. Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries.

For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself. Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

When a function is called, the values of the arguments are copied to the parameter variables. This means a copy of the reference is used for the parameter.

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when eggs() is called, a return value is not used to assign a new value to spam. Instead, it modifies the list in place, directly. Output will be [1,2,3,'Hello'] Even though spam and someParameter contain separate references, they both refer to the same list.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value.

For this, Python provides a module named copy that provides both the copy() and deepcopy() functions. The first of these, copy.copy(), can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference.

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

If the list you need to copy contains lists, then use the copy.deepcopy() function instead of copy.copy(). The deep copy() function will copy these inner lists as well.

❖
❖

Chapter- 5 Dictionaries and Structuring Data

Like a list, a dictionary is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair. In code, a dictionary is typed with braces, {}.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the myCat variable. This dictionary's keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

A dictionary is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type. You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key map to a value.

The association of a key and a value is called a key-value pair or sometimes an item. As an example, we'll build a dictionary those maps from English to Spanish words, so the keys and the values are all strings. The function dict creates a new dictionary with no items.

Because dict is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp) {}
```

The curly brackets, {}, represent an empty dictionary.

To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

This line creates an item that maps from the key 'one' to the value "uno".

If we print the dictionary again, we see a key-value pair with a colon between the key and value:

>>> print(eng2sp) {'one': 'uno'}

This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2sp, you might be surprised:

>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

>>> print(eng2sp) {'one': 'uno', 'three': 'tres', 'two': 'dos'}

The order of the key-value pairs is not the same. In general, the order of items in a dictionary is unpredictable. But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

>>> print(eng2sp['two']) 'dos' The key 'two' always maps to the value "dos" so the order of the items doesn't matter. If the key isn't in the dictionary, you get an exception

>>> print(eng2sp['four'])

KeyError: 'four'

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Example:

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

Output:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items().

The values returned by these methods are not true lists: They cannot be modified and do not have an append() method. But these data types (dict_keysdict_values, and dict_items, respectively) can be used in for loops.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

If you want a true list from one of these methods, pass its list-like return value to the list() function.

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age']. You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

We can use in and not in operators to see whether a certain key or value exists in a dictionary.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

The get() Method

Check whether a key exists in a dictionary before accessing that key's value. dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no 'eggs' key in the picnic Items dictionary, the default value 0 is returned by the get() method. Without using get(), the code would have caused an error message.

The setdefault() Method

Have to set a value in a dictionary for a certain key only if that key does not already have a value.

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The setdefault() method is a nice shortcut to ensure that a key exists.

Example: short program that counts the number of occurrences of each letter in a string.

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

The setdefault() method call ensures that the key is in the count dictionary (with a default value of 0).so the program doesn't throw a KeyError error when count[character] = count[character] + 1 is executed

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space character appears 13 times, and the uppercase letter *A* appears 1 time.

Pretty Printing

When you want a cleaner display of the items in a dictionary than what print () provides 2 methods pprint() and pformat() functions.

Example:

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

The output looks much cleaner, with the keys sorted.

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
```

```
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

The pprint.pprint() function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call pprint.pformat() instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

This is where lists and dictionaries can come in. You can use them to model real-world things, like chessboards. To play a game like Chessboard between players who are situated in different locations without internet will be by moving the chessboard move and mailing a postcard to each other describing each move. In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinates in Figure 5-1.

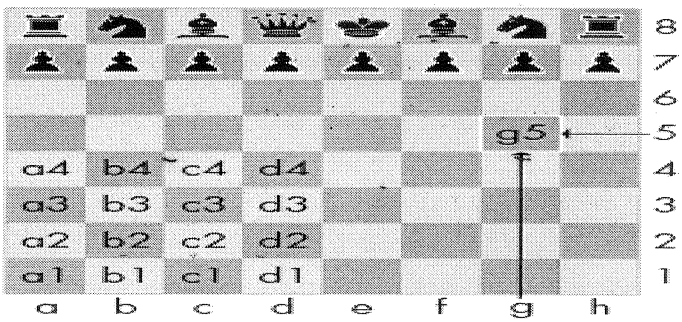


Figure 5-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination.

A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation *2. Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game. You can just read the mailed chess moves and update boards you have in your imagination.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-2.

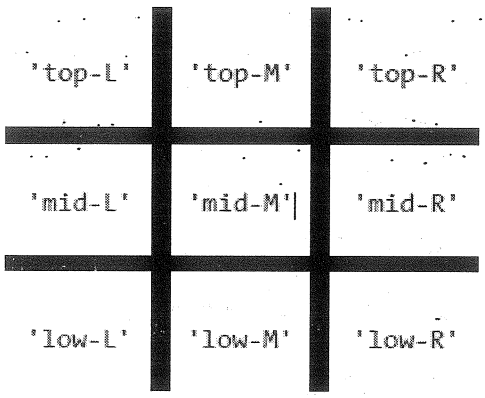


Figure 5-2: The slots of a tic-tac-toe board with their corresponding keys

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named theBoard.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

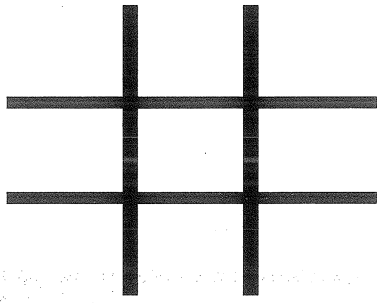


Figure 5-3: An empty tic-tac-toe board

Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in theBoard now represents the tic-tac-toe board in Figure 5-4.

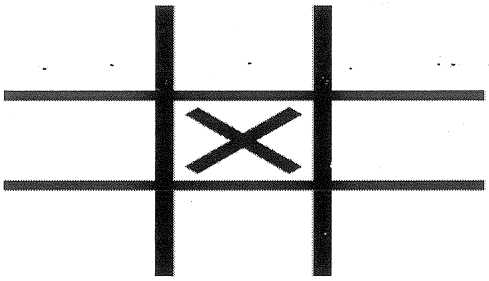


Figure 5-4: The first move

A board where player O has won by placing *O*s across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in theBoard now represents the tic-tac-toe board in Figure 5-5.

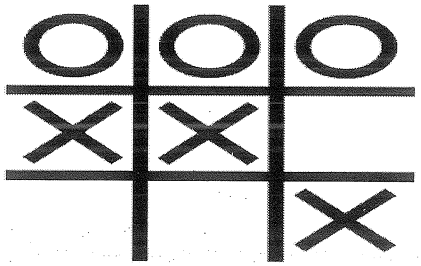


Figure 5-5: Player O wins.

Let's create a function to print the board dictionary onto the screen.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

printBoard() will print out a blank tic-tactoe board.

```
  |  |
-+-+-
  |  |
-+-+-
  |  |
```

The printBoard() function can handle any tic-tac-toe data structure you pass it. Because you created a data structure to represent a tic-tac-toe board and wrote code in print Board() to interpret that data structure, you now have a program that "models" the tic-tac-toe board the print Board() function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots.

If the dictionary you passed was missing, say, the 'mid-L' key, your program would no longer work. It throws Key Error: 'mid-L' Error. Now let's add code that allows the players to enter their moves.

The new code prints out the board at the start of each new turn u, gets the active player's move v, updates the game board accordingly w, and then swaps the active player x before moving on to the next turn.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

Nested Dictionaries and Lists

As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic.

The total Brought ()function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
    for k, v in guests.items():
        numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

The output:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

Chapter 6 - Manipulating Strings

Typing string values in Python code is fairly straightforward: They begin and end with a single quote There are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it

```
>>> spam = "That is Alice's cat."
```

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string. For example, the escape character for a single quote is \'. You can use this inside a string that begins and ends with single quotes.

```
>>> spam = 'Say hi to Bob\'s mother.'
```

The escape characters \' and \" let you put single quotes and double quotes inside your strings, respectively.

Table 6-1: Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

Place an r before the beginning quotation mark of a string to make it a raw string. A raw string completely ignores all escape characters and prints any backslash that appears in the string

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for regular expressions.

Multiline Strings with Triple Quotes While you can use the \n escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes.

Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

Escaping single and double quotes is optional in raw strings.

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""
```

Indexing and Slicing Strings Strings use indexes and slices the same way lists do. You can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

H	e	l	l	o		w	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Some operations

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
```

```
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

The in and not in Operators with Strings

The in and not in operators can be used with strings just like with list values. An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

Useful String Methods

The upper(), lower(), isupper(), and islower() String Methods

The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or www.it-ebooks.info Manipulating Strings 129 lowercase, respectively.

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well.

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX String Methods

These methods return a Boolean value that describes the nature of the string. Here are some common isX string methods:

- isalpha() returns True if the string consists only of letters and is not blank.
- isalnum() returns True if the string consists only of letters and numbers and is not blank.
- isdecimal() returns True if the string consists only of numeric characters and is not blank.
- isspace() returns True if the string consists only of spaces, tabs, and newlines and is not blank.
- istitle() returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Example Program:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

Output:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

The startswith() and endswith() String Methods

The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

The join() and split() String Methods

The join() method is useful when you have a list of strings that need to be joined together into a single string value. The join() method is called on a, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list.

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

The split() method does the opposite: It's called on a string value and returns a list of strings.

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

You can pass a delimiter string to the split() method to specify a different string to split upon.

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of split() is to split a multiline string along the newline characters.

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

Justifying Text with rjust(), ljust(), and center()

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string.

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

An optional second argument to rjust() and ljust() will specify a fill character other than a space character

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Example: These methods are especially useful when you need to print tabular data that has the correct spacing.

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.....     4
apples.............    12
cups...............     4
cookies.............  8000
```

Removing Whitespace with strip(), rstrip(), and lstrip()

The strip() string method will return a new string without any whitespace www.it-ebooks.info Manipulating Strings 135 characters at the beginning or end. The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

```
>>> spam = '    Hello World    '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World    '
>>> spam.rstrip()
'    Hello World'
```

Optionally, a string argument will specify which characters on the ends should be stripped

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Copying and Pasting Strings with the pyperclip Module

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software. Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A.(Refer to the procedure).

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

Of course, if something outside of your program changes the clipboard contents, the paste() function will return it.

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

Project: Password Locker

It's best to use password manager software on your computer that uses one master password to unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.Its for safety.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

Step 1: Program Design and Data Structures

t is the account's name—for instance, email or blog. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxFSSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable sys.argv. The first item in the sys.argv list should always be a string containing the program's filename ('pw.py'), and the second item should be the first command line argument.

For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the sys.argv list has fewer than two values in it).

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable account, you need to see whether it exists in the PASSWORDS dictionary as a key. If so, you want to copy the key's value to the clipboard using pyperclip.copy().

```
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

Type the following into the file editor and save the file as pw.bat in the C:\Windows folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing win-R and typing pw .

Project: Adding Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The bulletPointAdder.py script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

After running the above program the clipboard would then contain the following

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

Step 1: Copy and Paste from the Clipboard

py program should do the following:

1. Paste text from the clipboard
2. Do something to it
3. Copy the new text to the clipboard

Step 2: Separate the Lines of Text and Add the Star:

The call to pyperclip.paste() returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists".

Step 3: Join the Modified Lines: The lines list now contains modified lines that start with stars. But pyperclip.copy() is expecting a single string value, not a list of string values. To make this single string value, pass lines into the join() method to get a single string joined from the list's strings.

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes for "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

MODULE-3: Chapters 7-10 from T1

Contents:

Pattern Matching with Regular Expressions, Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The findall( ) Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the sub() Method, Managing Complex Regexes, Combining re .IGNORECASE, re .DOTALL, and re .VERBOSE, Project: Phone Number and Email Address Extractor,

Reading and Writing Files, Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module,Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard,

Organizing Files, The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,

Debugging, Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger.

Chapter 7

Pattern Matching with Regular Expressions

Most modern text editors and word processors, such as Microsoft Word or Open Office, have find and find-and replace features that can search based on regular expressions. Regular expressions are huge time-savers.

Finding Patterns of Text Without Regular Expressions:

- To find a phone number in a string.
  - pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.
- Let's use a function named isPhoneNumber() to check whether a string matches this pattern, returning either True or False.

```
def isPhoneNumber(text):
❶    if len(text) != 12:
        return False
    for i in range(0, 3):
❷        if not text[i].isdecimal():
            return False
❸    if text[3] != '-':
        return False
    for i in range(4, 7):
❹        if not text[i].isdecimal():
            return False
❺    if text[7] != '-':
        return False
    for i in range(8, 12):
❻        if not text[i].isdecimal():
            return False
❼    return True

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

When this program is run, the output looks like this:

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

- The isPhoneNumber() function has code that does several checks to see whether the string in text is a valid phone number.
- If any of these checks fail, the function returns False.

1. First the code checks that the string is exactly 12 characters

2. Then it checks that the area code (that is, the first three characters in text) consists of only numeric characters.
3. The rest of the function checks that the string follows the pattern of a phone number: The number must have the first hyphen after the area code,
4. three more numeric characters
5. then another hyphen,
6. Finally four more numbers.
7. If the program execution manages to get past all the checks, it returns True.

To find the above pattern of text in a larger string. Replace the last four print( ) function calls in isPhoneNumber.py with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
❶    chunk = message[i:i+12]
❷    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

1. On each iteration of the for loop, a new chunk of 12 characters from message is assigned to the variable chunk.
2. Pass chunk to isPhoneNumber( ) to see whether it matches the phone number pattern v, and if so, you print the chunk.

Finding Patterns of Text with Regular Expressions

- Regular expressions, called regexes for short, are descriptions for a pattern of text.
  - For example, a \d in a regex stands for a digit character— that is, any single numeral 0 to 9.
  - The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text the previous isPhoneNumber() function.(a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers)

- But regular expressions can be much more sophisticated.
  - For example, adding a 3 in curly brackets ({3}) after a pattern is like saying, "Match this pattern three times."
  - So the slightly shorter regex \d{3}-\d{3}-\d{4} also matches the correct phone number format.
- All the regex functions in Python are in the re module

```
>>> import re
```

a. **Creating Regex Objects**: Passing a string value representing your regular expression to re.compile() returns a Regex pattern object (or simply, a Regex object).

To create a Regex object that matches the phone number pattern, enter the following into the interactive shell. (Remember that \d means "a digit character" and \d\d\d-\d\d\d-\d\d\d\d is the regular expression for the correct phone number pattern.) using raw strings

```
>>> phoneNumRegex = re.compile( r '\d\d\d-\d\d\d-\d\d\d\d' )
```

Now the phoneNumRegex variable contains a Regex object.

Or **'\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'** (escape character should be used)

b. **Matching Regex Objects :**

- A Regex object's search() method searches the string it is passed for any matches to the regex.
  - The search() method will return None if the regex pattern is not found in the string. If the pattern is found, the search( ) method returns a Match object.
  - Match objects have a group( ) method that will return the actual matched text from the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

  - Desired pattern are passed to re.compile( ) and store the resulting Regex object in phoneNumRegex.

- Then we call search() on phoneNumRegex and pass search() the string we want to search for a match.
- The result of the search gets stored in the variable **mo**. Knowing that mo contains a Match object and not the null value None, we can call group() on mo to return the match.
- Writing mo.group() inside our print statement displays the whole match, 415-555-4242.

Review of Regular Expression :

Matching while there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to use a raw string.)
3. Pass the needed string to search into the Regex object's search() method. This returns a Match object.
4. Call the Match object's group() method to return a string of the actual matched text.

More Pattern Matching with Regular Expressions

(a) Grouping with Parentheses
(b) Matching Multiple Groups with the Pipe
(c) Optional Matching with the Question Mark
(d) Matching Zero or More with the Star
(e) Matching One or More with the Plus
(f) Matching Specific Repetitions with Curly Brackets

(a) **Grouping with Parentheses**: Parentheses have a special meaning in regular expressions.

(i) To separate the area code from the rest of the phone number.

Adding parentheses will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d).

- Then the group( ) match object method is used to grab the matching text from just one group.

- The first set of parentheses in a regex string will be group 1, The second set will be group 2.
- By passing the integer 1 or 2 to the group( ) match object method, different parts of the matched text can be retrieved.
- Passing 0 or nothing to the group( ) method will return the entire matched text.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

(ii) If you would like to retrieve all the groups at once, use the groups( ) method—note the plural form for the name.

Since mo.groups() returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous areaCode, mainNumber = mo.groups( ) line.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

(iii) if parenthesis needed to be matched in text, escape the (and) characters with a backslash should be used

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

The \(and \) escape characters in the raw string passed to re.compile() will match actual parenthesis characters.

(b) Matching Multiple Groups with the Pipe

- The | character is called a pipe. You can use it anywhere you want to match one of many expressions.
- For example, the regular expression **r' Batman | Tina Fey'** will match either 'Batman' or 'Tina Fey'. When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object.

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

- All matching occurrences can be found with the findall( ) method
- Can be used to match one of several patterns as part of regex.

For example,To match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it can be specified with parentheses.

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

- The method call mo.group( ) returns the full matched text 'Batmobile', while mo.group(1) returns just the part of the matched text inside the first parentheses group, 'mobile'. \
- If an actual pipe character is needed to match, escape character is used, like \|.

(c) Optional Matching with the Question Mark

- ? Will match "Match zero or one of the group preceding this question mark
- The ? character flags the group that precedes it as an optional part of the pattern.
- For example,

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

- The (wo)? part of the regular expression means that the pattern wo is an optional group. The regex will match text that has zero instances or one instance of wo in it.
- This is why the regex matches both 'Batwoman' and 'Batman'.
- " If actual question mark character is needed to match, it is escaped it with \?

(d) Matching Zero or More with the Star

- The * (called the star or asterisk) means "**match zero or more**"—the group that precedes the star can be completely absent or repeated over and over again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

  - For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string;
  - for 'Batwoman', the (wo)* matches one instance of wo;
  - and for 'Batwowowowoman', (wo)* matches four instances of wo.
- " If actual question * character is needed to match, it is escaped it with *.

(e) Matching One or More with the Plus

- The + (or plus) means **"match one or more."**
- The group preceding a plus must appear at least once. It is not optional.

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

- " If actual question + character is needed to match, it is escaped it with \+.

(f) Matching Specific Repetitions with Curly Brackets

- It repeats the group specific number of times, which is Followed in curly brackets.
- For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.
- Range can be specified by writing a minimum, a comma, and a maximum in between the curly brackets.
  - For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.
  - (Ha){3,} will match three or more instances of the (Ha) group,
  - (Ha){,5} will match zero to five instances.
- Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:
  - (i) (Ha){3} or (Ha)(Ha)(Ha)
  - (ii) (Ha){3,5} or ((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

- Here, (Ha){3} matches 'HaHaHa' but not 'Ha'. Since it doesn't match 'Ha', search() returns None.

Greedy and Nongreedy Matching

- Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible.
- The nongreedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a **question mark.**

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note : The question mark can have two meanings in regular expressions:

(i) declaring a nongreedy match or

(ii) flagging an optional group.

These meanings are entirely unrelated.

The findall( ) Method

- The findall( ) method will return the strings of every match in the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

- findall( ) will not return a Match object but ***a list of string***. Each string in the list is a piece of the searched text that matched the regular expression.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

- If there are groups in the regular expression, then findall( ) will return a ***list of tuples.***

- Each tuple represents a found match, and its items are the matched strings for each group in the regex.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

To summarize what the findall() method returns, remember the following:

1. When called on a regex with no groups, such as \d\d\d-\d\d\d-\d\d\d\d, the method findall() returns a **list of string matches**, such as ['415-555- 9999', '212-555-0000'].

2. When called on a regex that has groups, such as (\d\d\d)-(\d\d\d)-(\d\ d\d\d), the method findall() returns a list of tuples of strings

(one string for each group), such as [('415', '555', '1122'), ('212', '555', '0000')].

Character Classes

Character classes are nice for shortening regular expressions. \d is shorthand for the regular expression (0|1|2|3|4|5|6|7|8|9). There are many such shorthand character classes, as shown in Table 7-1.

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
\d	Any numeric digit from 0 to 9.
\D	Any character that is *not* a numeric digit from 0 to 9.
\w	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
\W	Any character that is *not* a letter, numeric digit, or the underscore character.
\s	Any space, tab, or newline character. (Think of this as matching "space" characters.)
\S	Any character that is *not* a space, tab, or newline.

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

- The regular expression \d+\s\w+ will match text that has one or more numeric digits (\d+), followed by a whitespace character (\s), followed by one or more letter/digit/underscore characters (\w+). The findall() method returns all matching strings of the regex pattern in a list.

Making Your Own Character Classes

- Character class can be defined by using **square brackets.** For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

- Ranges of letters or numbers can be specified by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.
- Inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the ., *, ?, or () characters with a preceding backslash. For example, the character class [0-5.] will match digits 0 to 5 and a period. You do not need to write it as [0-5\.].
- By placing a caret character (^) just after the character class's opening bracket, a negative character class can be made. A negative character class will match all the characters that are not in the character class. For example,

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

The Caret and Dollar Sign Characters

- The caret symbol (^) at the start of a regex indicates that a match must occur at the beginning of the searched text.
- Likewise, a dollar sign ($) at the end of the regex to indicate the string must end with this regex pattern.
- The ^ and $ together to indicate that the entire string must match the regex

(a) For example, the **r'^Hello'** regular expression string matches strings that begin with 'Hello'.

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

(b) The **r'\d$'** regular expression string matches strings that end with a numeric character from 0 to 9.

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

(c) The **r'^\d+$'** regular expression string matches strings that both begin and end with one or more numeric characters.

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12  34567890') == None
True
```

The Wildcard Character

- The **. (or dot) character** in a regular expression is called a wildcard and will match any character except for a newline.

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

- The dot character will match just one character, which is why the match for the text flat in the previous example matched only lat.
- To match an actual dot, escape the dot with a backslash: \. .

Matching Everything with Dot-Star

- The dot character means “any single character except the newline,” and the star character means “zero or more of the preceding character.”
- Matches everything and anything. For example, to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again. The dot-star (.*) can be used to stand in for that “anything.”

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

- The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and **question mark** (.*?). the following example shows difference between the greedy and nongreedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

- Both regexes roughly translate to "Match an opening angle bracket, followed by anything, followed by a closing angle bracket." But the string ' for dinner.>' has two possible matches for the closing angle bracket.
  - In the nongreedy version of the regex, Python matches the shortest possible string: ".
  - In the greedy version, Python matches the longest possible string: ' for dinner.>'.

Matching Newlines with the Dot Character

- The dot-star will match everything except a newline.
- By passing re.DOTALL as the second argument to re.compile(), the dot character can match all characters, including the newline character.

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

Review of Regex Symbols

- The ? matches zero or one of the preceding group.
- The * matches zero or more of the preceding group.
- The + matches one or more of the preceding group.
- The {n} matches exactly n of the preceding group.
- The {n,} matches n or more of the preceding group.
- The {,m} matches 0 to m of the preceding group.
- The {n,m} matches at least n and at most m of the preceding group.
- {n,m}? or *? or +? performs a nongreedy match of the preceding group.
- ^spam means the string must begin with spam.
- spam$ means the string must end with spam.
- The . matches any character, except newline characters.
- \d, \w, and \s match a digit, word, or space character, respectively.
- \D, \W, and \S match anything except a digit, word, or space character, respectively.
- [abc] matches any character between the brackets (such as a, b, or c).
- [^abc] matches any character that isn't between the brackets.

Case-Insensitive Matching

- Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
```

- To match the letters without worrying uppercase or lowercase. i.e. regex case-insensitive, **re.IGNORECASE** or **re.I** can be passed as a second argument to re.compile().

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group(
'robocop'
```

Substituting Strings with the sub( ) Method

- Regular expressions can also substitute new text in place of matched patterns.
- The sub() method for Regex objects is passed two arguments.
  - The first argument is a string to replace any matches.
  - The second is the string for the regular expression.
- The sub( ) method returns a string with the substitutions applied.

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

- If the matched text itself needed as part of the substitution. In the first argument to sub(), you can type \1, \2, \3, and so on, to mean "Enter the text of group 1, 2, 3, and so on, in the substitution."

- For example, to censor the names of the secret agents by showing just the first letters of their names.
  - use the regex '**Agent (\w)\w***' and pass **r'\1****'** as the first argument to sub(). The \1 in that string will be replaced by whatever text was matched by group 1— that is, the (\w) group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Managing Complex Regexes

- While matching complicated text patterns, re.compile() function can be made to ignore whitespace and comments inside the regular expression string.
- This "verbose mode" can be enabled by passing the variable **re.VERBOSE** as the second argument to re.compile(). Now instead of a hard-to-read regular expression like this:

phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4} (\s*(ext|x|ext.)\s*\d{2,5})?)')

This can be spread over **multiple lines with comments like this:**

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

- When,
  - re.VERBOSE is needed to write comments in regular expression
  - re.IGNORECASE is needed to ignore capitalization?

Unfortunately, the re.compile() function takes only a single value as its second argument

- This limitation can be overcome by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.
- So regular expression that's case-insensitive and includes newlines to match the dot character, you would form your re.compile() call like this:

>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)

All three options for the second argument will look like this:

>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)

Project: Phone Number and Email Address Extractor

- Say you have the boring task of finding every phone number and email address in a long web page or document.
- But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press ctrl-A to select all the text, press ctrl-C to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

consider the bigger picture of project.. For example, phone and email address extractor will need to do the following:

- Get the text off the clipboard.
- Find all phone numbers and email addresses in the text.
- Paste them onto the clipboard. Now you can start thinking about how this might work in code.

The code will need to do the following:

- Use the pyperclip module to copy and paste strings.
- Create two regexes, one for matching phone numbers and the other for matching email addresses.
- Find all matches, not just the first match, of both regexes.
- Neatly format the matched strings into a single string to paste.
- Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately. Each step is fairly manageable and expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as phoneAndEmail.py:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?              # area code
    (\s|-|\.)?                      # separator
    (\d{3})                         # first 3 digits
    (\s|-|\.)                       # separator
    (\d{4})                         # last 4 digits
    (\s*(ext|x|ext.)\s*(\d{2,5}))?  # extension
    )''', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

- The TODO comments are just a skeleton for the program. They'll be replaced as you write the actual code.
- The phone number begins with an optional area code, so the area code group is followed with a question mark. Since the area code can be
    - just three digits (that is, \d{3}) or
    - three digits within parentheses (that is, \(\d{3}\)),

A pipe joining those parts.

(\d{3}|\(\d{3}\))? is supposed to match.

- The phone number separator character can be a space (\s), hyphen (-), or period (.), so these parts should also be joined by pipes.
- The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by ext, x, or ext., followed by two to five digits.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following

```
import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Create email regex.
emailRegex = re.compile(r'''(
❶    [a-zA-Z0-9._%+-]+      # username
❷    @                      # @ symbol
❸    [a-zA-Z0-9.-]+         # domain name
     (\.[a-zA-Z]{2,4})      # dot-something
     )''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

- The username part of the email address

1. is one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen

   [a-zA-Z0-9._%+-].
2. The domain and username are separated by an @ symbol.
3. The domain name has a slightly less permissive character class with only letters, numbers, periods, and hyphens: [a-zA-Z0-9.-].
4. And last will be the "dot-com" part (technically known as the top-level domain), which can really be dot-anything. This is between two and four characters.

Step 3: Find All Matches in the Clipboard

- Python's re module finds all the matches on the clipboard.
- The pyperclip.paste( ) function will get a string value of the text on the clipboard, and the findall( ) regex method will return a list of tuples.

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())

❶ matches = []
❷ for groups in phoneRegex.findall(text):
      phoneNum = '-'.join([groups[1], groups[3], groups[5]])
      if groups[8] != '':
          phoneNum += ' x' + groups[8]
      matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
      matches.append(groups[0])

  # TODO: Copy results to the clipboard.
```

- There is one tuple for each match, and each tuple contains strings for each group in the regular expression.
- Remember that group 0 matches the entire regular expression, so the group at index 0 of the tuple is the one you are interested in. As you can see at

1. Stores the matches in a list variable named matches. It starts off as an empty list, and a couple for loops.
3. For the email addresses, it is appended to group 0 of each match .
2. For the matched phone numbers, not appended to group 0. While the program detects phone numbers in several formats, to append in a single, standard format. The phoneNum variable contains a string built from groups 1, 3, 5, and 8 of the matched text 2.

Step 4: Join the Matches into a String for the Clipboard

- The email addresses and phone numbers are in a list of strings in matches,
- The pyperclip.copy( ) function takes only a single string value, not a list of strings, so you call the join( ) method on matches to put them on the clipboard..

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

--snip--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

- Running the Program For an example,
  - open web browser to the No Starch Press contact page at http://www.nostarch.com/contactus.htm,
  - Press ctrl-A to select all the text on the page, and press ctrl-C to copy it to the clipboard.
  - When you run this program, the output will look something like this:

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

MODULE-3: Chapters 7-10 from T1

Contents:

Pattern Matching with Regular Expressions, Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The findall( ) Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the sub() Method, Managing Complex Regexes, Combining re .IGNORECASE, re .DOTALL, and re .VERBOSE, Project: Phone Number and Email Address Extractor,

Reading and Writing Files, Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module,Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard,

Organizing Files, The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,

Debugging, Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger.

Chapter 8

Reading and Writing Files

Files and File Paths

A file has two key properties:

- a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer.

  Ex: ***C:\Users\asweigart\Documents\project.docx***

  - The part of the filename after the last period is called the file's *extension* and tells file's type. *project.docx* is a Word document,
  - *Users*, *asweigart*, and *Documents* all refer to *folders* (also called *directories*). Folders can contain files and other folders. Figure 8-1 shows folder organization.

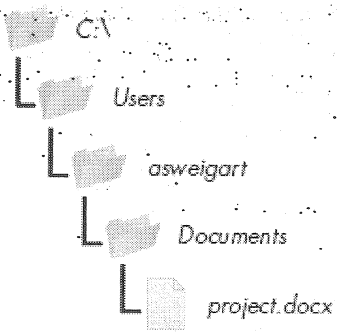


Figure 8-1: A file in a hierarchy of folders

- The *C:* part of the path is the *root folder*, which contains all other folders.
  - On Windows, the root folder is named *C:* and is also called the *C: drive*.
  - On OS X and Linux, the root folder is */*.

Note: folder names and filenames are not case sensitive on Windows and OS X, they are case sensitive on Linux.

Backslash on Windows and Forward Slash on OS X and Linux

- On Windows, paths are written using backslashes (\) as the separator between folder names.
- OS X and Linux uses the forward slash (/) as their path separator.
- If programs are to work on all operating systems, Python scripts handles both cases with the os.path.join( ) : function. If string is passed with values of individual file and folder names in

your path, os.path.join( ) will return a string with a file path using the correct path separators. Enter the following into the interactive shell:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
 'usr\\bin\\spam'
```

- Above shows windows interactive shell example so os.path.join('usr', 'bin', 'spam') returned 'usr\\bin\\spam'
- If called this function on OS X or Linux, the string would have been 'usr/bin/spam'.
- For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(os.path.join('C:\\Users\\asweigart', filename))

C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

The Current Working Directory

- Every program that runs on computer has a *current working directory*, or *cwd*.
- Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.
- The current working directory as a string value with the **os.getcwd( )** function and it can be changed with **os.chdir( ).**

```
>>> import os
>>> os.getcwd( )
'C:\\Python34'

>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd( )
'C:\\Windows\\System32'
```

- Here, the current working directory is set to *C:\Python34*, so the filename *project.docx* refers to *C:\Python34\project.docx*.
- When it is changed the current working directory is 'C:\\Windows\\System32'
- Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>
os.chdir('C:\\ThisFolderDoesNotExist')
```

FileNotFoundError: [WinError 2] The system cannot find the file specified: 'C:\\ThisFolderDoesNotExist'

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory

There are also the ***dot* (.)** and ***dot-dot* (..)** folders. These are not real folders but special names that can be used in a path.

- A single period ("dot") for a folder name is shorthand for "this directory."
- Two periods ("dot-dot") means "the parent folder."

Figure 8-2 is an example of some folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

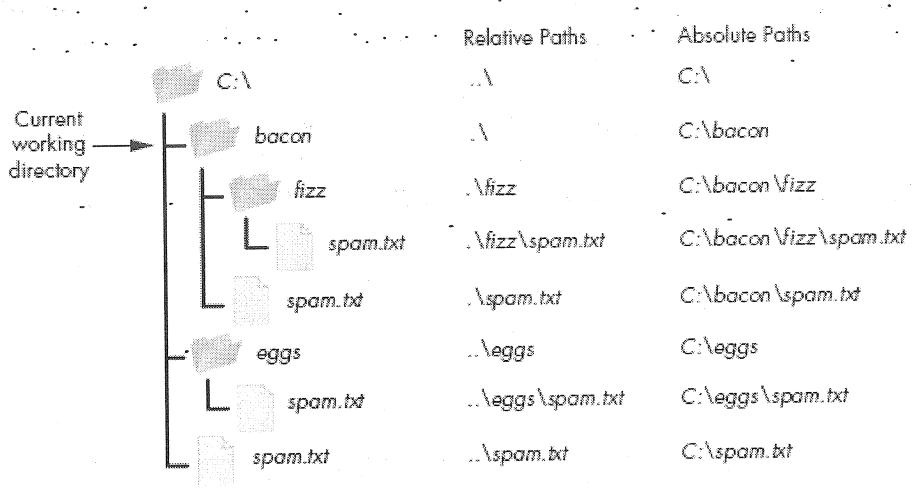


Figure 8-2: The relative paths for folders and files in the working directory C:\bacon

The .\ at the start of a relative path is optional.
For example, .*spam.txt* and *spam.txt* refer to the same file.

Creating New Folders with os.makedirs( )

new folders (directories) can be created with the os.makedirs( ) function.

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

- This will create not just the *C:\delicious* folder but also a *walnut* folder inside and a *waffles* folder inside *C:\delicious\walnut*.

- That is, os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists.

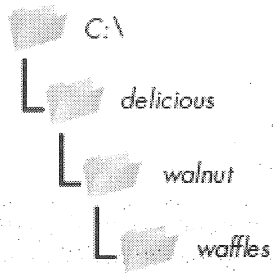


Figure 8-3: The result of os.makedirs('C:\\delicious\\walnut\\waffles')

The os.path Module

- The os.path module contains many helpful functions related to filenames and file paths. For os.path.join() to build paths in a way that will work on any operating system.
- Since os.path is a module inside the os module, it can be imported by simply running import os.

Otherwise, an error is raised: NameError: name 'os' is not defined error message.

Handling Absolute and Relative Paths

The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

• Calling **os.path.abspath(*path*)** will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.

• Calling **os.path.isabs(*path*)** will return True if the argument is an absolute path and False if it is a relative path.

• Calling **os.path.relpath(*path*, *start*)** will return a string of a relative path from the *start* path to *path*. If *start* is not provided, the current working directory is used as the start path.

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd( )
'C:\\Python34'
```

Note: Since *C:\Python34* was the working directory when os.path.abspath() was called, the "single-dot" folder represents the absolute path 'C:\\Python34'.

- Calling os.path.dirname(*path*) will return a string of everything that comes before the last slash in the path argument.
- Calling os.path.basename(*path*) will return a string of everything that comes after the last slash in the path argument. The dir name and base name of a path are outlined in Figure 8-4.

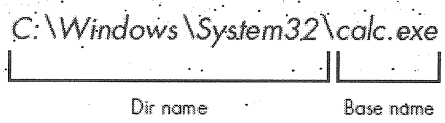


Figure 8-4: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```


- Calling os.path.split( ) returns tuple value path's dir name and base name together,

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Note:the above snippet is same as calling os.path.dirname( ) and os.path.basename( ) and placing their return values in a tuple.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

- To get list, use the split( ) string method: The split() string method will work to return a list of each part of the path. It will work on any operating system if os.path.sep is used.

- Where the os.sep variable is set to the correct folder-separating slash for the computer running the program.

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

On OS X and Linux systems, there will be a blank string at the start of the returned list:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

Finding File Sizes and Folder Contents

The os.path module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling **os.path.getsize(*path*)** will return the size in bytes of the file in the *path* argument.
- Calling **os.listdir(*path*)** will return a list of filename strings for each file in the *path* argument. (Note that this function is in the os module, not os.path.)

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

- os.path.getsize( ) and os.listdir( ) returns the total size of all the files in this directory.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
        totalSize = totalSize +
        os.path.getsize(os.path.join('C:\\Windows\\System32', filename))
>>> print(totalSize)
1117846456
```

- As it is looped each filename in the *C:\Windows\System32* folder, the totalSize variable is incremented by the size of each file.
  - Along with os.path.getsize( ), os.path.join( ) is used to join the folder name with the current filename.
  - The integer that os.path.getsize( ) returns is added to the value of totalSize. After looping through all the files,*C:\Windows\System32* folder.

Checking Path Validity

os.path module provides functions to check whether a given path exists and whether it is a file or folder.

- Calling os.path.exists(*path*) will return True if the file or folder referred to in the argument exists and will return False if it does not exist.
- Calling os.path.isfile(*path*) will return True if the path argument exist and is a file and will return False otherwise.
- Calling os.path.isdir(*path*) will return True if the path argument exists and is a folder and will return False otherwise.

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

- The os.path.exists( ) function can be used to check whether there is a DVD or flash drive currently attached to the computer . For instance, To check for a flash drive with the volume named *D:* on Windows computer,

```
>>> os.path.exists('D:\\')
False   # Oops! It looks like I forgot to plug in my flash drive.
```

The File Reading/Writing Process

- *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X's TextEdit application..

- *Binary files* are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If binary file is opened in Notepad or TextEdit, it will look like scrambled nonsense, like in Figure 8-5.

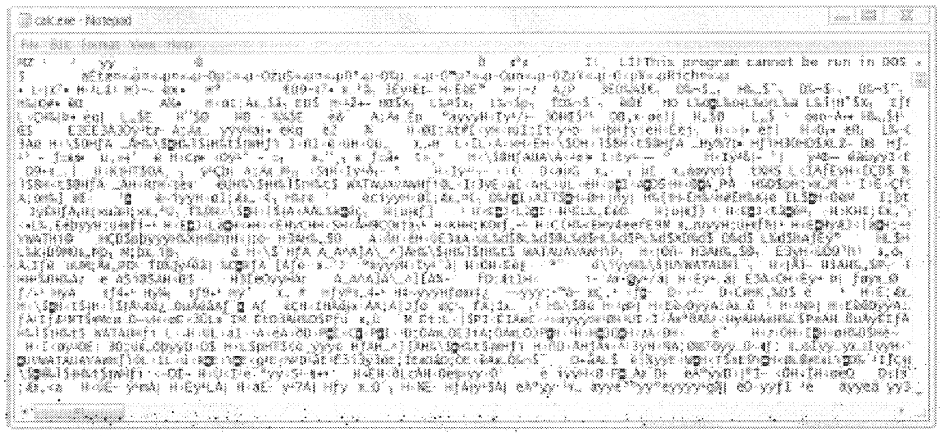


Figure 8-5: The Windows calc.exe program opened in Notepad

There are three steps to reading or writing files in Python.

1. Call the open( ) function to return a File object.
2. Call the read( ) or write( ) method on the File object.
3. Close the file by calling the close( ) method on the File object.

1. *Opening Files with the open( ) Function*

- To open a file with the open( ) function, string is passed with the path indicating the file to open; it can be either an absolute or relative path.
- The open( ) function returns a File object which is a file on computer
- Create a text file named *hello.txt* using Notepad or TextEdit. Type **Hello world!** as the content of this text file and save it in your user home folder. Then, In Windows,

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

If you're using OS X, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

- When a file is opened in read mode, Python lets you only read data from the file cannot be modified. Read mode is the default mode for files you open in Python.
- But mode can be explicitly specified by passing the string value 'r' as a second argument to open( ) .

```
>>> helloFile = open('/Users/asweigart/ hello.txt', 'r')
```

2. *(a) Reading the Contents of Files*

the File object's read() method reads the entire contents of a file as a string value, Let's continue with the *hello.txt* File object you stored in helloFile .

```
>>> helloContent = helloFile.read( )
>>> helloContent
'Hello world!'
```

- The readlines( ) method is used to get a *list* of string values from the file, one string for each line of text.
- For example, create a file named *sonnet29.txt* and contents with a paragraph

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines( )
```

[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And look upon myself and curse my fate,']

- Each of the string values ends with a newline character, \n , except for the last line of the file. A list of strings is often easier to work with than a single large string value.

2 *(b) Writing to Files*

- To write content to a file should be opened in "write plaintext" mode or "append plaintext" mode, or *write mode* and *append mode* for short.
  - Write mode will overwrite the existing file and start from scratch. Pass 'w' as the second argument to open( ) to open the file in write mode.
  - Append mode, on the other hand, will append text to the end of the existing file. Pass 'a' as the second argument to open( ) to open the file in append mode.
  - If the filename passed to open( ) does not exist, both write and append mode will create a new, blank file.
- After reading or writing a file, call the close() method before opening the file again.
- write() method does not automatically add a newline character to the end of the stringit has to be added explicitly.

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close( )
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
```

```
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

1. First, we open *bacon.txt* in write mode. Since there isn't a *bacon.txt* yet, Python creates one.
2. Calling write( ) on the opened file and passing write( ) the string argument 'Hello world! /n' writes the string to the file and returns the number of characters written, including the newline. Then we close the file.
3. To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write 'Bacon is not a vegetable.' to the file and close it.
4. Finally, to print the file contents to the screen, we open the file in its default read mode, call read( ), store the resulting File object in content, close the file, and print content.

Saving Variables with the shelve Module

If data needed to be saved Python programs, shelve modules are used.

- Variables can be saved in Python programs to binary shelf files using the shelve module.
- This way, program can restore data to variables from the hard drive.
- The shelve module will let add, Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close( )
```

To read and write data using the shelve module,

1. first import shelve.
2. Call shelve.open( ) and pass it a filename, and then store the returned shelf value in a variable.

3. Make changes to the shelf value as if it were a dictionary.
4. When done, call close( ) on the shelf value.

Here, our shelf value is stored in shelfFile. We create a list cats and write shelfFile['cats'] = cats to store the list in shelfFile as a value associated with the key 'cats' (like in a dictionary). Then we call close( ) on shelfFile. After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On OS X, only a single *mydata.db* file will be created.

- Shelf values don't have to be opened in read or write mode—they *can do both once opened.*

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close( )
```

Here, we open the shelf files to check that our data was stored correctly. Entering shelfFile['cats'] returns the same list that we stored earlier, so we know that the list is correctly stored, and we call close( ).

- Just like dictionaries, shelf values have ***keys() and values()*** methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, it should be passed the list() function to get them in list form.

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Saving Variables with the pprint.pformat() Function

- The pprint.pprint() function will "pretty print" the contents of a list or dictionary to the screen, while the pprint.pformat() function will return this same text as a string instead of printing it, that can be written to *.py* file.
- This file will be your very own module that you can import whenever you want to use the variable stored in it.

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Here,

1. import pprint to let us use pprint.pformat().
2. We have a list of dictionaries, stored in a variable cats. To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string. Once we have the data in cats as a string, it's easy to write the string to a file, which we'll call *myCats.py*.

- When the string from pprint.pformat(..) is saved to a *.py* file, the file is a module that can be imported just like any other.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

Project: Generating Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python. Here is what the program does:

- Creates 35 different quizzes.
- Creates 50 multiple-choice questions for each quiz, in random order.
- Provides the correct answer and three random wrong answers for each question, in random order.
- Writes the quizzes to 35 text files.
- Writes the answer keys to 35 text files. This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call open(), write(), and close() for the quiz and answer key text files.
- Use random.shuffle() to randomize the order of the questions and multiple- choice options.

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
import random
# The quiz data. Keys are states and values are their capitals.
capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix', 'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver', 'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee', 'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois': 'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas': 'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine': 'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan': 'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri': 'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada': 'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh', 'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City', 'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence', 'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee': 'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont': 'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
for quizNum in range(35):
    # TODO: Create the quiz and answer key files.
    # TODO: Write out the header for the quiz.
    # TODO: Shuffle the order of the states.
    # TODO: Loop through all 50 states, making a question for each.
```

- Since this program will be randomly ordering the questions and answers, import the random module.
- The capitals variable contains a dictionary with US states as keys and their capitals as values.
- To create 35 quizzes, the code that actually generates the quiz and answer key files (marked with TODO comments for now) will go inside a for loop that loops 35 times.

Step 2: Create the Quiz File and Shuffle the Question Order

The code in the loop will be repeated 35 times—once for each quiz.

- First create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period.
- Then get a list of states in randomized order, which can be used later to create the questions and answers for the quiz. *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--
# Generate 35 quiz files. for quizNum in range(35):
# Create the quiz and answer key files.
quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')
# Write out the header for the quiz.
quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
quizFile.write('\n\n')
# Shuffle the order of the states.
states = list(capitals.keys())
random.shuffle(states)

# TODO: Loop through all 50 states, making a question for each.
```

- The filenames for the quizzes will be *capitalsquiz<N>.txt*, where *<N>* is a unique number for the quiz that comes from quizNum, the for loop's counter.
- The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*.
- Each time through the loop, the %s placeholder in 'capitalsquiz%s.txt' and 'capitalsquiz_answers%s.txt' will be replaced by (quizNum + 1), so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*.
- These files will be created with calls to the open( ) function , with 'w' as the second argument to open them in write mode.

- The write( ) statements create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the random.shuffle( ) function , which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate the answer options for each question, which will be multiple choices from A to D. which needs another for loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third for loop nested inside to generate the multiple-choice options for each question.

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
--snip--

# Loop through all 50 states, making a question for each.
for questionNum in range(50):
# Get right and wrong answers.
correctAnswer = capitals[states[questionNum]]
wrongAnswers = list(capitals.values())
del wrongAnswers[wrongAnswers.index(correctAnswer)]
wrongAnswers = random.sample(wrongAnswers, 3)
answerOptions = wrongAnswers + [correctAnswer]
random.shuffle(answerOptions)

# TODO: Write the question and answer options to the quiz file.
# TODO: Write the answer key to a file.
```

The correct answer is easy to get—it's stored as a value in the capitals dictionary. This loop will loop through the states in the shuffled states list, from states[0] to states[49], find each state in capitals, and store that state's corresponding capital in correctAnswer.

The list of possible wrong answers is trickier.

- duplicate *all* the values in the capitals dictionary, deleting the correct answer

and selecting three random values from this list x.

- The random.sample( ) function makes it easy to do this selection. The full list of answer options is the combination of these three wrong answers with the correct answers . Finally, the answers need to be randomized so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

#! python3

randomQuizGenerator.py - Creates quizzes with questions and answers in

random order, along with the answer key.

--snip—

Loop through all 50 states, making a question for each.

for questionNum in range(50):

--snip--

 # Write the question and the answer options to the quiz file. quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1, states[questionNum]))

or i in range(4):

quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i])) quizFile.write('\n')

 # Write the answer key to a file.

answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[

answerOptions.index(correctAnswer)])) quizFile.close() answerKeyFile.close( )

- A for loop that goes through integers 0 to 3 will write the answer options in the answerOptions list.
- The expression 'ABCD'[i] at v treats the string 'ABCD' as an array and will evaluate to 'A','B', 'C', and then 'D' on each respective iteration through the loop.
- In the final line, the expression answerOptions.index(correctAnswer) will find the integer index of the correct answer in the randomly ordered answer options, and 'ABCD'[answerOptions.index(correctAnswer)] will evaluate to the correct answer's letter to be written to the answer key file.
- After you run the program, this is how *capitalsquiz1.txt* file will look, though of course questions and answer options may be different from those shown here, depending on the outcome of random.shuffle( ) calls:

Name:

Date:

Period:

State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
A. Hartford
B. Santa Fe
C. Harrisburg
D. Charleston
2. What is the capital of Colorado?
A. Raleigh
B. Harrisburg
C. Denver
D. Lincoln
--snip—

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

1. D
2. C
3. A
4. C

--snip—

Project: Multiclipboard

The task of filling out many forms in a web page or software with several text fields. The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time. If you have several different pieces of text that you need to copy and paste, you have to keep highlighting and copying the same few things over and over again.

You can write a Python program to keep track of multiple pieces of text. This "multiclipboard" will be named *mcb.pyw*. The *.pyw* extension means that Python won't show a Terminal window when it runs this program.

The program will save each piece of clipboard text under a keyword. For example, when you run ***py mcb.pyw save spam***, the current contents of the clipboard will be saved with the keyword *spam*. This text can later be loaded to the clipboard again by running ***py mcb.pyw spam***. And if the user forgets what keywords they have, they can run ***py mcb.pyw*** list to copy a list of all keywords to the clipboard.

Here's what the program does:

The command line argument for the keyword is checked.

• If the argument is save, then the clipboard contents are saved to the keyword.

• If the argument is list, then all the keywords are copied to the clipboard.

• Otherwise, the text for the keyword is copied to the keyboard. This means the code will need to do the following:

• Read the command line arguments from sys.argv.

• Read and write to the clipboard.

• Save and load to a shelf file. If you use Windows, you can easily run this script from the Run... window by creating a batch file named *mcb.bat* with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
# Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
# py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
# py.exe mcb.pyw list - Loads all keywords to clipboard. v import shelve, pyperclip, sys
mcbShelf = shelve.open('mcb')
# TODO: Save clipboard content.
# TODO: List keywords and load content.
mcbShelf.close( )
```

import

- Copying and pasting will require the pyperclip module,
- Reading the command line arguments will require the sys module.
- The shelve module : Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into program. The shelf file will be named with the prefix *mcb*.

Step 2: Save Clipboard Content with a Keyword

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case.

following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--
# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
        # TODO: List keywords and load content.
mcbShelf.close()
```

- If the first command line argument (which will always be at index 1 of the sys.argv list) is 'save' ,
- The second command line argument is the keyword for the current content of the clipboard.

The keyword will be used as the key for mcbShelf, and the value will be the text currently on the clipboard . If there is only one command line argument, assume it is either 'list' or a keyword to load content onto the clipboard. For now, just put a TODO comment there .

Step 3: List Keywords and Load a Keyword's Content

Finally, let's implement the two remaining cases: The user wants to load clipboard text in from a keyword, or they want a list of all available keywords.

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--
# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower( ) == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste( )
elif len(sys.argv) == 2:
        # List keywords and load content.
```

```
    if sys.argv[1].lower( ) == 'list':
        pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
        pyperclip.copy(mcbShelf[sys.argv[1]])
mcbShelf.close( )
```

If there is only one command line argument, first let's check whether it's

- 'list'. If so, a string representation of the list of shelf keys will be copied to the clipboard. The user can paste this list into an open text editor to read it. Otherwise, you can assume the command line argument is a keyword.
- If this keyword exists in the mcbShelf shelf as a key, you can load the value onto the clipboard.

Launching this program has different steps depending on what operating system.

MODULE-3: Chapters 7-10 from T1

Contents:

Pattern Matching with Regular Expressions, Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The findall( ) Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the sub() Method, Managing Complex Regexes, Combining re .IGNORECASE, re .DOTALL, and re .VERBOSE, Project: Phone Number and Email Address Extractor,

Reading and Writing Files, Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module,Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard,

Organizing Files, The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,

Debugging, Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger.

Chapter 9

Organizing Files

Python programs can also organize preexisting files on the hard drive like copying, renaming, moving, or compressing folder full of dozens, hundreds, or even thousands of files

The shutil Module

The shutil (or shell utilities) module has functions to copy, move, rename, and delete files in Python programs. To use the shutil functions, it has to be imported by **import shutil.**

(a) Copying Files and Folders
(b) Moving and Renaming Files and Folders
(c) Permanently Deleting Files and Folders
(d) Safe Deletes with the send2trash Module

(a) Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders:

- Calling shutil.copy(*source*, *destination*) will copy the file at the path *source* to the folder at the path *destination*. (Both *source* and *destination* are strings.) If *destination* is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

- The first shutil.copy( ) call copies the file at *C:\spam.txt* to the folder *C:\delicious*. The return value is the path of the newly copied file. Note that since a folder was specified as the destination , the original *spam.txt* filename is used for the new, copied file's filename.
- The second shutil.copy( ) call also copies the file at *C:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt*.

- shutil.copytree(*source*, *destination*) will copy an entire folder and every folder and file contained in it. *source*, along with all of its files and subfolders, to the folder at the path *destination*. The *source* and *destination* parameters are both strings. The function returns a string of the path of the copied folder.

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

The shutil.copytree( ) call creates a new folder named *bacon_backup* with the same content as the original *bacon* folder. You have now safely backed up your precious, precious bacon.

(b) Moving and Renaming Files and Folders

- shutil.move(*source*, *destination*) will move the file or folder at the path *source* to the path *destination* and will return a string of the absolute path of the new location. If *destination* points to a folder, the *source* file gets moved into *destination* and keeps its current filename.

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named *eggs* already exists in the *C:* directory, this shutil.move( ) calls says, "Move *C:\bacon.txt* into the folder *C:\eggs*." If there had been a *bacon.txt* file already in *C:\eggs*, it would have been overwritten.

Since it's easy to accidentally overwrite files in this way, user should take some care when using move( ).

- The *destination* path can also specify a filename. In the following example, the *source* file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, "Move *C:\bacon.txt* into the folder *C:\eggs*, and while you're at it, rename that *bacon.txt* file to *new_bacon.txt*."

- if there is no *eggs* folder, then move( ) will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Here, move( ) can't find a folder named *eggs* in the *C:* directory and so assumes that *destination* must be specifying a filename, not a folder. So the *bacon.txt* text file is renamed to *eggs* (a text file without the *.txt* file extension)— probably not what you wanted!

- Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
```

```
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
  FileNotFoundError: [WinError 3] The system cannot find the path specified:
  'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:
 Traceback (most recent call last):
----------
----------
 FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\ eggs\\ham'
```

Python looks for *eggs* and *ham* inside the directory *does_not_exist*. It doesn't find the nonexistent directory, so it can't move *spam.txt* to the path you specified.

(c) Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.

• Calling os.unlink(*path*) will delete the file at *path*.

• Calling os.rmdir(*path*) will delete the folder at *path*. This folder must be empty of any files or folders.

• Calling shutil.rmtree(*path*) will remove the folder at *path*, and all files and folders it contains will also be deleted.

Here is a Python program that was intended to delete files that have the *.txt* file extension but has a typo (highlighted in bold) that causes it to delete *.rxt* files instead:

```
import os
for filename in os.listdir( ):
    if filename.endswith('.rxt'):
os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
for filename in os.listdir( ):
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

Now the os.unlink( ) call is commented, so Python ignores it. Instead, it prints the filename of the file that would have been deleted. Once it is certain, delete the print(filename) line and uncomment the os.unlink(filename) line. Then run the program again to actually delete the files.

(c) Safe Deletes with the send2trash Module

- A much better way to delete files and folders is with the third-party send2trash module.
- You can install this module by running *pip install send2trash* from a Terminal window. (
- Using send2trash is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with send2trash you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed send2trash,

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
```

```
>>> baconFile.close( )
>>> send2trash.send2trash('bacon.txt')
```

- send2trash.send2trash( ) function is used to delete files and folders.
- But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does.

Note: that the send2trash( ) function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

Python provides functions to rename every file in some folder and also every file in every subfolder of that folder. That is, walk through the directory tree. Let's look at the *C:\delicious* folder with its contents, shown in Figure 9-1.

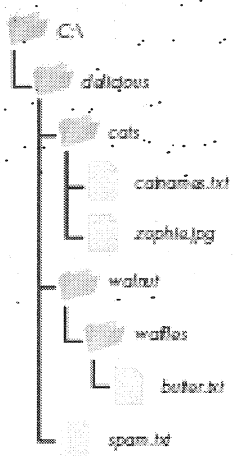


Figure 9-1: An example folder that contains three folders and four files

Here is an example program that uses the os.walk( ) function on the directory tree from Figure 9-1:

```
import os
for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)
    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)
    print(' ')
```

- The os.walk( ) function is passed a single string value: the path of a folder.
- os.walk( ) in a for loop statement to walk a directory tree, and returns three values on each iteration through the loop:

1. A string of the current folder's name (
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

(current folder: the folder for the current iteration of the for loop. The current working directory of the program is *not* changed by os.walk().)

The variable names for the three values listed earlier can be choosed. usually the names foldername, subfolders, and filenames are used. When running this program, it will output the following:

The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt
The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg
The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles
The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.

- Since os.walk( ) returns lists of strings for the subfolder and filename variables,these lists can be used in their own for loops. Replace the print( ) function calls with your own custom code.

Compressing Files with the zipfile Module

- ZIP files (with the *.zip* file extension), which can hold the compressed contents of many other files.
- Compressing a file reduces its size, which is useful when transferring it over the Internet. And since a ZIP file can also contain multiple files and subfolders, This single file, called an *archive file*, can then be, say, attached to an email.
- Python programs can both create and open (or *extract*) ZIP files using functions in the ***zipfile module.***
- Example: a ZIP file named *example.zip* that has the contents shown in Figure 9-2.

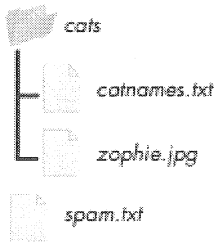


Figure 9-2: The contents of example.zip

(a) Reading ZIP Files
(b) Extracting from ZIP Files
(c) Creating and Adding to ZIP Files

(a) Reading ZIP Files

- To read the contents of a ZIP file, first a ***ZipFile*** object must be created (note the capital letters *Z* and *F*). ZipFile objects are conceptually similar to the File objects, they are values through which the program interacts with the file.
- To create a ZipFile object, call the ***zipfile.ZipFile( )*** function, passing it a string of the *.zip* file's filename. *Note that zipfile is the name of the Python module, and ZipFile( ) is the name of the function.*

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist( )
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
>>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close( )
```

- A ZipFile object has a namelist( ) method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the getinfo( )
- ZipFile method to return a ZipInfo object about that particular file. ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively.
- While a ZipFile object represents an entire archive file, a ZipInfo object holds useful information about a *single file* in the archive.
- *example.zip* is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with %s.

(b) Extracting from ZIP Files

- The extractall( ) method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.extractall( )
>>> exampleZip.close( )
```

- After running this code, the contents of *example.zip* will be extracted to *C:*. Optionally, a folder name to extractall( ) can be passed to have it extract the files into a folder other than the current working directory.
- If the folder passed to the extractall( ) method does not exist, it will be created.
- The extract( ) method for ZipFile objects will extract a single file from the ZIP file.

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close( )
```

- The string passed to extract( ) must match one of the strings in the list returned by namelist( ). Optionally, It can be passed as second argument to extract( ) to extract the file into a folder other than the current working directory.

- If this second argument is a folder that doesn't yet exist, Python will create the folder. The value that extract( ) returns is the absolute path to which the file was extracted.

(c) Creating and Adding to ZIP Files

- To create own compressed ZIP files, the ZipFile object must be opened in *write mode* by passing 'w' as the second argument.
- When you pass a path to the write( ) method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file.
- The write( ) method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; it can be always set to zipfile.ZIP_DEFLATED. (This specifies the *deflate* compression algorithm, which works well on all types of data.)

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close( )
```

- This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Note: *write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in append mode.*

Project: Renaming Files with American-Style Dates to European-Style Dates

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to Europeanstyle dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead. Here's what the program does:

- It searches all the filenames in the current working directory for American-style dates.
- When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

• Create a regex that can identify the text pattern of American-style dates.

• Call os.listdir(.) to find all the files in the working directory.

• Loop over each filename, using the regex to check whether it has a date.

• If it has a date, rename the file with shutil.move( ). For this project, open a new file editor window and save code as

renameDates.py.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as TODO makes them easy to find using IDLE's ctrl-F find feature. Make your code look like the following:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

import shutil, os, re

# Create a regex that matches files with the American date format.
datePattern = re.compile(r"""^(.*?)      # all text before the date
((0|1)?\d)-        # one or two digits for the month
((0|1|2|3)?\d)-   # one or two digits for the day
((19|20)\d\d)    # four digits for the year
(.*?)$             # all text after the date
""", re.VERBOSE)

# TODO: Loop over the files in the working directory.
# TODO: Skip files without a date.
# TODO: Get the different parts of the filename.
# TODO: Form the European-style filename.
# TODO: Get the full, absolute file paths.
# TODO: Rename the files.
```

- shutil.move( ) function can be used to rename files: Its arguments are the name of the file to rename and the new filename. Because this function exists in the shutil module, import that module.
- But before renaming the files, identify which files to be renamed. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.
- regular expression can be used to identify this pattern. After importing the ***re*** module at the top, call re.compile( ) to create a Regex objectv.
- Passing re.VERBOSE for the second argument allows whitespace and comments in the regex string to make it more readable.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from os.listdir( ) and match them against the regex. Any files that do not have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Filling the first three TODOs in program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.
--snip—


# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
        mo = datePattern.search(amerFilename)
# Skip files without a date.
        if mo == None:
                continue
        # Get the different parts of the filename.
        beforePart = mo.group(1)
        monthPart = mo.group(2)
        dayPart = mo.group(4)
```

```
    yearPart = mo.group(6)
    afterPart = mo.group(8)
--snip--
```

- If the Match object returned from the search( ) method is None, then the filename in amerFilename does not match the regular expression. The continue statement will skip the rest of the loop and move on to the next filename.
- Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart.
- The strings in these variables will be used to form the European-style filename in the next step. To keep the group numbers straight, try reading the regex from the beginning and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. For example:

```
datePattern = re.compile(r"""^(1) # all text before the date
(2 (3) )- # one or two digits for the month
(4 (5) )- # one or two digits for the day
(6 (7) ) # four digits for the year
(8)$ # all text after the date
""", re.VERBOSE)
```

Here, the numbers **1** through **8** represent the groups in the regular expression.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: The date comes before the month. Fill in the three remaining TODOs in with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--
```

```
# Form the European-style filename.
euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart + afterPart
# Get the full, absolute file paths.
absWorkingDir = os.path.abspath('.')
amerFilename = os.path.join(absWorkingDir, amerFilename)
euroFilename = os.path.join(absWorkingDir, euroFilename)
# Rename the files.
print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
#shutil.move(amerFilename, euroFilename) # uncomment after testing
```

- Store the concatenated string in a variable named euroFilename.
- Then, pass the original filename in amerFilename and the new euroFilename variable to the shutil.move( ) function to rename the file.

This program has the shutil.move( ) call commented out and instead prints the filenames that will be renamed . Running the program like this first can let you double-check that the files are renamed correctly. Then uncommenting the shutil.move( ) call and run the program again to actually rename the files.

Complete Code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

import shutil, os, re

# Create a regex that matches files with the American date format.
datePattern = re.compile(r"""^(.*?)      # all text before the date
((0|1)?\d)-        # one or two digits for the month
((0|1|2|3)?\d)-   # one or two digits for the day
((19|20)\d\d)    # four digits for the year
(.*?)$           # all text after the date
""", re.VERBOSE)

# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)
# Skip files without a date.
    if mo == None:
        continue
    # Get the different parts of the filename.
    beforePart = mo.group(1)
    monthPart = mo.group(2)
    dayPart = mo.group(4)
    yearPart = mo.group(6)
    afterPart = mo.group(8)

    # Form the European-style filename.
    euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart + afterPart
    # Get the full, absolute file paths.
    absWorkingDir = os.path.abspath('.')
    amerFilename = os.path.join(absWorkingDir, amerFilename)
    euroFilename = os.path.join(absWorkingDir, euroFilename)
    # Rename the files.
    print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
    #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

Project: Backing Up a Folder into a ZIP File

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. It would be simpler to run a program that does this boring task for you. For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named backupToZip( ). This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

import zipfile, os
def backupToZip(folder):
    # Backup the entire contents of "folder" into a ZIP file.
    folder = os.path.abspath(folder) # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
    number = 1
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # TODO: Create the ZIP file.
    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')
    backupToZip('C:\\delicious')
```

- Do the basics first: Add the shebang (#!) line, describe what the program does,
- import the zipfile and os modules.
- Define a backupToZip( ) function that takes just one parameter, folder. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder folder, and add each of the subfolders and files to the ZIP file.
- Write TODO comments for these steps in the source code to remind yourself to do them later x. The first part, naming the ZIP file, uses the base name of the absolute path of folder. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where *N* = 1 is the first time you run the program, *N* = 2 is the second time, and so on.

You can determine what *N* should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named number for *N*v, and keep ineremenling it inside the loop that calls os.path.exists( ) to check whether the file exists. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # Create the ZIP file.
    print('Creating %s...' % (zipFilename))
    backupZip = zipfile.ZipFile(zipFilename, 'w')
```

```
        # TODO: Walk the entire folder tree and compress the files in each folder.
        print('Done.')

backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the zipFilename variable, you can call zipfile.ZipFile( ) to actually create the ZIP file u. Be sure to pass 'w' as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the os.walk( ) function to do the work of listing every file in the folder and its subfolders.

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.
--snip--

        # Walk the entire folder tree and compress the files in each folder.
        for foldername, subfolders, filenames in os.walk(folder):
                print('Adding files in %s...' % (foldername))
                # Add the current folder to the ZIP file.
                backupZip.write(foldername)
                # Add all the files in this folder to the ZIP file.
                for filename in filenames:
                        newBase / os.path.basename(folder) + '_'
                        if filename.startswith(newBase) and filename.endswith('.zip'):
                                continue     # don't backup the backup ZIP files
                        backupZip.write(os.path.join(foldername, filename))
        backupZip.close()
        print('Done.')
backupToZip('C:\\delicious')
```

- os.walk( ) in first for loop , and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder. In the second for loop, the folder is added to the ZIP file.

The nested for loop can go through each filename in the filenames list . Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

The second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

Complete code:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

import zipfile, os
def backupToZip(folder):
# Backup the entire contents of "folder" into a ZIP file.
    folder = os.path.abspath(folder) # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
    number = 1
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

        # Create the ZIP file.
        print('Creating %s...' % (zipFilename))
        backupZip = zipfile.ZipFile(zipFilename, 'w')

        # Walk the entire folder tree and compress the files in each folder.
        for foldername, subfolders, filenames in os.walk(folder):
            print('Adding files in %s...' % (foldername))
            # Add the current folder to the ZIP file.
            backupZip.write(foldername)
            # Add all the files in this folder to the ZIP file.
            for filename in filenames:
                newBase / os.path.basename(folder) + '_'
            if filename.startswith(newBase) and filename.endswith('.zip'):
                continue      # don't backup the backup ZIP files
            backupZip.write(os.path.join(foldername, filename))
        backupZip.close()
        print('Done.')
        backupToZip('C:\\delicious')
```

MODULE-3: Chapters 7-10 from T1

Contents:

Pattern Matching with Regular Expressions, Finding Patterns of Text Without Regular Expressions, Finding Patterns of Text with Regular Expressions, More Pattern Matching with Regular Expressions, Greedy and Nongreedy Matching, The findall( ) Method, Character Classes, Making Your Own Character Classes, The Caret and Dollar Sign Characters, The Wildcard Character, Review of Regex Symbols, Case-Insensitive Matching, Substituting Strings with the sub() Method, Managing Complex Regexes, Combining re .IGNORECASE, re .DOTALL, and re .VERBOSE, Project: Phone Number and Email Address Extractor,

Reading and Writing Files, Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module,Saving Variables with the pprint.pformat() Function, Project: Generating Random Quiz Files, Project: Multiclipboard,

Organizing Files, The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,

Debugging, Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE's Debugger.

Chapter 10

Debugging

The debugger is a feature of IDLE that executes a program one instruction at a time, giving a chance to inspect the values in variables while code runs, and track how the values change over the course of program.

Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code which can be handled by Python's exceptions with try and except statements
- But users can also raise own exceptions. Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the except statement." Exceptions are raised with a raise statement. In code, a raise statement consists of the following:
  - The raise keyword
  - A call to the Exception( ) function
  - A string with a helpful error message passed to the Exception( ) function

For example,

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception's error message. Often it's the code that calls the function, not the fuction itself, that knows how to handle an expection.

So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

For example, open a new file editor window, enter the following code, and save the program as *boxPrint.py*:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
        raise Exception('Symbol must be a single character string.')
    if width <= 2:
        raise Exception('Width must be greater than 2.')
    if height <= 2:
        raise Exception('Height must be greater than 2.')
    print(symbol * width)
```

```
        for i in range(height - 2):
            print(symbol + (' ' * (width - 2)) + symbol)
        print(symbol * width)
for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
        try:
            boxPrint(sym, w, h)
        except Exception as err:
            print('An exception happened: ' + str(err))
```

- Here boxPrint( ) function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height.
- Say the character to be a single character, and the width and height to be greater than 2.
- if statements to raise exceptions if these requirements aren't satisfied.
- Later, when called boxPrint( ) with various arguments, try/except will handle invalid arguments. This program uses the ***except Exception as err*** form of the except statement. If an Exception object is returned from boxPrint( ), this except statement will store it in a variable named **err.**
- The Exception object can then be converted to a string by passing it to str( ) to produce a user friendly error message.

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the try and except statements, errors can be handled more gracefully instead of letting the entire program crash.

Getting the Traceback as a String

When Python encounters an error, it produces a treasure trove of error information called the *traceback*. The traceback includes the error message,

- the line number of the line that caused the error,
- and the sequence of the function calls that led to the error.

This sequence of calls is called the ***call stack.***

Open a new file editor window in IDLE, enter the following program, and save it as *errorExample.py*:

```
def spam( ):
    bacon( )
def bacon( ):
    raise Exception('This is the error message.')
spam( )
```

the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in <module>
    spam( )
  File "errorExample.py", line 2, in spam
    bacon( )
  File "errorExample.py", line 5, in bacon
    raise Exception('This is the error message.')
Exception: This is the error message.
```

From the traceback,

- The error happened on line 5, in the bacon( ) function. This particular call to bacon( ) came from line 2, in the spam( ) function, which in turn was called on line 7.
- In programs where functions can be called from multiple places, the call stack can help determining which call led to the error. The traceback is displayed by Python whenever a raised exception goes unhandled.

But it can be obtained as a string by calling ***traceback.format_exc( ).*** This function helps to handle the exception gracefully. Import Python's **traceback** module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a log file and keep program running. the log file can be checked later to debug the program.

```
>>> import traceback
>>> try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close( )
        print('The traceback info was written to errorInfo.txt.')
        116
        The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the write( ) method, since 116 characters were written to the file. The traceback text was written to *errorInfo.txt*.

```
Traceback (most recent call last):
File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by *assert statements*. If the sanity check fails, then an *AssertionError exception is raised*. In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

For example, enter the following into the interactive shell:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".
AssertionError: The pod bay doors need to be "open".
```

- Here podBayDoorStatus is set to 'open', so from now on, the value of this variable is expected to be 'open'.
- In a program that uses this variable, might have a lot of code under the assumption that the value is 'open'—code that depends on its being 'open' in order to work as expected.
- So add an assertion to make sure to assumption is right, podBayDoorStatus is 'open'. Here, the message *'The pod bay doors need to be "open". is included'* so it'll be easy to see what's wrong if the assertion fails.
- Later, assigning podBayDoorStatus another value, but don't notice it among many lines of code. The assertion catches this mistake and clearly tells what's wrong.
- In plain English, an assert statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program. *if an assert fails, program should crash.* This will reduce the amount of code you will have to check before finding the code that's causing the bug.
- Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Using an Assertion in a Traffic Light Simulation

Say you're building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with keys 'ns' and 'ew', for the stoplights facing north-south and east-

west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street.

To start the project, write a switchLights( ) function, which will take an intersection dictionary as an argument and switch the lights. At first, think that switchLights() should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'.

The code to implement this idea might look like this:

```
def switchLights(stoplight):
  for key in stoplight.keys( ):
    if stoplight[key] == 'green':
      stoplight[key] = 'yellow'
    elif stoplight[key] == 'yellow':
      stoplight[key] = 'red'
    elif stoplight[key] == 'red':
      stoplight[key] = 'green'
switchLights(market_2nd)
```

- Adding an assertion to check that *at least one of the lights is always red in* switchLights( ) , add the following at the bottom of the function:

  assert 'red' in stoplight.values( ), 'Neither light is red! ' + str(stoplight)

  With this assertion in place, program would crash with this errormessage:

```
Traceback (most recent call last):
  File "carSim.py", line 14, in <module>
  switchLights(market_2nd)
  File "carSim.py", line 13, in switchLights
    assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

The important line here is the ***AssertionError***. While program crashing is not ideal, it immediately points out that a sanity check failed: Neither direction of traffic has a red light, By failing fast early in the program's execution, saving a lot of future debugging effort.

Disabling Assertions

- Assertions can be disabled by passing the **-O** option when running Python.
- This is good for when finished writing and testing program and don't want it to be slowed down by performing sanity checks
- Assertions are for development, not the final product. By the time program reaches to someone else to run, it should be free of bugs and not require the sanity checks.

Logging

Logging is a great way to understand what's happening in program and in what order its happening. Python's logging module makes it easy to create a record of custom messages.

These log messages will describe when the program execution has reached the logging function call and list any variables that have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the #! python shebang line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
```

- When Python logs an event, it creates a LogRecord object that holds information about that event. The logging module's basicConfig( ) function specifies what details about the LogRecord object to see and how those details are displayed.
- Example: function to calculate the *factorial* of a number. In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24.

- It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as *factorialLog.py*.

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')
logging.debug('Start of program')
def factorial(n):
    logging.debug('Start of factorial(%s%%)' % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)' % (n))
    return total
print(factorial(5))
logging.debug('End of program')
```

- logging.debug( ) function is used to print log information. This debug( ) function will call basicConfig( ), and a line of information will be printed. This information will be in the format specified in basicConfig( ) and will include the messages passed to debug( ).
- Theprint(factorial(5)) call is part of the original program, so the result is displayed even if logging messages are disabled. The output of this program looks like this:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial (5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

The factorial( ) function is returning 0 as the factorial of 5, which isn't right. The for loop should be multiplying the value in total by the numbers from 1 to 5. But the log messages displayed by logging.debug () show that the i variable is starting at 0 instead of 1. Since zero times anything is zero, the rest of the iterations also have the wrong value for total.

Change the for i in range(n + 1): line to for i in range(**1,** n + 1):, and run the program again. The output will look like this:

2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program

The factorial(5) call correctly returns 120. The log messages showed what was going on inside the loop, which led straight to the bug.

- logging.debug( ) calls printed out not just the strings passed to them but also a timestamp and the word *DEBUG*.

Don't Debug with print( )

- Typing import logging and logging.basicConfig(level=logging.DEBUG, format= '%(asctime)s - %(levelname)s - %(message)s') is somewhat unwieldy.
- If print( ) calls are used instead, programmer end up spending a lot of time removing print( ) calls from code for each log message. Which may lead accidentally to remove some print( ) calls that were being used for nonlog messages.
- The nice thing about log messages is that you're free to fill your program with as many as you like, and you can always disable them later by adding a single ***logging.disable(logging.CRITICAL)*** call.
- Log messages are intended for the programmer, not the user.

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Table 10-1: Logging Levels in Python

Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to programmer to decide which category log message falls into

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

- Passing logging.DEBUG to the basicConfig() function's level keyword argument will show messages from all the logging levels (DEBUG being the lowest level).
- But after developing program some more, basicConfig()'s level argument can be set to logging.ERROR which prioritizes to errors. This will show only ERROR CRITICAL messages and skip the DEBUG, INFO, and WARNING messages.

Disabling Logging

After debugging, log messages are no needed. The logging.disable( ) function disables these without having to remove all the logging calls by hand.

By passing logging.disable( ) a logging level, suppresses all log messages at that level or lower. So, logging.disable(logging.CRITICAL) to program disables logging entirely.

For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
    %(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!

>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since logging.disable() will disable all messages after it, you will probably want to add it near the import logging line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Logging to a File

Instead of displaying the log messages to the screen, It can be written them to a text file. The logging.basicConfig( ) function takes a filename keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

The log messages will be saved to *myProgramLog.txt*. Later this text file can be opened in any text editor, such as Notepad or TextEdit.

IDLE 's Debugger

- The *debugger* is a feature of IDLE that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for user to tell it to continue.
- By running program "under the debugger" like this, values in the variables can be examined at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.
- To enable IDLE's debugger, click **Debug4Debugger** in the interactive shell window. This will bring up the Debug Control window, which looks like Figure 10-1. When the Debug Control window appears, select all four of the **Stack**, **Locals**, **Source**, and **Globals** checkboxes so that the window shows the full set of debug information. While the Debug Control window is displayed any time you run a program from the file editor, the debugger will pause execution before the first instruction and display the following:
  - The line of code that is about to be executed
  - A list of all local variables and their values
  - A list of all global variables and their values

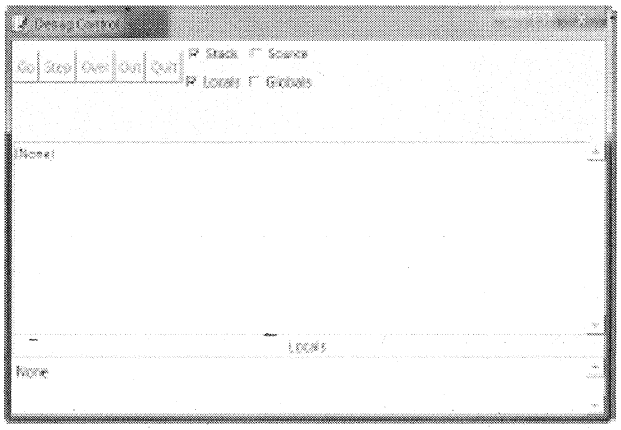


Figure 10-1: The Debug Control window

In the list of global variables there are several variables haven't defined, such as __builtins__, __doc__, __file__, and so on. These are variables that Python automatically sets whenever it runs a program. The program will stay paused until you press one of the five buttons in the Debug Control window: ***Go, Step, Over, Out, or Quit***. Clicking the

Go: the program to execute normally until it terminates or reaches If bugging is done and want the program to continue normally, click the **Go** button.

Step: the debugger executes the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over: executes the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns.

For example, if the next line of code is a print( ) call, you don't really care about code inside the built-in print( ) function; you just want the string you pass it printed to the screen. For this reason, using the Over button is more common than the Step button.

Out: will cause the debugger to execute lines of code at full speed until it returns from the current function. If user have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the **Out** button to "step out" of the current function call.

Quit: stops debugging entirely and not bother to continue executing the rest of the program. The Quit button will immediately terminate the program. If you want to run your program normally again, select **Debug4Debugger** again to disable the debugger.

Debugging a Number Adding Program

```
print('Enter the first number to add:')
first = input( )
print('Enter the second number to add:')
second = input( )
print('Enter the third number to add:')
third = input( )
print('The sum is ' + first + second + third)
```

Save it as buggyAddingProgram.py and run it first without the debuggerenabled.

The program will output something like this:

Enter the first number to add:

5

Enter the second number to add:

3

Enter the third number to add:

42

The sum is 5342

The program hasn't crashed, but the sum is obviously wrong. Let's enable the Debug Control window and run it again, this time under the debugger.

When you press F5 or select **Run→ Run Module** (with **Debug4Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1. The debugger will always pause on the line of code it is about to execute. The Debug Control window will look like Figure 10-2:

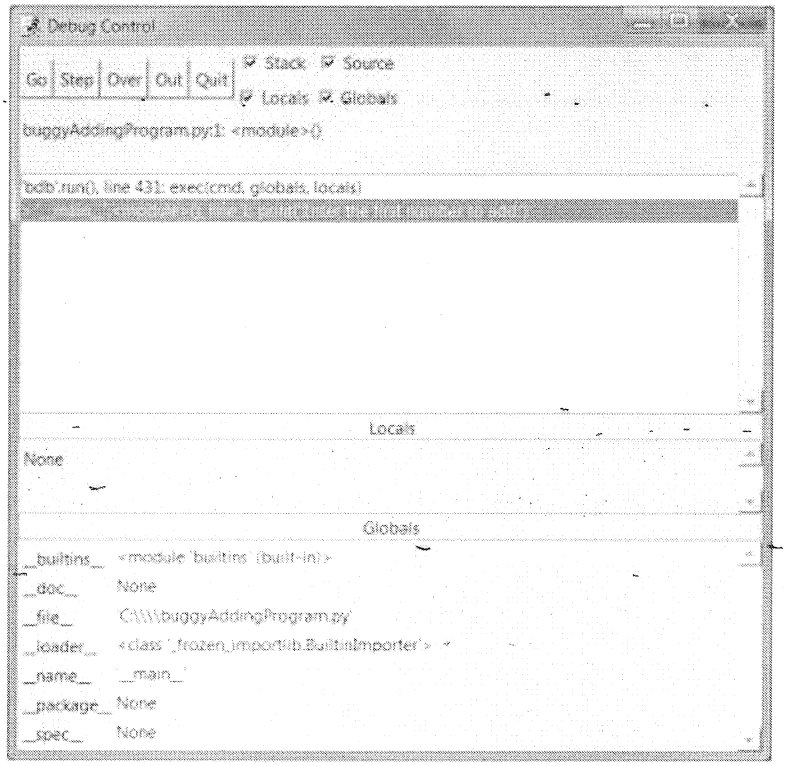


Figure 10-2: The Debug Control window when the program first starts under the debugger

Click the **Over** button once to execute the first print( ) call. use Over instead of Step here, since you don't want to step into the code for the print( ) function. The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted, as shown in Figure 10-3.

This shows you where the program execution currently is.

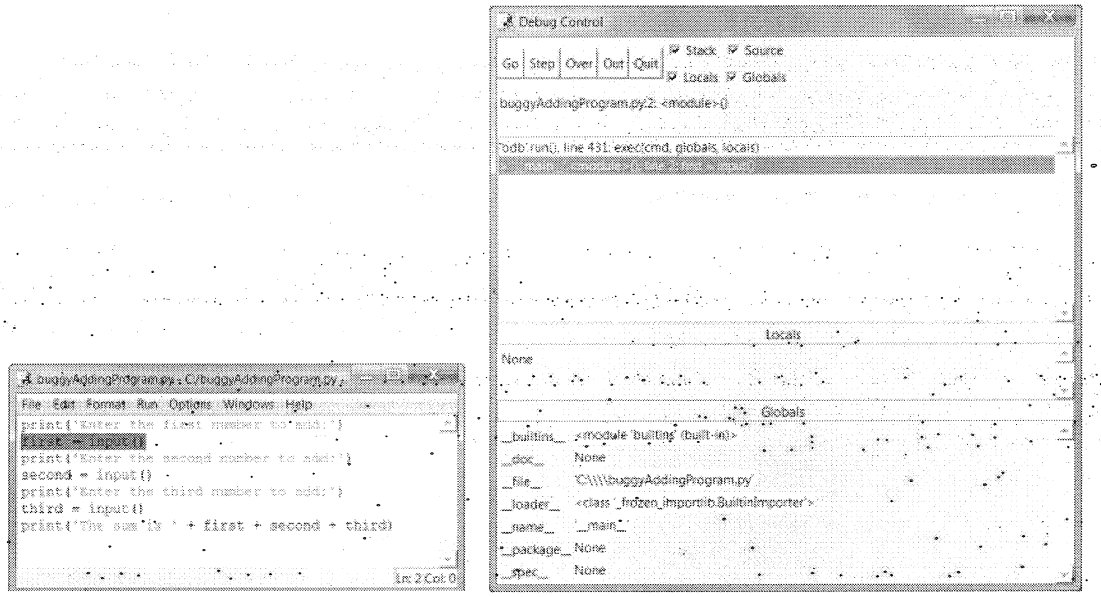


Figure 10-3: The Debug Control window after clicking Over

Click **Over** again to execute the input( ) function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the input( ) call into the interactive shell window.

Enter **5** and press Return. The Debug Control window buttons will be reenabled. Keep clicking **Over**, entering **3** and **42** as the next two numbers, until the debugger is on line 7, the final print( ) call in the program. The Debug Control window should look like Figure 10-4. You can see in the Globals section that the first, second, and third variables are set to string values '5', '3', and '42' instead of integer values 5, 3, and 42. When the last line is executed, these strings are concatenated instead of added together, causingthe bug.

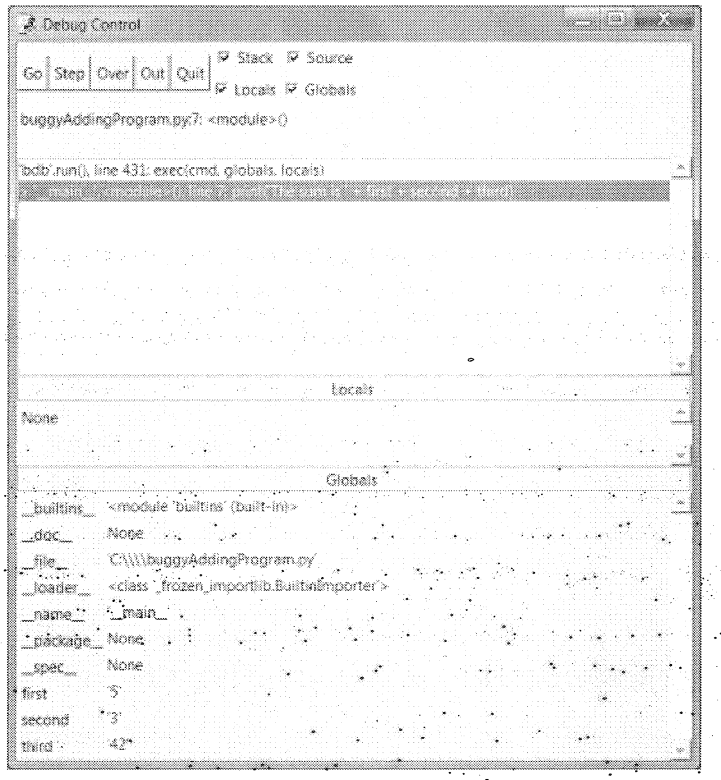


Figure 10-4: The Debug Control window on the last line. The variables are set to strings, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Breakpoints

A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times. Save it as *coinFlip.py*.

```
import random
heads = 0
for i in range(1, 1001):
    if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
        print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The random.randint(0, 1) call will return 0 half of the time and 1 the other half of the time. This can be used to simulate a 50/50 coin flip where 1 represents heads. When you run this program without the debugger, it quickly outputs something like the following:

Halfway done!

Heads came up 490 times.

If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated. If you were interested in the value of heads at the halfway point of the program's execution, when 500 of 1000 coin flips have been completed, you could breakpoint, right-click the line in the file editor and select **Set Breakpoint**, as shown in Figure 10-5.

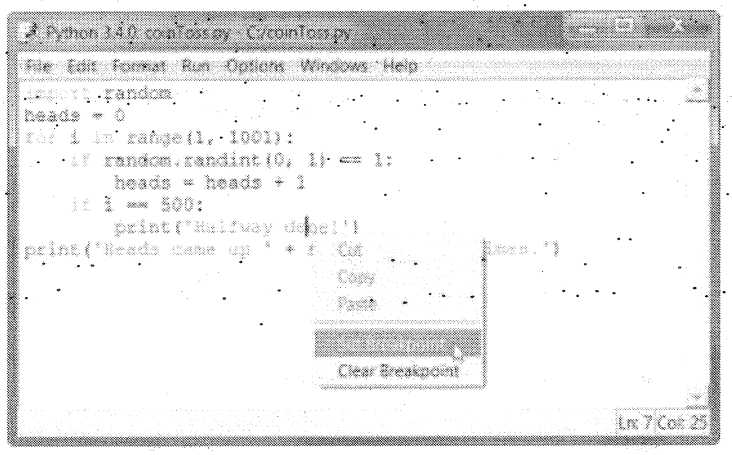


Figure 10-5: Setting a breakpoint

You don't want to set a breakpoint on the if statement line, since the if statement is executed on every single iteration through the loop. By setting the breakpoint on the code in the if statement, the debugger breaks only when the execution enters the if clause. The line with the breakpoint will be highlighted in yellow in the file editor. When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Go, the program will run at full speed until it reaches the line with the breakpoint set on it. You can then click Go, Over, Step, or Out to continue as normal.

If you want to remove a breakpoint, right-click the line in the file editor and select **Clear Breakpoint** from the menu. The yellow highlighting will go away, and the debugger will not break on that line in the future.

PYTHON APPLICATION PROGRAMMING (18CS55)

SEMESTER – V

LECTURER NOTES

Module 4

Classes and objects, Classes and functions, Classes and methods

Chapter 15: Classes and objects

object-oriented programming", which uses programmer defined types to organize both code and data.

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects. **Object** is simply a collection of data (variables) and methods (functions) that act on those data. And, **class** is a blueprint for the object. We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called **an instance of a class** and the process of creating this object is called **instantiation.**

Like function definitions begin with the keyword def, in Python, we define **a class** using the **keyword class**. The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.

Here is a simple class definition.

```
class MyNewClass:
'''This is a docstring. I have created a new class'''
pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class MyClass:
"This is my second class"
```

```
a = 10
def func(self):
print('Hello')
# Output: 10
print(MyClass.a)
# Output: <function MyClass.func at 0x0000000003079BF8>
print(MyClass.func)
# Output: 'This is my second class'
print(MyClass.__doc__)
```

The procedure to create an object is similar to a function call.

```
>>> ob = MyClass()
```

This will create a new instance object named ob. We can access attributes of objects using the object name prefix.

Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class.

The simplest form of class definition looks like this:

```
class ClassName:
<statement-1>
.
.
.
<statement-N>
```

A very basic class would look something like this:

```
class MyClass:
variable = "blah"
def function(self):
print("This is a message inside the class.")
```

First, to assign the above class(template) to an object you would do the following:

```
myobjectx = MyClass()
```

To access the variable inside of the newly created object "myobjectx" you would do the following:

```
Myobjectx.variable
```

Class objects support two kinds of operations: **attribute references and instantiation**. **Attribute references** use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class"s namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
```

```
"""A simple example class"""
i = 12345
def f(self):
return 'hello world'
```

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. __doc__ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

The **instantiation operation** ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named __init__(), like this:

```
def __init__(self):
self.data = []
```

When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance.

```
>>> class Complex:
... def __init__(self, realpart, imagpart):
... self.r = realpart
... self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning. Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
def __init__(self,r = 0,i = 0):
self.real = r
self.imag = i
def getData(self):
print("{0}+{1}j".format(self.real,self.imag))
# Create a new ComplexNumber object
```

```
c1 = ComplexNumber(2,3)
# Call getData() function
# Output: 2+3j
c1.getData()
# Create another ComplexNumber object
# and create a new attribute 'attr'
c2 = ComplexNumber(5)
c2.attr = 10
# Output: (5, 0, 10)
print((c2.real, c2.imag, c2.attr))
# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
c1.attr
```

15.1 Programmer-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called Point that represents a point in two-dimensional space. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this:

```
class Point:"""Represents a point in 2-D space."""
```

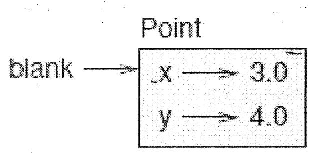


Figure 15.1: **Object diagram.**

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named Point creates a **class object**.

```
>>> Point
<class '__main__.Point'>
```

Because Point is defined at the top level, its "full name" is __main__.Point.

The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to blank. Creating a new object is called **instantiation**, and the object is an **instance** of the class. When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal). Every object is an instance of some class, so "object" and "instance" are interchangeable. But in this chapter I use "instance" to indicate that I am talking about a programmer defined type.

15.2 Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**. As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**; see Figure 15.1.

The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number. You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
print('(%g, %g)' % (p.x, p.y))
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

15.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal. There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example. Here is the class definition:

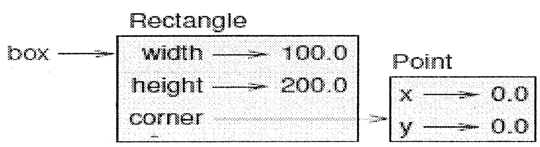


Figure 15.2: **Object diagram.**

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
```

```
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

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." Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4 Instances as return values

Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
```

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, grow_rectangle takes a Rectangle object and two numbers, dwidth and dheight, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
rect.width += dwidth
rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, rect is an alias for box, so when the function modifies rect, box changes.

15.6 Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The **copy module** contains a function called copy that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
>>> import copy
>>> p2 = copy.copy(p1)
```

p1 and p2 contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

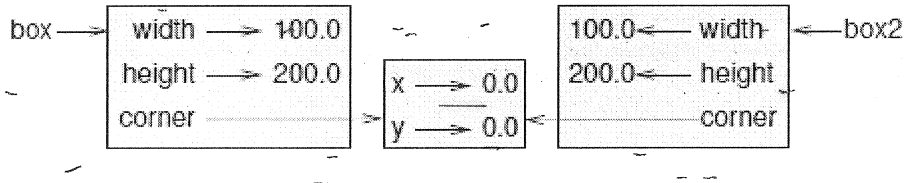


Figure 15.3: **Object diagram**.

The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined

types, Python doesn't know what should be considered equivalent. At least, not yet. If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 and box are completely separate objects.

Chapter 16

Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results.

16.1 Time

As another example of a programmer-defined type, we'll define a class called Time that records the time of day. The class definition looks like this:

```
class Time:
"""Represents the time of day.
attributes: hour, minute, second
"""
```

We can create a new Time object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

```
def print_time(time)
        print( str(time.hour)+":"+str(time.minute)+":" str(time.second))
```

The state diagram for the Time object looks like Figure 16.1.

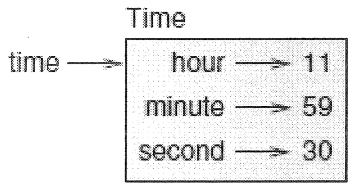


Figure 16.1: **Object diagram.**

16.2 Pure functions

Two kinds of functions exist: pure functions and modifiers.

A function that does not modify any of the objects it receives as arguments are called **pure functions**. Most pure functions are fruitful.

Here is a simple prototype of add_time:

```
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
return sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: start contains the start time of a movie, like Monty Python and the Holy Grail, and duration contains the run time of the movie, which is one hour 35 minutes. add_time figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>> duration = Time()
>>> duration.hour = 1
```

```
>>> duration.minute = 35
>>> duration.second = 0
>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, 10:80:00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column. Here's an improved version:

16.3. Modifiers

A function that changes one or more of the objects it receives as arguments are called **Modifiers.** Most modifiers are void; that is, they return None.

```
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
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later. Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

increment, which adds a given number of seconds to a Time object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
time.second += seconds
if time.second >= 60:
time.second -= 60
time.minute += 1
```

```
if time.minute >= 60:
time.minute -= 60
time.hour += 1
```

In general, it is recommended to write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

16.4 Prototyping versus planning

Prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

Designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development. Modifiers are void; that is, they return none.

Functional programming style: A style of program design in which the majority of functions are pure.

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way. This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60. The second attribute is the "ones column", the minute attribute is the "sixties column", and the hour attribute is the "thirty-six hundreds column". When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
minutes = time.hour * 60 + time.minute
seconds = minutes * 60 + time.second
return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
time = Time()
minutes, time.second = divmod(seconds, 60)
time.hour, time.minute = divmod(minutes, 60)
return time
```

Once you are convinced they are correct, you can use them to rewrite add_time:

```
def add_time(t1, t2):
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
```

Chapter 17

Classes and methods

The next step is to transform those functions into methods that make the relationships explicit.

17.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining 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 things in the real world interact.

A **method** is a function that is associated with a particular class. **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.

17.2 Printing objects

A function is written to print time objects

```
named print_time:
class Time:
"""Represents the time of day."""
def print_time(time):
print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Time object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make print_time a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
def print_time(time):
print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call print_time. The first (and less common) way is to use **function syntax:**

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

The second (and more concise) way is to use **method syntax:**

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.

By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
def print_time(self):
print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

• The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, "Hey print_time! Here's an object for you to print."

• In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says "Hey start! Please print yourself."

17.3 Another example

Here's a version of increment (from Section 16.3) rewritten as a method:

```
# inside class Time:
def increment(self, seconds):
seconds += self.time_to_int()
return int_to_time(seconds)
```

This version assumes that time_to_int is written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

17.4 A more complicated example

Rewriting is_after (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameter other:

```
# inside class Time:
def is_after(self, other):
return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument

```
>>> end.is_after(start)
True
```

17.5 The init method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is __init__ (two underscore characters, followed by init, and then two more underscores). An init method for the Time class might look like this:

```
# inside class Time:
def __init__(self, hour=0, minute=0, second=0):
self.hour = hour
self.minute = minute
self.second = second
```

It is common for the parameters of __init__ to have the same names as the attributes. The statement self.hour = hour

17.6. The __str__ method stores the value of the parameter hour as an attribute of self. The parameters are optional, so if you call Time with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

For example, here is a str method for Time objects:

```
# inside class Time:
def __str__(self):
return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

17.7 Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects. Here is what the definition might look like:

```
# inside class Time:
def __add__(self, other):
seconds = self.time_to_int() + other.time_to_int()
return int_to_time(seconds)
And here is how you could use it:
>>> start = Time(9, 45)
```

```
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__. Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**.

17.8 Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment:

```
# inside class Time:
def __add__(self, other):
if isinstance(other, Time):
return self.add_time(other)
else:
return self.increment(other)
def add_time(self, other):
seconds = self.time_to_int() + other.time_to_int()
return int_to_time(seconds)
def increment(self, seconds):
seconds += self.time_to_int()
return int_to_time(seconds)
```

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class. If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments. Here are examples that use the + operator with different types:

17.9. Polymorphism

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the

first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method __radd__, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:
def __radd__(self, other):
return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types. Many of the functions we wrote for strings also work for other sequence types.

```
def histogram(s):
d = dict()
for c in s:
if c not in d:
d[c] = 1
else:
d[c] = d[c]+1
return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of are hashable, so they can be used as keys in d.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition. Since Time objects provide an add method, they work with sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
```

```
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type. The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

17.10 What is Inheritance?

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Python Inheritance Syntax

```
class BaseClass:
Body of base class
class DerivedClass(BaseClass):
Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Example1 of Inheritance in Python

To demonstrate the use of inheritance, let us take an example. A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
def __init__(self, no_of_sides):
self.n = no_of_sides
self.sides = [0 for i in range(no_of_sides)]
def inputSides(self):
self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]
def dispSides(self):
for i in range(self.n):
print("Side",i+1,"is",self.sides[i])
```

A triangle is a polygon with 3 sides. So, we can created a class called Triangle which inherits from Polygon. This makes all the attributes available in class Polygon readily available in Triangle. We don't need to define them again (code re-usability). Triangle is defined as follows.

```
class Triangle(Polygon):
def __init__(self):
Polygon.__init__(self,3)
```

```
def findArea(self):
    a, b, c = self.sides
    # calculate the semi-perimeter
    s = (a + b + c) / 2
    area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
    print('The area of the triangle is %0.2f' %area)
```

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()
>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4
>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0
>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.

Example2 of Inheritance in Python

```
class Parent:        # define parent class
  parentAttr = 100
  def __init__(self):
    print "Calling parent constructor"
  def parentMethod(self):
    print 'Calling parent method'
  def setAttr(self, attr):
    Parent.parentAttr = attr
  def getAttr(self):
    print "Parent attribute :", Parent.parentAttr
```

```
class Child(Parent): # define child class
   def __init__(self):
      print "Calling child constructor"
   def childMethod(self):
      print 'Calling child method'
c = Child()          # instance of child
c.childMethod()      # child calls its method
c.parentMethod()     # calls parent's method
c.setAttr(200)       # again call parent's method
c.getAttr()          # again call parent's method
```

When the above code is executed, it produces the following result –

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Similar way, you can drive a class from multiple parent classes as follows –

```
class A:        # define your class A
.....
class B:        # define your class B
.....
class C(A, B):  # subclass of A and B
.....
```

You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class