APPLICATION DEVELOPMENT USING PYTHON

[(Effective from the academic year 2018 -2019)]

SEMESTER – V	**Course Code 18CS55**	IA Marks 40
Number of Lecture Hours/Week 03		Exam Marks 60
Total Number of Lecture Hours 40	Exam Hours 03	CREDITS – 03

Course Learning Objectives:This course (18CS55) will enable students to

- Learn the syntax and semantics of Python programming language.
- Illustrate the process of structuring the data using lists, tuples and dictionaries.
- Demonstrate the use of built-in functions to navigate the file system.
- Implement the Object Oriented Programming concepts in Python.
- Appraise the need for working with various documents like Excel, PDF, Word and Others.

Module – 1 Teaching Hours 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, def 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

Textbook 1: Chapters 1 – 3 RBT: L1, L2 08

Module – 2 Lists, The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, 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

Textbook 1: Chapters 4 – 6 RBT: L1, L2, L3 08

Module – 3 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.

Textbook 1: Chapters 7 – 10 08 RBT: L1, L2, L3

Module – 4 Classes and objects, Programmer-defined types, Attributes, Rectangles, Instances as return values, Objects are mutable, Copying, Classes and functions, Time, Pure functions, Modifiers, Prototyping versus planning, Classes and methods, Object-oriented features, Printing objects, Another example, A more complicated example,The init method, The __str__ method, Operator overloading, Type-based dispatch, Polymorphism, Interface and implementation,Inheritance, Card objects, Class attributes, Comparing cards, Decks, Printing the deck, Add, remove, shuffle and sort, Inheritance, Class diagrams, Data encapsulation

Textbook 2: Chapters 15 – 18 RBT: L1, L2, L3 08

Module – 5 Web Scraping, Project: MAPIT.PY with the webbrowser Module, Downloading Files from the Web with the requests Module, Saving Downloaded Files to the Hard Drive, HTML, Parsing HTML with the BeautifulSoup Module, Project: "I'm Feeling Lucky" Google Search,Project: Downloading All XKCD Comics, Controlling the Browser with the selenium Module, Working with Excel Spreadsheets, Excel Documents, Installing the openpyxl Module, Reading Excel Documents, Project: Reading Data from a Spreadsheet, Writing Excel Documents, Project: Updating a Spreadsheet, Setting the Font Style of Cells, Font Objects, Formulas, Adjusting Rows and Columns, Charts, Working with PDF and Word Documents, PDF Documents, Project: Combining Select Pages from Many PDFs, Word Documents, Working with CSV files and JSON data, The csv Module, Project: Removing the Header from CSV Files, JSON and APIs, The json Module, Project: Fetching Current Weather Data

Textbook 1: Chapters 11 – 14 RBT: L1, L2, L3 08

Course Outcomes: After studying this course, students will be able to

- Demonstrate proficiency in handling of loops and creation of functions.
- Identify the methods to create and manipulate lists, tuples and dictionaries.
- Discover the commonly used operations involving regular expressions and file system.
- Interpret the concepts of Object-Oriented Programming as used in Python.
- Determine the need for scraping websites and working with CSV, JSON and other file formats.

Question paper pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Text Books:

1. Al Sweigart,"Automate the Boring Stuff with Python",1 stEdition, No Starch Press, 2015. (Available under CC-BY-NC-SA license at https://automatetheboringstuff.com/) (Chapters 1 to 18)

2. Allen B. Downey, "Think Python: How to Think Like a Computer Scientist", 2nd Edition, Green Tea Press, 2015. (Available under CC-BY-NC license at http://greenteapress.com/thinkpython2/thinkpython2.pdf) (Chapters 13, 15, 16, 17, 18) (Download pdf/html files from the above links)

Reference Books:

1. Gowrishankar S, Veena A, "Introduction to Python Programming", 1st Edition, CRC Press/Taylor & Francis, 2018. ISBN-13: 978-0815394372

2. Jake VanderPlas, "Python Data Science Handbook: Essential Tools for Working with Data", 1 st Edition, O'Reilly Media, 2016. ISBN-13: 978-1491912058

3. Charles Dierbach, "Introduction to Computer Science Using Python", 1 st Edition, Wiley India Pvt Ltd, 2015. ISBN-13: 978-8126556014 4. Wesley J Chun, "Core Python Applications Programming", 3rd Edition, Pearson Education India, 2015. ISBN-13: 978-9332555365

APPLICATION DEVELOPMENT USING PYTHON (18CS55)

MODULE-1: Python Basics

What Is Python?

Python is a programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.

Entering Expressions into the Interactive Shell

You can run the interactive shell by launching the Mu editor, On Windows, open the Start menu, type "Mu," and open the Mu app.which will open as a new pane that opens at the bottom of the Mu editor's window.

You should see a >>> prompt in the interactive shell.

Enter 2 + 2 at the prompt to have Python do some simple math.

The Mu window should now look like this:

```
>>> 2 + 2
4
>>>
```


In Python, 2 + 2 is called an expression, which is the most basic kind of programming instruction in the language. Expressions consist of values (such as 2) and operators (such as +), and they can always evaluate (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

You can use plenty of other operators in Python expressions, too. For example, Table lists all the math operators in Python.

Table: Math Operators from Highest to Lowest Precedence

Operator	Operation	Example Evaluates to . . .
**	Exponent	2 ** 3 = 8
%	Modulus	22 % 8= 6
/	Division	22 / 8= 2.75
//	Integer division/floored quotient	22 // 8= 2
*	Multiplication	3 * 5= 15
-	Subtraction	5- 2= 3
+	Addition	2 + 2 4

Python will keep evaluating parts of the expression until it becomes a single value, as shown here:

```
(5 - 1) * ((7 + 1) / (3 - 1))
 4 * ((7 + 1) / (3 - 1))
 4 * (8 )  / (3 - 1)
 4 * (8 ) / (2 )
 4 * 4.0
```

16.0

The Integer, Floating-Point, and String Data Types:

A data type is a category for values, and every value belongs to exactly one data type.

The **integer (or int) data type** indicates values that are whole numbers.

Numbers with a decimal point, such as 3.14, are called **floating-point numbers (or floats).**

Python programs can also have text values called **strings**, or strs (pronounced "stirs").

Always surround your string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. You can even have a string with no characters in it, '', called a blank string or an empty string.

String Concatenation and Replication

+ is used on two string values, it joins the strings as the string concatenation operator.

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

when the * operator is used on one string value and one integer value, it becomes the **string replication operator.**

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```


Storing Values in Variables

A variable is like a box in the computer's memory where you can store a single value.

Assignment Statements

You'll store values in variables with an assignment statement. An assignment statement consists of a variable name, an equal sign (called the assignment operator), and the value to be stored.

```
>>> spam = 40
>>> spam
40
```

A variable is initialized (or created) the first time a value is stored in it.

Variable Names

Python does have some naming restrictions.

. You can name a variable anything as long as it obeys the following three rules:

• It can be only one word with no spaces.

• It can use only letters, numbers, and the underscore (_) character.

• It can't begin with a number.

Comments

The following line is called a comment.

This program says hello and asks for my name.

The print() Function

The print() function displays the string value inside its parentheses on the screen.

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

The input() Function

The input() function waits for the user to type some text on the keyboard and press enter.

Printing the User's Name

The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.

```
print('It is good to meet you, ' + myName)
myName = input()
```

The len() Function

You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
print('The length of your name is:')
print(len(myName))
```


The str(), int(), and float() Functions

If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29. The str() function can be passed an integer value and will evaluate to a string value version of the integer, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

The str(), int(), and float() functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively.

```
>>> str(0) '0'
>>> str(-3.14)
'-3.14' >>>
int('42')
42
>>> int('-99')
-99
>>> int(1.25)
```

```
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

FLOW CONTROL

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the Boolean data type has only two values: True and False.

When entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital T or F, with the rest of the word in lowercase.

```
>>> spam = True
>>> spam
True
```


Comparison Operators

Comparison operators, also called relational operators, compare two values and evaluate down to a single Boolean value.

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

```
>>> 42 == 42
True
>>> 'hello' == 'hello'
True
```

Boolean Operators

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value.

Binary Boolean Operators

The and and or operators always take two Boolean values (or expressions), so they're considered binary operators. The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False.

>>>True and True

True

On the other hand, 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.

>>> False or True

True

The not Operator

Unlike and and or, the not operator operates on only one Boolean value (or expression). This makes it a unary operator. The not operator simply evaluates to the opposite Boolean value.

>>> not True

False

>>> not not True

True

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

>>> (4 < 5) and (5 < 6)

True

Flow Control Statements

if Statements

The most common type of flow control statement is the if statement.

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

In Python, an if statement consists of the following:

• The if keyword •

A condition (that is, an expression that evaluates to True or False) Flow Control 29

• A colon

• Starting on the next line, an indented block of code (called the if clause)

For example, let's say you have some code that checks to see whether someone's name is Alice.

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

else Statements

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, and in code, an 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)

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

elif Statements

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 all of the previous conditions were False.

In code, 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)

Let's add an elif to the name checker to see this statement in action.

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


while Loop Statements

You can make a block of code execute over and over again using a while statement. The code in a while clause will be executed as long as the while statement's condition is True.

In code, a while statement always consists of the following:

• The while 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 while clause)

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

In the while loop, the condition is always checked at the start of each iteration (that is, each time the loop is executed). 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.

break Statements

There is a shortcut to getting the program execution to break out of a while loop's clause early. 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!')
```

continue Statements

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.

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


for Loops and the range() Function

if you want to execute a block of code only a certain number of times, You can do this with a for loop statement and the range() function.

In code, a for statement looks something like for i in range(5): and includes the following:

- The for keyword
- A variable name
- The 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

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

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

Some functions can be called with multiple arguments separated by a comma, and range() is one of them.

This lets you change the integer passed to range() 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 will be where the for loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
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 increased by after each iteration.

```
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.

```
0 2 4 6 8
```


Importing Modules

All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions you've seen before.

Python also comes with 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.

Before you can use the functions in a module, you must import the module with an import statement.

In code, an import statement consists of the following:

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

Once you import a module, you can use all the cool functions of that module.

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

from import Statements

An alternative form of the import statement is composed of the from keyword, followed by the module name, the import keyword, and a star; for example, from random import *. With this form of import statement, calls to functions in random will not need the random. prefix.

Ending a Program Early with the sys.exit() Function

Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, before the last instruction by calling the sys.exit() function.

Since this function is in the sys module, you have to import sys before your program can use it.

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

FUNCTIONS

A function is like a miniprogram within a program.

The first line is a def statement , which defines a function named hello().

The code in the block that follows the def statement ϖ is the body of the function. This code is executed when the function is called, not when the function is first defined.

In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses.

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.

```
def hello():
    print('Howdy!')
    print('Howdy!!!')
    print('Hello there.')
hello()
```

```
Howdy!
Howdy!!!
Hello there.
```

def Statements with Parameters

When you call the print() or len() function, you pass them values, called arguments, by typing them between the parentheses. You can also define your own functions that accept arguments.

```
def hello(name):
        print('Hello, ' + name)
hello('Alice')
hello('Bob')
```

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

```
Hello, Alice
Hello, Bob
```

Define, Call, Pass, Argument, Parameter

To define a function is to create it, just like an assignment statement like spam = 42 creates the spam variable.

The def statement defines the sayHello() function . The sayHello('Al') line calls the now-created function, sending the execution to the top of the function's code. This function call is also known as passing the string value 'Al' to the function. A value being passed to a function in a function call is an argument. The argument 'Al' is assigned to a local variable named name. Variables that have arguments assigned to them are parameters.

Return Values and return Statements

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:

- The return keyword
- The 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.

```
import random
def getAnswer(answerNumber):
        if answerNumber == 1:
                return 'It is certain'
        elif answerNumber == 2:
                return 'It is decidedly so'
        elif answerNumber == 3:
                return 'Yes'
        elif answerNumber == 4:
                return 'Reply hazy try again'
        elif answerNumber == 5:
                return 'Ask again later'
```

```
        elif answerNumber == 6:
                return 'Concentrate and ask again'
        elif answerNumber == 7:
                return 'My reply is no'
        elif answerNumber == 8:
                return 'Outlook not so good'
        elif answerNumber == 9:
                return 'Very doubtful'
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

The None Value

In Python, there is a value called None, which represents the absence of a value. The None value is the only value of the None Type data type. None must be typed with a capital N.

Keyword Arguments and the print() Function

Most arguments are identified by their position in the function call.

keyword 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.

If you ran a program with the following code:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two outputted 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 the newline character to a different string.

For example, if the code were this:

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output is printed on a single line because there is no longer a newline printed after 'Hello'. Instead, the blank string is printed. This is useful if you need 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.

Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But you could replace the default separating string by passing the sep keyword argument a different string.

Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

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.

There is only one **global scope**, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A **local scope** is created whenever a function is called. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call the function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in 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 different scopes. That is, there can be a local variable named spam and a global variable also named spam.

Local Variables Cannot Be Used in the Global Scope

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

```
def spam():
    eggs = 31337
spam()
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):
File "C:/test1.py", line 4, in print(eggs)
NameError: name 'eggs' is not defined
```

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.

Consider this program:

```
def spam():
    eggs = 99
    bacon()
    print(eggs)
def bacon():
    ham = 101
    eggs = 0
spam()
```

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Local and Global Variables with the Same Name Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes in Python.

```
def spam():
    eggs = 'spam local'
    print(eggs)   # prints 'spam local'
def bacon():
    eggs = 'bacon local'
    print(eggs) # prints 'bacon local'
    spam()
    print(eggs) # prints 'bacon local' ω
eggs = 'global'
bacon()
print(eggs) # prints 'global'
```

The global Statement

If you need to 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)
```

When you run this program, the final print() call will output this:

Spam

There are four rules to tell whether a variable is in a local scope or global scope:

• If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.

• If there is a global statement for that variable in a function, it is a global variable

. • Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.

• But if the variable is not used in an assignment statement, it is a global variable.

Exception Handling

Right now, getting an error, or exception, in your Python program means the entire program will crash. 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.

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 divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs

```
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 program is as follows:

21.0 3.5 Error: Invalid argument. None 42.0

UNIT-2: LISTS

LISTS: The List Data Type

A list is a value that contains multiple values in an ordered sequence.

The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value.

A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].

A list begins with an opening square bracket and ends with a closing square bracket, [].

Values inside the list are also called items. Items are separated with commas.

```
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The value [] is an empty list that contains no values.

Getting Individual Values in a List with Indexes

The integer inside the square brackets that follows the list is called an index. The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
```


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

While indexes start at 0 and go up, you can also use negative integers 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 a List from Another List with Slices

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. Notice the difference between indexes and slices.

• 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. A slice evaluates to a new list value.

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

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

```
>>> 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 the len() Function

The len() function will return the number of values that are in a list value passed to it.

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

Changing Values in a List with Indexes

you can also use an index of a list to change the value at that index.

```
>>> spam[1] = 'aardvark'
 >>> spam
['cat', 'aardvark', 'rat', 'elephant']
```

List Concatenation and List Replication

Lists can be concatenated and replicated just like strings.

The + operator combines two lists to create a new list value and

the * operator can be used with a list and an integer value to replicate the list.

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

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.

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

Using for Loops with Lists

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

The previous for loop actually loops 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.

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
 >>> for i in range(len(supplies)):
            ... print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
```

Index 2 in supplies is: flamethrowers

Index 3 in supplies is: binders

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.

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

EXAMPLE 2:

```
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 output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```


Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself.

For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator += to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam 43
```

There are augmented assignment operators for the +, -, *, /, and % operators.

Methods

A method is the same thing as a function, except it is "called on" a value.

For example, if a list value were stored in spam, you would call the index() list method on that list like so: spam.index('hello').

The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index()

Method List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a ValueError error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>>
spam.index('heyas')
3
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the append() and insert() methods.

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

The previous append() method call adds the argument to the end of the list.

The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Removing Values from Lists with the remove() Method

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']
```

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.

Example1:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
 [-7, 1, 2, 3.14, 5]
```

Example2:

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
 >>> spam.sort()
 >>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also 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']
```

Reversing the Values in a List with the reverse() Method

If you need to quickly reverse the order of the items in a list, you can call the reverse() list method.

```
>>> spam = ['cat', 'dog', 'moose']
 >>> spam.reverse()
>>> spam
 ['moose', 'dog', 'cat']
```


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)])
```

Mutable and Immutable Data Types

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.

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']

DICTIONARIES AND STRUCTURING DATA

The Dictionary Data Type Like a list, a dictionary is a mutable collection of many values.

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:

>>> myCat['size']

'fat'

>>> 'My cat has ' + myCat['color'] + ' fur.'

'My cat has gray fur.'

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at 0 and can be any number.

>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}

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_keys, dict_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

Here, a for loop iterates over each of the values in the spam dictionary. 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']
```

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

You 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
```

The get() Method

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.'
```

The setdefault() Method You'll often 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 is a nice shortcut to ensure that a key exists.

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'}
```

Example: Here is a 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)
```

{' ': 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}

Pretty Printing

If you import the pprint module into your programs, you'll have access to the pprint() and pformat() functions that will "pretty print" a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what print() provides.

```
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)
```

when the program is run, the output looks much cleaner, with the keys sorted.

```
 {' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, --snip-- '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

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

```
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):
  printBoard(theBoard)
  print('Turn for ' + turn + '. Move on which space?')
  move = input()
  theBoard[move] = turn
 if turn == 'X':
    turn = 'O'
else:
    turn = 'X'
printBoard(theBoard)
```


MANIPUL ATING STRINGS

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: they begin and end with a single quote.

```
>>>spam='That is Alice'
```

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."
```

if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

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.'
```

Raw Strings

You can 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.
```

Multiline Strings with Triple Quotes

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.

```
print('''Dear Alice,
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
Sincerely,
Bob''')
```

The output will look like this:

```
Dear Alice,
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
Sincerely,
Bob
```

Multiline Comments

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.
"""
def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

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 12
```

The space and exclamation point are included in the character count, so 'Hello, world!' is 13 characters long, from H at index 0 to ! at index 12.

```
>>> spam = 'Hello, world!'
>>> spam[0
] 'H'
 >>> spam[4]
 'o'
>>> spam[-1]
 '!'
>>> spam[0:5]
 'Hello'
>>> spam[:5]
 'Hello'
>>> spam[7:]
 '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
```

Putting Strings Inside Other Strings

string interpolation, in which the %s operator inside the string acts as a marker to be replaced by values following the string. One benefit of string interpolation is that str() doesn't have to be called to convert values to strings.

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' % (name, age)
'My name is Al. I am 4000 years old.'
```

f-strings, which is similar to string interpolation except that braces are used instead of %s, with the expressions placed directly inside the braces.

Like raw strings, f-strings have an f prefix before the starting quotation mark.

```
>>> name = 'Al'
>>> age = 4000
>>> f'My name is {name}. Next year I will be {age + 1}.'
'My name is Al. Next year I will be 4001.'
```

Useful String Methods

The upper(), lower(), isupper(), and islower() 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.

Nonletter characters in the string remain unchanged.

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam 'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

The isX() Methods

isalpha() Returns True if the string consists only of letters and isn't 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

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

For example, the following program repeatedly asks users for their age and a password until they provide valid input.

```
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.')
```

The startswith() and endswith() 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() 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 string, gets passed a list of strings, and returns a string.

The returned string Manipulating Strings is the concatenation of each string in the passed-in list.

For example

```
>>> ', '.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']
```

Splitting Strings with the partition() Method

The partition() string method can split a string into the text before and after a separator string.

This method searches the string it is called on for the separator string it is passed, and returns a tuple of three substrings for the "before," "separator," and "after" substrings.

```
>>> 'Hello, world!'.partition('w')
('Hello, ', 'w', 'orld!')
```

```
>>> 'Hello, world!'.partition('world')
('Hello, ', 'world', '!')
```

If the separator string you pass to partition() occurs multiple times in the string that partition() calls on, the method splits the string only on the first occurrence:

```
>>> 'Hello, world!'.partition('o')
('Hell', 'o', ', world!')
```

If the separator string can't be found, the first string returned in the tuple will be the entire string, and the other two strings will be empty:

```
>>> 'Hello, world!'.partition('XYZ')
('Hello, world!', '', '')
```

You can use the multiple assignment trick to assign the three returned strings to three variables:

```
>>> before, sep, after = 'Hello, world!'.partition(' ')
>>> before
'Hello,'
>>> after
'world!'
```

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

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     '
```

'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

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========'
```

Removing Whitespace with the strip(), rstrip(), and lstrip() Methods The strip() string method will return a new string without any whitespace 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'
```


Numeric Values of Characters with the ord() and chr() Functions

Every text character has a corresponding numeric value called a Unicode code point.

For example, the numeric code point is 65 for 'A', 52 for '4', and 33 for '!'.

You can use **the ord() function** to get the code point of a one-character string, and the **chr() function** to get the one-character string of an integer code point.

```
>>> ord('A')
 65
 >>> ord('4')
52
 >>> ord('!')
 33
>>> chr(65)
'A'
```

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 into an email, word processor, or some other software.

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

```
>>> pyperclip.paste()
'Hello, world!'
```

Project: Multi-Clipboard Automatic Messages

If you've responded to a large number of emails with similar phrasing, you've probably had to do a lot of repetitive typing. Maybe you keep a text document with these phrases so you can easily copy and paste them using the clipboard. But your clipboard can only store one message at a time, which isn't very convenient. Let's make this process a bit easier with a program that stores multiple phrases.

Step 1: Program Design and Data Structures You want to be able to run this program with a command line argument that is a short key phrase—for instance, agree or busy. The message associated with that key phrase will be copied to the clipboard so that the user can paste it into an email. This way, the user can have long, detailed messages without having to retype them.

Open a new file editor window and save the program as mclip.py. You need to start the program with a #! (shebang) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each piece of text with its key phrase, you can store these as strings in a dictionary. The dictionary will be the data structure that organizes your key phrases and text.

```
#! python3
# mclip.py - A multi-clipboard program.
TEXT = {'agree': """Yes, I agree. That sounds fine to me.""",
        'busy': """Sorry, can we do this later this week or next week?""",
        'upsell': """Would you consider making this a monthly donation?"""}
```


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 ('mclip.py'), and the second item should be the first command line argument.

```
#! python3
# mclip.py - A multi-clipboard program.
TEXT = {'agree': """Yes, I agree. That sounds fine to me.""", 'busy': """Sorry, can we do this later this week or next week?""", 'upsell': """Would you consider making this a monthly donation?"""}
import sys
if len(sys.argv) < 2:
    print('Usage: python mclip.py [keyphrase] - copy phrase text')
    sys.exit()
keyphrase = sys.argv[1] # first command line arg is the keyphrase
```

Step 3: Copy the Right Phrase

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

```
#! python3
```

```
# mclip.py - A multi-clipboard program.
TEXT = {'agree': """Yes, I agree. That sounds fine to me.""", 'busy': """Sorry, can we do this later this week or next week?""", 'upsell': """Would you consider making this a monthly donation?"""}
import sys, pyperclip
 if len(sys.argv) < 2:
    print('Usage: py mclip.py [keyphrase] - copy phrase text')
    sys.exit()
keyphrase = sys.argv[1] # first command line arg is the keyphrase
 if keyphrase in TEXT:
     pyperclip.copy(TEXT[keyphrase])
     print('Text for ' + keyphrase + ' copied to clipboard.')
 else:
       print('There is no text for ' + keyphrase)
```

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.


```
 #! 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)
```

UNIT-3: Pattern Matching with Regular Expressions

Finding Patterns of Text Without Regular Expressions

Say you want to find an American phone number in a string. You know the pattern if you're American: 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):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
for i in range(8, 12):
    if not text[i].isdecimal():
        return False
return True
print('Is 415-555-4242 a phone number?')
print(isPhoneNumber('415-555-4242'))
print('Is Moshi moshi a phone number?')
print(isPhoneNumber('Moshi moshi'))
```

ouptput:

```
Is 415-555-4242 a phone number?
True
Is Moshi moshi a phone number?
False
```

If you wanted to find a phone number within a larger string, you would have to add even more code to find the phone number pattern. 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)):
```

```
    chunk = message[i:i+12]
    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
```

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 from 0 to 9.

The regex \d\d\d-\d\d\d-\d\d\d\d is used by Python to match the same text pattern the previous isPhoneNumber() function did: 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 braces ({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.

Creating Regex Objects All the regex functions in Python are in the re module.

Passing a string value representing your regular expression to re.compile() returns a Regex pattern 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 a phone number pattern.

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

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, which 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
```

More Pattern Matching with Regular Expressions

Say you want 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 you can use the group() match object method 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, you can grab different parts of the matched text. 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'
```

Since mo.groups() returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable

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


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
```

You can also use the pipe to match one of several patterns as part of your regex.

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

```
 'Batmobile'
>>> mo.group(1)
 'mobile'
```

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match regardless of whether that bit of text is there. The ? character flags the group that precedes it as an optional part of the pattern.

```
>>> 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'
```

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code.

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
 >>> mo1 = phoneRegex.search('My number is 415-555-4242')
 >>> mo1.group()
 '415-555-4242'
>>> mo2 = phoneRegex.search('My number is 555-4242')
 >>> mo2.group()
 '555-4242'
```

You can think of the ? as saying, “Match zero or one of the group preceding this question mark.”

If you need to match an actual question mark character, escape it with \?.

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 occur any number of times in the text. It 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'
```

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

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.

```
>>> 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
```

Matching Specific Repetitions with Braces

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in braces.

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.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the braces.

For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

You can also leave out the first or second number in the braces to leave the minimum or maximum unbounded.

For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances.

Braces can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((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
```

Greedy and Non-greedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible.

The nongreedy (also called lazy) version of the braces, which matches the shortest string possible, has the closing brace 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'
```

The findall() Method

In addition to the search() method, Regex objects also have a findall() method. the findall() method will return the strings of every match in the searched string.

findall() will not return a Match object but a list of strings—as long as there are no groups in the regular expression. 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', '9999'), ('212', '555', '0000')]
```

Character Classes

In the earlier phone number regex example, you learned that \d could stand for any numeric digit. That is, \d is shorthand for the regular expression (0|1|2|3|4|5|6|7|8|9).

There are many such shorthand 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

You can define your own character class 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']

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. **A negative character class** will match all the characters that are not in the character class.

>>> 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', '.']

The Caret and Dollar Sign Characters

You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.

Likewise, you can put a dollar sign ($) at the end of the regex to indicate the string must end with this regex pattern.

And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example,

the r'^Hello' regular expression string matches strings that begin with 'Hello'.

>>> beginsWithHello = re.compile(r'^Hello')

>>> beginsWithHello.search('Hello, world!')

>>> beginsWithHello.search('He said hello.') == None

True

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')

>>> endsWithNumber.search('Your number is forty two.') == None

True

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')
>>> 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']
```

Matching Everything with Dot-Star Sometimes you will want to match everything and anything.

For example, say you want to match the string 'First Name:', followed by any and all text, followed by 'Last Name:', and then followed by anything again.

You can use the dot-star (.*) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

```
>>> 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 non-greedy fashion, use the dot, star, and question mark (.*?).

Like with braces, the question mark tells Python to match in a non-greedy way.

the difference between the greedy and non-greedy 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 '<To serve man> for dinner.>' has two possible matches for the closing angle bracket. In the non-greedy version of the regex, Python matches the shortest possible string: '<To serve man>'.

In the greedy version, Python matches the longest possible string: '<To serve man> 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(), you can make the dot character 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.'
```

The regex noNewlineRegex, which did not have re.DOTALL passed to the re.compile() call that created it, will match everything only up to the first newline character, whereas newlineRegex, which did have re.DOTALL passed to re.compile(), matches everything. This is why the newlineRegex.search() call matches the full string, including its newline characters.

Review of Regex Symbols

This chapter covered a lot of notation, so here's a quick review of what you learned about basic regular expression syntax:

- 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 non-greedy 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')
>>> regex4 = re.compile('RobocOp')
```

matching the letters without worrying whether they're uppercase or lowercase. To make your regex caseinsensitive, you can pass re.IGNORECASE or re.I 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 not only find text patterns but can also substitute new text in place of those 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.'

Sometimes you may need to use the matched text itself 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, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could 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 Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the re.compile() function 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})?)')

you can spread the regular expression 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

What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization? Unfortunately, the re.compile() function takes only a single value as its second argument.

You can get around this limitation 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 if you want a 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)
```

Including all three options in the second argument will look like this:

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

Project: Phone Number and Email Address Extractor

#! 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)
# Create email regex.
 emailRegex = re.compile(r'''(
 [a-zA-Z0-9._%+-]+       # username  @ # @ symbol
 [a-zA-Z0-9.-]+             # domain name
 (\.[a-zA-Z]{2,4})         # dot-something
)''', re.VERBOSE)
# Find matches in clipboard text.
 text = str(pyperclip.paste())
matches = []
 for groups in phoneRegex.findall(text):
            phoneNum = '-'.join([groups[1], groups[3], groups[5]])
            if groups[8] != '': phoneNum += ' x' + groups[8]
                  matches.append(phoneNum)
 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 your web browser to the No Starch Press contact page at https://nostarch.com/contactus/, 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:

800-420-7240

415-863-9900

415-863-9950

info@nostarch.com

media@nostarch.com

academic@nostarch.com

info@nostarch.com

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.

For example, there is a file on my Windows laptop with the filename project.docx in the path C:\Users\Al\Documents.

The part of the filename after the last period is called the file's extension and tells you a file's type.

The filename project.docx is a Word document, and Users, Al, and Documents all refer to folders (also called directories).

Folders can contain files and other folders.

For example, project.docx is in the Documents folder, which is inside the Al folder, which is inside the Users folder.

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 macOS and Linux, the root folder is /.

Additional volumes, such as a DVD drive or USB flash drive, will appear differently on different operating systems.

On Windows, they appear as new, lettered root drives, such as D:\ or E:\.

On macOS, they appear as new folders under the /Volumes folder.

On Linux, they appear as new folders under the /mnt ("mount") folder.

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 macOS's TextEdit application.

Your programs can easily read the contents of plaintext files and treat them as an ordinary string value. Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs.

If you open a binaryfile in Notepad or TextEdit, it will look like scrambled nonsense.

The pathlib module's read_text() method returns a string of the full contents of a text file.

Its write_text() method creates a new text file (or overwrites an existing one) with the string passed to it.

```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

The more common way of writing to a file involves using the open() function and file objects.

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.

Opening Files with the open() Function

To open a file with the open() function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path. The open() function returns a File object.

Try it by creating 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.

```
>>> helloFile = open(Path.home() / 'hello.txt')
```

The open() function can also accept strings.

If you're using Windows,

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

Reading the Contents of Files

Now that you have a File object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the File object's read() method.

Let's continue with the hello.txt File object you stored in helloFile.

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

you can use the readlines() method to get a list of string values from the file, one string for each line of text.

```
>>> sonnetFile = open(Path.home() / '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,']
```

Writing to Files

Python allows you to write content to a file in a way similar to how the print() function "writes" strings to the screen. You can't write to a file you've opened in read mode.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value.

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. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether.

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.

```
>>> 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.
```


Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the shelve module.

This way, your program can restore data to variables from the hard drive.

The shelve module will let you 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.

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

To read and write data using the shelve module, you first import shelve. Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary.

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

```
>>> 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.

Using pprint.pformat() will give you a string that you can write to a .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()
```

Project: Updatable Multi-Clipboard

```
#! python3
import shelve, pyperclip, sys
mcbShelf = shelve.open('mcb')
# 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()

Organizing Files:

The shutil Module

The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs.

To use the **shutil functions**, you will first need to use import shutil. 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 can be strings or Path objects.).If destination is a filename, it will be used as the new name of the copied file.This function returns a string or Path object of the copied file.

```
>>> import shutil, os
 >>> from pathlib import Path
 >>> p = Path.home()
>>> shutil.copy(p / 'spam.txt', p / 'some_folder')
 'C:\\Users\\Al\\some_folder\\spam.txt'
>>> shutil.copy(p / 'eggs.txt', p / 'some_folder/eggs2.txt')
WindowsPath('C:/Users/Al/some_folder/eggs2.txt')
```

While **shutil.copy()** will copy a single file**, shutil.copytree()** will copy an entire folder and every folder and file contained in it.

Calling shutil .copytree(source, destination) will copy the folder at the path 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
>>> from pathlib import Path
 >>> p = Path.home()
>>> shutil.copytree(p / 'spam', p / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

The shutil.copytree() call creates a new folder named spam_backup with the same content as the original spam folder. You have now safely backed up your precious, precious spam.

Moving and Renaming Files and Folders

Calling 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'
```

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.

```
import os
 from pathlib import Path
 for filename in Path.home().glob('*.rxt'):
        os.unlink(filename)
```

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 --user 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.

```
 >>> import send2trash
 >>> baconFile = open('bacon.txt', 'a')                # creates the file
 >>> baconFile.write('Bacon is not a vegetable.')
 25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```


Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go.

Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

```
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.

You can use os.walk() in a for loop statement to walk a directory tree, much like how you can use the range() function to walk over a range of numbers. Unlike range(), the os.walk() function will return three values on each iteration through the loop:

• A string of the current folder's name

• A list of strings of the folders in the current folder

- A list of strings of the files in the current folder

Compressing Files with the zipfile Module

You may be familiar with 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, it's a handy way to package several files into one.

This single file, called an archive file, can then be, say, attached to an email.

Your Python programs can create and open (or extract) ZIP files using functions in the zipfile module.

Reading ZIP Files

To read the contents of a ZIP file, first you must create a ZipFile object (note the capital letters Z and F). ZipFile objects are conceptually similar to the File objects you saw returned by the open() function in the previous chapter: 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
>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / '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
>>> f'Compressed file is {round(spamInfo.file_size / spamInfo .compress_size, 2)}x smaller!' )
'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.

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
>>> from pathlib import Path
>>> p = Path.home()
```

```
>>> exampleZip = zipfile.ZipFile(p / 'example.zip')
 >>> exampleZip.extractall()
>>> exampleZip.close()
```

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the ZipFile object 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; you can always just set this value 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.

Debugging

Raising Exceptions Python raises an exception whenever it tries to execute invalid code.

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

```
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))
```

Using the try and except statements, you can handle errors 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.

```
def spam():
    bacon()
def bacon():
    raise Exception('This is the error message.')
spam()
```

When you run errorExample.py, the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in
    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.
```


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

In plain English, an assert statement says, "I assert that the condition holds true, and if not, there is a bug somewhere, so immediately stop the program."

```
>>> ages.sort()
>>> ages
[15, 17, 22, 26, 47, 54, 57, 73, 80, 92]
>>> assert ages[0] <= ages[-1]                    # Assert that the first age is <= the last age.
```

The assert statement here asserts that the first item in ages should be less than or equal to the last one. This is a sanity check; if the code in sort() is bug-free and did its job, then the assertion would be true. Because the ages[0] <= ages[-1] expression evaluates to True, the assert statement does nothing.

Logging

Logging is a great way to understand what's happening in your program and in what order it's happening.

Python's logging module makes it easy to create a record of custom messages that you write.

These log messages will describe when the program execution has reached the logging function call and list any variables you 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 lets you specify what details about the LogRecord object you want to see and how you want those details displayed.

IDLE /Mu's Debugger

The debugger is a feature of the Mu editor, IDLE, and other editor software 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 you to tell it to continue.

By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime.

This is a valuable tool for tracking down bugs.

UNIT-4: Classes and objects

Programmer-defined types : (class)

A programmer-defined type is also called a class. A class definition looks like this:

class Point:

"""Represents a point in 2-D space."""

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.

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.

Every object is an instance of some class, so "object" and "instance" are interchangeable.

Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

In this case, though, we are assigning values to named elements of an object. These elements are called attributes.

You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> z = blank.x
>>> z
3.0
```

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:

```
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
```

An object that is an attribute of another object is embedded.

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)
```

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.

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
```

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
```

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.

Classes and functions

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
```


Pure functions

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.

```
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
```

Modifiers 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
```


Anything that can be done with modifiers can also be done with pure functions.

In fact, some programming languages only allow pure functions.

There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers.

But modifiers are convenient at times, and functional programs tend to be less efficient.

Prototyping versus planning

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.

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
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that time_to_int(int_to_time(x)) == x for many values of x.

This is an example of a consistency check. 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)
```

Classes and methods:

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.

Methods: 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.

Printing objects

```
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))
```

Another example

Here's a version of increment 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
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.

positional argument is an argument that doesn't have a parameter name; that is, it is not a keyword argument.

In function call: sketch(parrot, cage, dead=True)

parrot and cage are positional, and dead is a keyword argument.

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
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

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 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.

The __str__ method __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
```


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**.

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.

Polymorphism 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.

For example, a histogram to count the number of times each letter appears in a word.

```
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 s 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.

Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations.

For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include time_to_int, is_after, and add_time.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

Inheritance:

Inheritance is the ability to define a new class that is a modified version of an existing class.

Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge).

The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King.

Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be.

One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to encode the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades 7→ 3

Hearts 7→ 2

Diamonds 7→ 1

Clubs 7→ 0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack 7→ 11

Queen 7→ 12

King 7→ 13

I am using the 7→ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for Card looks like this:

```
class Card:
    """Represents a standard playing card."""
    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call Card with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits.

A natural way to do that is with lists of strings. We assign these lists to **class attributes**:

```
# inside class Card:
suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']
def __str__(self):
    return '%s of %s' % (Card.rank_names[self.rank], Card.suit_names[self.suit])
```


Variables like suit_names and rank_names, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object Card.

This term distinguishes them from variables like suit and rank, which are called **instance attributes** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation.

For example, in __str__, self is a Card object, and self.rank is its rank.

Similarly, Card is a class object, and Card.rank_names is a list of strings associated with the class.

Comparing cards

For built-in types, there are relational operators (<,>, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another.

For programmer-defined types, we can override the behavior of the built-in operators by providing a method named __lt__, which stands for "less than".

__lt__ takes two parameters, self and other, and returns True if self is strictly less than other.

The correct ordering for cards is not obvious.

For example, which is better, the 3 of Clubs or the 2 of Diamonds?

One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write __lt__:

```
# inside class Card:
def __lt__(self, other): # check the suits
    if self.suit < other.suit:   return True
  if self.suit > other.suit:     return False
# suits are the same... check ranks
    return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:
def __lt__(self, other):
    t1 = self.suit, self.rank t2 = other.suit, other.rank
        return t1 < t2
```

Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute. The following is a class definition for Deck.

The init method creates the attribute cards and generates the standard set of fifty-two cards:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13.

Each iteration creates a new Card with the current suit and rank, and appends it to self.cards.

Printing the deck

Here is a __str__ method for Deck:

```
#inside class Deck:
def __str__(self):
    res = []
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method join.

The built-in function str invokes the __str__ method on each card and returns the string representation. Since we invoke join on a newline character, the cards are separated by newlines.

Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method pop provides a convenient way to do that:

```
#inside
class Deck:
    def pop_card(self):
        return self.cards.pop()
```

Since pop removes the last card in the list, we are dealing from the bottom of the deck. To add a card, we can use the list method append: #inside class Deck:

```
def add_card(self, card):
    self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a veneer. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case add_card is a "thin" method that expresses a list operation in terms appropriate for decks.

It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named shuffle using the function shuffle from the random module

```
: # inside class Deck:
def shuffle(self):
    random.shuffle(self.cards)
```

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class.

To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that Hand inherits from Deck; that means we can use methods like pop_card and add_card for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the parent and the new class is called the child.

In this example, Hand inherits __init__ from Deck, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize cards with an empty list.

If we provide an init method in the Hand class, it overrides the one in the Deck class:

```
# inside class Hand:
def __init__(self, label=''):
    self.cards = []
    self.label = label
```

When you create a Hand, Python invokes this init method, not the one in Deck.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```


The other methods are inherited from Deck, so we can use pop_card and add_card to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called move_cards:

```
#inside class Deck:
def move_cards(self, hand, num):
    for i in range(num):
        hand.add_card(self.pop_card())
```

move_cards takes two arguments, a Hand object and the number of cards to deal. It modifies both self and hand, and returns None.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it.

Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them.

In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition.

The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values.

These diagrams represent a snapshot in the execution of a program, so they change as the program runs. They are also highly detailed; for some purposes, too detailed.

A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

• Objects in one class might contain references to objects in another class.

For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called HAS-A, as in, "a Rectangle has a Point."

• One class might inherit from another. This relationship is called IS-A, as in, "a Hand is a kind of a Deck."
• One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation.

This kind of relationship is called a dependency. A class diagram is a graphical representation of these relationships.

For example, Figure 18.2 shows the relationships between Card, Deck and Hand

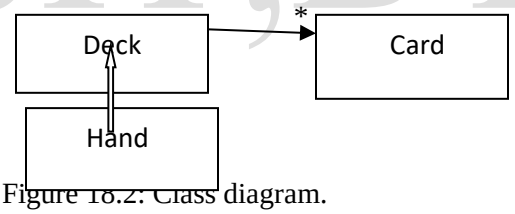

Figure 18.2: Class diagram.

The hollow arrow head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has.

A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a list of Cards, but built-in types like list and dict are usually not included in class diagrams.

Data encapsulation

We identified objects we needed—like Point, Rectangle and Time—and defined classes to represent them.

In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan.

In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by data encapsulation.

Markov analysis, it uses two global variables—suffix_map and prefix—that are read and written from several functions.

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time.

If we read two texts, their prefixes and suffixes would be added to the same data .

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

```
class Markov:
    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here's process_word:

```
def process_word(self, word, order=2):
    if len(self.prefix) < order:
        self.prefix += (word,)
        return
try:
    self.suffix_map[self.prefix].append(word)
except KeyError: # if there is no entry for this prefix, make one self.
    suffix_map[self.prefix] = [word]
    self.prefix = shift(self.prefix, word)
```


Transforming a program like this—changing the design without changing the behavior—is another example of refactoring .

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).

2. Once you get the program working, look for associations between global variables and the functions that use them.

3. Encapsulate related variables as attributes of an object.

4. Transform the associated functions into methods of the new class.

MODULE-5: Web Scraping

Web scraping is the term for using a program to download and process content from the web. For example, Google runs many web scraping programs to index web pages for its search engine.

Project: mapIt.py with the webbrowser Module

The webbrowser module's open() function can launch a new browser to a specified URL.

```
>>> import webbrowser
>>> webbrowser.open('https://inventwithpython.com/')
```

A simple script to automatically launch the map in your browser using the contents of your clipboard.

This is what your program does:

1. Gets a street address from the command line arguments or clipboard
2. Opens the web browser to the Google Maps page for the address

This means your code will need to do the following:

1. Read the command line arguments from sys.argv.
2. Read the clipboard contents.
3. Call the webbrowser.open() function to open the web browser.

Open a new file editor tab and save it as mapIt.py

Step 1: Figure Out the URL

C:\> mapIt 870 Valencia St, San Francisco, CA 94110

Step 2: Handle the Command Line Arguments AND

Step 3: Handle the Clipboard Content and Launch the Browser

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
 # command line or clipboard
import webbrowser, sys, pyperclip
 if len(sys.argv) > 1: # Get address from command line.
      address = ' '.join(sys.argv[1:])
else: # Get address from clipboard.
      address = pyperclip.paste()
 webbrowser.open('https://www.google.com/maps/place/' + address)
```


Downloading Files from the Web with the requests Module

The requests module lets you easily download files from the web without having to worry about complicated issues such as network errors, connection problems, and data compression.

Downloading a Web Page with the requests.get() Function

The requests.get() function takes a string of a URL to download.

By calling type() on requests.get()'s return value, you can see that it returns a Response object, which contains the response that the web server gave for your request.

```
>>> import requests
 >>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
```

```
>>> type(res)
<class 'requests.models.Response'>
>>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare
This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project.
```

Saving Downloaded Files to the Hard Drive

you can save the web page to a file on your hard drive with the standard open() function and write() method.

There are some slight differences, though.

First, you must open the file in write binary mode by passing the string 'wb' as the second argument to open(). Even if the page is in, you need to write binary data instead of text data in order to maintain the Unicode encoding of the text.

To write the web page to a file, you can use a for loop with the Response object's iter_content() method.

```
>>> import requests
>>> res = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
        playFile.write(chunk)
    100000
    78981
>>> playFile.close()
```


The iter_content() method returns "chunks" of the content on each iteration through the loop.

Each chunk is of the bytes data type, and you get to specify how many bytes each chunk will contain.

One hundred thousand bytes is generally a good size.

To review, here's the complete process for downloading and saving a file:

1. Call requests.get() to download the file.
2. Call open() with 'wb' to create a new file in write binary mode.
3. Loop over the Response object's iter_content() method.
4. Call write() on each iteration to write the content to the file.
5. Call close() to close the file.

HTML

Hypertext Markup Language (HTML) is the format that web pages are written in.

You'll also see how to access your web browser's powerful developer tools, which will make scraping information from the web much easier.

An HTML file is a plaintext file with the .html file extension.

The text in these files is surrounded by tags, which are words enclosed in angle brackets.

The tags tell the browser how to format the web page.

A starting tag and closing tag can enclose some text to form an element.

The text (or inner HTML) is the content between the starting and closing tags.

For example, the following HTML will display Hello, world! in the browser, with Hello in bold:

 Hello</ strong>, world!

The opening **tag says that the enclosed text will appear in bold. The closing** tags tells the browser where the end of the bold text is.

There are many different tags in HTML.

Some of these tags have extra properties in the form of attributes within the angle brackets.

For example, the tag encloses text that should be a link.

The URL that the text links to is determined by the href attribute. Here's an example:

Al's free< a href=https://inventwithpython.com> Python books.

Some elements have an id attribute that is used to uniquely identify the element in the page.

Parsing HTML with the bs4 Module

Beautiful Soup is a module for extracting information from an HTML page .

The Beautiful Soup module's name is bs4 (for Beautiful Soup, version 4).

The Beautiful Soup examples will parse (that is, analyze and identify the parts of) an HTML file on the hard drive.

Creating a BeautifulSoup Object from HTML The bs4.

BeautifulSoup() function needs to be called with a string containing the HTML it will parse.

The bs4.BeautifulSoup() function returns a BeautifulSoup object.

```
>>> import requests, bs4
>>> res = requests.get('https://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text, 'html.parser')
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

This code uses requests.get() to download the main page from the No Starch Press website and then passes the text attribute of the response to bs4.BeautifulSoup().

The BeautifulSoup object that it returns is stored in a variable named noStarchSoup.

You can also load an HTML file from your hard drive by passing a File object to bs4.BeautifulSoup() along with a second argument that tells Beautiful Soup which parser to use to analyze the HTML.

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile, 'html.parser')
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

Once you have a BeautifulSoup object, you can use its methods to locate specific parts of an HTML document.

Finding an Element with the select() Method

We can retrieve a web page element from a BeautifulSoup object by calling the select()method and passing a string of a CSS selector for the element you are looking for.

Selectors are like regular expressions: they specify a pattern to look for—in this case, in HTML pages instead of general text strings.

Table: Examples of CSS Selectors

Selector passed to the select() method	Will match . . .
soup.select('div')	All elements named
soup.select('#author')	The element with an id attribute of author
soup.select('.notice')	All elements that use a CSS class attribute named notice
soup.select('div span')	All elements named that are within an element named
soup.select('div > span')	All elements named that are directly within an element named , with no other element in between
soup.select('input[name]')	All elements named<input>that have a name attribute with any value
soup.select('input[type="button"]')	All elements named <input>that have an attribute named type with value button

The various selector patterns can be combined to make sophisticated matches.

For example, soup.select('p #author') will match any element that has an id attribute of author, as long as it is also inside a element.

The select() method will return a list of Tag objects, which is how Beautiful Soup represents an HTML element.

The list will contain one Tag object for every match in the BeautifulSoup object's HTML.

Tag values can be passed to the str() function to show the HTML tags they represent.

Tag values also have an attrs attribute that shows all the HTML attributes of the tag as a dictionary.

```
>>> import bs4
>>> exampleFile = open('example.html')
```

```
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read(), 'html.parser')
>>> elems = exampleSoup.select('#author')
>>> type(elems)              # elems is a list of Tag objects.
<class ‘list>
>>> len(elems)
 1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> str(elems[0]) # The Tag object as a string.
<span id=”author”> 'Al Sweigart' </span>
>>> elems[0].getText()
 'Al Sweigart'
 >>> elems[0].attrs
 {'id': 'author'}
```

This code will pull the element with id="author" out of our example HTML.

We use select('#author') to return a list of all the elements with id="author".

We store this list of Tag objects in the variable elems, and len(elems) tells us there is one Tag object in the list; there was one match.

Calling getText() on the element returns the element’s text, or inner HTML.

The text of an element is the content between the opening and closing tags: in this case, 'Al Sweigart'.

Passing the element to str() returns a string with the starting and closing tags and the element’s text.

Finally, attrs gives us a dictionary with the element’s attribute, 'id', and the value of the id attribute, 'author'.

We can also pull all the elements from the BeautifulSoup object.

```
>>> pElems = exampleSoup.select('p')
 >>> str(pElems[0])
<p>Download my <strong>Python< /strong> book from <a href="https:// inventwithpython.com">
my website </a>.</p>
>>> pElems[0].getText()
 'Download my Python book from my website.'
 >>> str(pElems[1])
‘<p class="slogan">Learn Python the easy way!</p>’
>>> pElems[1].getText()
 'Learn Python the easy way!'
>>> str(pElems[2])
‘<p>By <span id="author">Al Sweigart< /span></p>’
```

```
>>> pElems[2].getText()
'By Al Sweigart'
```

Getting Data from an Element's Attributes

The get() method for Tag objects makes it simple to access attribute values from an element.

The method is passed a string of an attribute name and returns that attribute's value.

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'), 'html.parser'
 >>> spanElem = soup.select('span')[0]
 >>> str(spanElem) '
'<p>By <span id="author">Al Sweigart< /span></p>'
>>> spanElem.get('id')
 'author'
>>> spanElem.get('some_nonexistent_addr') == None
 True
>>> spanElem.attrs
 {'id': 'author'}
```


Here we use select() to find any elements and then store the first matched element in spanElem. Passing the attribute name 'id' to get() returns the attribute's value, 'author'.

Controlling the Browser with the selenium Module

The selenium module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there were a human user interacting with the page.

Using selenium, you can interact with web pages in a much more advanced way than with requests and bs4; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the web.

Still, if you need to interact with a web page in a way that, say, depends on the JavaScript code that updates the page, you'll need to use selenium instead of requests.

That's because major ecommerce websites such as Amazon almost certainly have software systems to recognize traffic that they suspect is a script harvesting their info or signing up for multiple free accounts.

These sites may refuse to serve pages to you after a while, breaking any scripts you've made.
The selenium module is much more likely to function on these sites long-term than requests.

Starting a selenium-Controlled Browser

Importing the modules for selenium is slightly tricky.

Instead of import selenium, you need to run *from selenium import webdriver.*

You can launch the Firefox browser with selenium.

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'
> >>> browser.get('https://inventwithpython.com')
```


You'll notice when webdriver.Firefox() is called, the Firefox web browser starts up.
Calling type() on the value webdriver.Firefox() reveals it's of the WebDriver data type.
And calling browser.get('https://inventwithpython.com') directs the browser to https://inventwithpython.com/.

Finding Elements on the Page

WebDriver objects have quite a few methods for finding elements on a page.

They are divided into the find_element_* and find_elements_* methods.

The find_element_* methods return a single WebElement object, representing the first element on the page that matches your query.
The find_elements_* methods return a list of WebElement_* objects for every matching element on the page.

Table shows several examples of find_element_* and find_elements_* methods being called on a WebDriver object that's stored in the variable browser.

Table: Selenium's WebDriver Methods for Finding Elements

Method name	WebElement object/list returned
browser.find_element_by_class_name(name)	Elements that use the CSS class name
browser.find_elements_by_class_name(name)	
browser.find_element_by_css_selector(selector)	
browser.find_elements_by_css_selector(selector)	Elements that match the CSS selector
browser.find_element_by_id(id)	
browser.find_elements_by_id(id)	Elements with a matching id attribute value
browser.find_element_by_link_text(text)	
browser.find_elements_by_link_text(text)	elements that completely match the text provided
browser.find_element_by_partial_link_text(text)	
browser.find_elements_by_partial_link_text(text)	elements that contain the text provided
browser.find_element_by_name(name)	
browser.find_elements_by_name(name)	Elements with a matching name attribute value
browser.find_element_by_tag_name(name)	
browser.find_elements_by_tag_name(name)	Elements with a matching tag name (case-insensitive; an element is matched by 'a' and 'A')

Except for the *_by_tag_name() methods, the arguments to all the methods are case sensitive.

If no elements exist on the page that match what the method is looking for, the selenium module raises a NoSuchElement exception.

Once you have the WebElement object, you can find out more about it by reading the attributes or calling the methods in Table.

Table : WebElement Attributes and Methods

Attribute or method	Description

tag_name	The tag name, such as 'a' for an element
get_attribute(name)	The value for the element's name attribute
text	The text within the element, such as 'hello' in hello
clear()	For text field or text area elements, clears the text typed into it
is_displayed()	Returns True if the element is visible; otherwise returns False
is_enabled()	For input elements, returns True if the element is enabled; otherwise returns False
is_selected()	For checkbox or radio button elements, returns True if the element is selected; otherwise returns False
location	A dictionary with keys 'x' and 'y' for the position of the element in the page

For example:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('https://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name(' cover-thumb')
    print('Found <%s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```


Here we open Firefox and direct it to a URL. On this page, we try to find elements with the class name 'bookcover', and if such an element is found, we print its tag name using the tag_name attribute. If no such element was found, we print a different message.

This program will output the following:

Found < img> element with that class name!

We found an element with the class name 'bookcover' and the tag name 'img'.

Clicking the Page

WebElement objects returned from the find_element_* and find_elements_* methods have a click() method that simulates a mouse click on that element.

This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse.

For example

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://inventwithpython.com')
>>> linkElem = browser.find_element_by_link_text('Read Online for Free')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.FirefoxWebElement'>
>>> linkElem.click()          # follows the "Read Online for Free" link
```

This opens Firefox to https://inventwithpython.com/, gets the WebElement object for the element with the text Read It Online, and then simulates clicking that element.

It's just like if you clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the<input> or <textarea> element for that text field and then calling the send _keys()method.

For example

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('https://login.metafilter.com')
>>> userElem = browser.find_element_by_id('user_name')
>>> userElem.send_keys('your_real_username_here')
>>> passwordElem = browser.find_element_by_id('user_pass')
>>> passwordElem.send_keys('your_real_password_here')
>>> passwordElem.submit()
```

As long as login page for MetaFilter hasn't changed the id of the Username and Password text fields since this book was published, the previous code will fill in those text fields with the provided text.

(You can always use the browser's inspector to verify the id.)

Calling the submit() method on any element will have the same result as clicking the Submit button for the form that element is in.

(You could have just as easily called emailElem.submit(), and the code would have done the same thing.)

Clicking Browser Buttons

The selenium module can simulate clicks on various browser buttons as well through the following methods:

browser.back() Clicks the Back button.

browser.forward() Clicks the Forward button.

browser.refresh() Clicks the Refresh/Reload button.

browser.quit() Clicks the Close Window button.

Working with Excel Spreadsheets:

spreadsheets are used to organize information into two-dimensional data structures, perform calculations with formulas, and produce output as charts.

we'll integrate Python into two popular spreadsheet applications:

Microsoft Excel and Google Sheets.

Excel is a popular and powerful spreadsheet application for Windows.

The openpyxl module allows your Python programs to read and modify Excel spreadsheet files.

Excel Documents

An Excel spreadsheet document is called a workbook.

A single workbook is saved in a file with the .xlsx extension.

Each workbook can contain multiple sheets (also called worksheets).

The sheet the user is currently viewing (or last viewed before closing Excel) is called the active sheet. Each sheet has columns (addressed by letters starting at A) and rows (addressed by numbers starting at 1).

A box at a particular column and row is called a cell.

Each cell can contain a number or text value.

The grid of cells with data makes up a sheet.

Reading Excel Documents:

Opening Excel Documents with OpenPyXL

Once you've imported the openpyxl module, you'll be able to use the openpyxl .load_workbook() function.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The openpyxl.load_workbook() function takes in the filename and returns a value of the workbook data type. This Workbook object represents the Excel file.

Getting Sheets from the Workbook

We can get a list of all the sheet names in the workbook by accessing the sheetnames attribute.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
 >>> wb.sheetnames # The workbook's sheets' names.
 ['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb['Sheet3'] # Get a sheet from the workbook.
 >>> sheet
<Worksheet "Sheet3">
 >>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title # Get the sheet's title as a string.
 'Sheet3'
 >>> anotherSheet = wb.active # Get the active sheet.
>>> anotherSheet
<Worksheet "Sheet1">
```

Getting Cells from the Sheets

Once you have a Worksheet object, you can access a Cell object by its name.


```
 >>> import openpyxl
 >>> wb = openpyxl.load_workbook('example.xlsx')
 >>> sheet = wb['Sheet1'] # Get a sheet from the workbook.
 >>> sheet['A1'] # Get a cell from the sheet.
<cell 'Sheet1'.A1>
 >>> sheet['A1'].value # Get the value from the cell.
 datetime.datetime(2015, 4, 5, 13, 34, 2)
 >>> c = sheet['B1'] # Get another cell from the sheet.
>>> c.value
 'Apples'
>>> # Get the row, column, and value from the cell.
 >>> 'Row %s, Column %s is %s' % (c.row, c.column, c.value)
 'Row 1, Column B is Apples'
 >>> 'Cell %s is %s' % (c.coordinate, c.value)
 'Cell B1 is Apples'
>>> sheet['C1'].value
 73
```

The Cell object has a value attribute that contains, the value stored in that cell.

you can also get a cell using the sheet's cell() method and passing integers for its row and column keyword arguments. The first row or column integer is 1, not 0.

```
>>> sheet.cell(row=1, column=2)
<Cell 'Sheet1'.B1>
 >>> sheet.cell(row=1, column=2).value
 'Apples' >>>
 for i in range(1, 8, 2): # Go through every other  row:
 ... print(i, sheet.cell(row=i, column=2).value)
 ...
1 Apples
3 Pears
 5 Apples
 7 Strawberries
```

You can determine the size of the sheet with the Worksheet object's max_row and max_column attributes.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb['Sheet1']
 >>> sheet.max_row        # Get the highest row number.
7
>>> sheet.max_column     # Get the highest column number.
3
```


Project: Reading Data from a Spreadsheet

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds. This is what your program does:

1. Reads the data from the Excel spreadsheet
2. Counts the number of census tracts in each county
3. Counts the total population of each county
4. Prints the results

This means your code will need to do the following:

1. Open and read the cells of an Excel document with the openpyxl module.
2. Calculate all the tract and population data and store it in a data structure.
3. Write the data structure to a text file with the .py extension using the pprint module.

Example:

There is just one sheet in the censuspopdata.xlsx spreadsheet, named 'Population by Census Tract', and each row holds the data for a single census tract.

The columns are the tract number (A),
the state abbreviation (B),
the county name (C), and
the population of the tract (D).

```
#! python3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

import openpyxl, pprint

print('Opening workbook...')
wb = openpyxl.load_workbook('censuspopdata.xlsx')

sheet = wb['Population by Census Tract']
countyData = {}


# Fill in countyData with each county's population and tracts.

countyData ={'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1}, 'Aleutians West': {'pop': 5561, 'tracts': 2}, 'Anchorage': {'pop': 291826, 'tracts': 55}, 'Bethel': {'pop': 17013, 'tracts': 3}, 'Bristol Bay': {'pop': 997, 'tracts': 1}}

print('Reading rows...')
for row in range(2, sheet.max_row + 1):

    # Each row in the spreadsheet has data for one census tract.

    state = sheet['B' + str(row)].value

    county = sheet['C' + str(row)].value
```

```
        pop = sheet['D' + str(row)].value

# Make sure the key for this state exists.

countyData.setdefault(state, {})

 # Make sure the key for this county in this state exists.

 countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})
 # Each row represents one census tract, so increment by one.

countyData[state][county]['tracts'] += 1

 # Increase the county pop by the pop in this census tract.

 countyData[state][county]['pop'] += int(pop)

# Open a new text file and write the contents of countyData to it.

 print('Writing results...')
 resultFile = open('census2010.py', 'w')

 resultFile.write('allData = ' + pprint.pformat(countyData))
 resultFile.close()
 print('Done.')
```


Writing Excel Documents

OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files.

With Python, it's simple to create spreadsheets with thousands of rows of data.

Creating and Saving Excel Documents Call the openpyxl.Workbook() function to create a new, blank Workbook object.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()          # Create a blank workbook.
>>> wb.sheetnames                     # It starts with one sheet.
['Sheet']
 >>> sheet = wb.active
 >>> sheet.title
```

'Sheet'

>>> sheet.title = 'Spam Bacon Eggs Sheet' # Change title.

>>> wb.sheetnames

['Spam Bacon Eggs Sheet']

Any time you modify the Workbook object or its sheets and cells, the spreadsheet file will not be saved until you call the save() workbook method.

>>> import openpyxl

>>> wb = openpyxl.load_workbook('example.xlsx')

>>> sheet = wb.active

>>> sheet.title = 'Spam Spam Spam'

>>> wb.save('example_copy.xlsx') # Save the workbook.

Creating and Removing Sheets

Sheets can be added to and removed from a workbook with the create_sheet() method and del operator.

>>> import openpyxl

>>> wb = openpyxl.Workbook()

>>> wb.sheetnames ['Sheet']

>>> wb.create_sheet() # Add a new sheet.

>>> wb.sheetnames ['Sheet', 'Sheet1']

>>> # Create a new sheet at index 0.

>>> wb.create_sheet(index=0, title='First Sheet')

>>> wb.sheetnames ['First Sheet', 'Sheet', 'Sheet1']

>>> wb.create_sheet(index=2, title='Middle Sheet')

>>> wb.sheetnames ['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']

You can use the del operator to delete a sheet from a workbook, just like you can use it to delete a key-value pair from a dictionary.

Remember to call the save() method to save the changes after adding sheets to or removing sheets from the workbook.

>>> wb.sheetnames

['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']

>>> del wb['Middle Sheet']

>>> del wb['Sheet1']

>>> wb.sheetnames

['First Sheet', 'Sheet']

Writing Values to Cells

Writing values to cells is much like writing values to keys in a dictionary.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Hello, world!' # Edit the cell's value.
>>> sheet['A1'].value
'Hello, world!'
```

Project: Updating a Spreadsheet

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices.

Your program does the following:

1. Loops over all the rows
2. If the row is for garlic, celery, or lemons, changes the price.

This means your code will need to do the following:

1. Open the spreadsheet file.
2. For each row, check whether the value in column A is Celery, Garlic, or Lemon.
3. If it is, update the price in column B.
4. Save the spreadsheet to a new file (so that you don't lose the old spreadsheet, just in case).

Step 1: Set Up a Data Structure with the Update Information

The prices that you need to update are as follows:

```
Celery 1.19
Garlic 3.07
Lemon 1.27
```

You could write code like this:

```
if produceName == 'Celery':
    cellObj = 1.19
if produceName == 'Garlic':
    cellObj = 3.07
if produceName == 'Lemon':
    cellObj = 1.27
```

Step 2: Check All Rows and Update Incorrect Prices

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

 import openpyxl wb = openpyxl.load_workbook('produceSales.xlsx')

 sheet = wb['Sheet']

# The produce types and their updated prices
 PRICE_UPDATES = {'Garlic': 3.07, 'Celery': 1.19, 'Lemon': 1.27}

 # Loop through the rows and update the prices.

for rowNum in range(2, sheet.max_row):
 # skip the first row
      produceName = sheet.cell(row=rowNum, column=1).value ω
      if produceName in PRICE_UPDATES:
          sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]
wb.save('updatedProduceSales.xlsx')
```


Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet.
To customize font styles in cells, important, import the Font() function from the openpyxl.styles module.

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
 >>> sheet = wb['Sheet']
>>> italic24Font = Font(size=24, italic=True)          # Create a font.
>>> sheet['A1'].font = italic24Font                    # Apply the font to A1.
 >>> sheet['A1'] = 'Hello, world!'
 >>> wb.save('styles.xlsx')
```

Font Objects

To set font attributes, you pass keyword arguments to Font().
You can call Font() to create a Font object and store that Font object in a variable.
You then assign that variable to a Cell object's font attribute.

```
>>> import openpyxl
>>> from openpyxl.styles import Font
 >>> wb = openpyxl.Workbook()
 >>> sheet = wb['Sheet']
 >>> fontObj1 = Font(name='Times New Roman', bold=True)
>>> sheet['A1'].font = fontObj1
>>> sheet['A1'] = 'Bold Times New Roman'
>>> fontObj2 = Font(size=24, italic=True)
>>> sheet['B3'].font = fontObj2
>>> sheet['B3'] = '24 pt Italic'
>>> wb.save('styles.xlsx')
```

Formulas

Excel formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells.

An Excel formula is set just like any other text value in a cell.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
 >>> sheet = wb.active
 >>> sheet['A1'] = 200
 >>> sheet['A2'] = 300
 >>> sheet['A3'] = '=SUM(A1:A2)'       # Set the formula.
 >>> wb.save('writeFormula.xlsx')
```


Adjusting Rows and Columns

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header.

Setting Row Height and Column Width

Worksheet objects have row_dimensions and column_dimensions attributes that control row heights and column widths.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
 >>> sheet = wb.active
 >>> sheet['A1'] = 'Tall row'
 >>> sheet['B2'] = 'Wide column'
>>> # Set the height and width:
```

```
>>> sheet.row_dimensions[1].height = 70
 >>> sheet.column_dimensions['B'].width = 20
 >>> wb.save('dimensions.xlsx')
```

Merging and Unmerging Cells A rectangular area of cells can be merged into a single cell with the merge_cells() sheet method.

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> sheet.merge_cells('A1:D3') # Merge all these cells.
 >>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5') # Merge these two cells.
 >>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

To unmerge cells, call the unmerge_cells() sheet method.


```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.active
>>> sheet.unmerge_cells('A1:D3') # Split these cells up.
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

Freezing Panes

For spreadsheets too large to be displayed all at once, it's helpful to "freeze" a few of the top rows or leftmost columns onscreen.

Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as ***freeze panes***.

In OpenPyXL, each Worksheet object has a freeze_panes attribute that can be set to a Cell object or a string of a cell's coordinates.

Note that all rows above and all columns to the left of this cell will be frozen, but the row and column of the cell itself will not be frozen.

To unfreeze all panes, set freeze_panes to None or 'A1'.

Table shows which rows and columns will be frozen for some example settings of freeze_panes.

Table: Frozen Pane Examples

freeze_panes setting	Rows and columns frozen
sheet.freeze_panes = 'A2'	Row 1
sheet.freeze_panes = 'B1'	Column A
sheet.freeze_panes = 'C1'	Columns A and B
sheet.freeze_panes = 'C2'	Row 1 and columns A and B
sheet.freeze_panes = 'A1' or sheet.freeze_panes = None	No frozen panes

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.active
 >>> sheet.freeze_panes = 'A2'       # Freeze the rows above A2.
>>> wb.save('freezeExample.xlsx')
```

Charts OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells.

To make a chart, you need to do the following:

1. Create a Reference object from a rectangular selection of cells.
2. Create a Series object by passing in the Reference object.
3. Create a Chart object.
4. Append the Series object to the Chart object.
5. Add the Chart object to the Worksheet object, optionally specifying which cell should be the top-left corner of the chart.

The Reference object requires some explaining. You create Reference objects by calling the openpyxl.chart.Reference() function and passing three arguments:

1. The Worksheet object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data:

the first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.

3. A tuple of two integers, representing the bottom-right cell of the rectangular selection of cells containing your chart data:

the first integer in the tuple is the row, and the second is the column.

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
 >>> wb = openpyxl.Workbook()
 >>> sheet = wb.active
```

```
>>> for i in range(1, 11): # create some data in column A
...         sheet['A' + str(i)] = i
...
>>> refObj = openpyxl.chart.Reference(sheet, min_col=1, min_row=1, max_col=1, max_row=10)
>>> seriesObj = openpyxl.chart.Series(refObj, title='First series')
 >>> chartObj = openpyxl.chart.BarChart()
 >>> chartObj.title = 'My Chart'
 >>> chartObj.append(seriesObj)
 >>> sheet.add_chart(chartObj, 'C5')
 >>> wb.save('sampleChart.xlsx')
```

We've created a bar chart by calling openpyxl.chart.BarChart().

You can also create line charts, scatter charts, and pie charts by calling openpyxl.charts .LineChart(), openpyxl.chart.ScatterChart(), and openpyxl.chart.PieChart().

Working with PDF and Word Documents

PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information.

If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to open().

PDF Documents PDF stands for Portable Document Format and uses the .pdf file extension.

Extracting Text from PDFs *PyPDF2* does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python string.

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
 >>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
>>> pdfReader.numPages 19
 >>> pageObj = pdfReader.getPage(0)
 >>> pageObj.extractText()
'OOFFFFIICCIIAALL BBOOAARRDD MMIINNUUTTEESS Meeting of March 7, 2015 \n The Board of Elementary and Secondary Education shall provide leadership and create policies for education that expand opportunities for children, empower families and communities, and advance Louisiana in an increasingly competitive global market. BOARD of ELEMENTARY and SECONDARY EDUCATION '
```

```
>>> pdfFileObj.close()
```

First, import the PyPDF2 module. Then open meetingminutes.pdf in read binary mode and store it in pdfFileObj.

To get a PdfFileReader object that represents this PDF, call PyPDF2.PdfFileReader() and pass it pdfFileObj. Store this PdfFileReader object in pdfReader.

The total number of pages in the document is stored in the numPages attribute of a PdfFileReader object . The example PDF has 19 pages, but let's extract text from only the first page.

Decrypting PDFs Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password.

Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password rosebud:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb')).
>>> pdfReader.isEncrypted
True
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
>>> pdfReader.decrypt('rosebud')
1
>>> pageObj = pdfReader.getPage(0)
```


To read an encrypted PDF*,*

call the decrypt() function and pass the password as a string. After you call decrypt() with the correct password, you'll see that calling getPage() no longer causes an error. If given the wrong password, the decrypt() function will return 0 and getPage() will continue to fail.

Note that the decrypt() method decrypts only the PdfFileReader object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted.

Creating PDFs

PyPDF2's counterpart to PdfFileReader is PdfFileWriter, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF.

PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document.

The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into PdfFileReader objects.
2. Create a new PdfFileWriter object.
3. Copy pages from the PdfFileReader objects into the PdfFileWriter object.
4. Finally, use the PdfFileWriter object to write the output PDF.

Creating a PdfFileWriter object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the PdfFileWriter's write() method.

Copying Pages

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

```
>>> import PyPDF2
>>> pdf1File = open('meetingminutes.pdf', 'rb')
>>> pdf2File = open('meetingminutes2.pdf', 'rb')
>>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
>>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdf1Reader.numPages):
        pageObj = pdf1Reader.getPage(pageNum)
        pdfWriter.addPage(pageObj)
>>> for pageNum in range(pdf2Reader.numPages):
        pageObj = pdf2Reader.getPage(pageNum)
        pdfWriter.addPage(pageObj)
>>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
>>> pdfWriter.write(pdfOutputFile)
>>> pdfOutputFile.close()
>>> pdf1File.close()
>>> pdf2File.close()
```

Rotating Pages

The pages of a PDF can also be rotated in 90-degree increments with the rotateClockwise() and rotateCounterClockwise() methods. Pass one of the integers 90, 180, or 270 to these methods.

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
>>> page = pdfReader.getPage(0)
```

```
>>> page.rotateClockwise(90)
 {'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
 --snip--
 }
>>> pdfWriter = PyPDF2.PdfFileWriter()
 >>> pdfWriter.addPage(page)
>>> resultPdfFile = open('rotatedPage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

```
>>> import PyPDF2
 >>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
 >>> minutesFirstPage = pdfReader.getPage(0)
>>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
 >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(minutesFirstPage)
 >>> for pageNum in range(1, pdfReader.numPages):
            pageObj = pdfReader.getPage(pageNum)
           pdfWriter.addPage(pageObj)
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
 >>> pdfWriter.write(resultPdfFile)
 >>> minutesFile.close()
 >>> resultPdfFile.close()
```

Encrypting PDFs

A PdfFileWriter object can also add encryption to a PDF document

```
>>> import PyPDF2
 >>> pdfFile = open('meetingminutes.pdf', 'rb')
 >>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
```

```
>>> for pageNum in range(pdfReader.numPages):
        pdfWriter.addPage(pdfReader.getPage(pageNum))
>>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

Project: Combining Select Pages from Many PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file.

Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result.

Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together.

Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

1. Find all PDF files in the current working directory.
2. Sort the filenames so the PDFs are added in order.
3. Write each page, excluding the first page, of each PDF to the output file.

In terms of implementation, your code will need to do the following:

1. Call os.listdir() to find all the files in the working directory and remove any non-PDF files.
2. Call Python's sort() list method to alphabetize the filenames.
3. Create a PdfFileWriter object for the output PDF.
4. Loop over each PDF file, creating a PdfFileReader object for it.
5. Loop over each page (except the first) in each PDF file.
6. Add the pages to the output PDF.
7. Write the output PDF to a file named allminutes.pdf.

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# into a single PDF.

import PyPDF2, os
```

```
# Get all the PDF filenames.
 pdfFiles = []
 for filename in os.listdir('.'):
        if filename.endswith('.pdf'):
              pdfFiles.append(filename)
   pdfFiles.sort(key = str.lower)
   pdfWriter = PyPDF2.PdfFileWriter()

# Loop through all the PDF files.

for filename in pdfFiles:
        pdfFileObj = open(filename, 'rb')
       pdfReader = PyPDF2.PdfFileReader(pdfFileObj)


 # Loop through all the pages (except the first) and add them.

for pageNum in range(1, pdfReader.numPages):
     pageObj = pdfReader.getPage(pageNum)
     pdfWriter.addPage(pageObj)
 # Save the resulting PDF to a file.

pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()
```


Word Documents

Python can create and modify Word documents, which have the .docx file extension, with the docx module.

Compared to plaintext, .docx files have a lot of structure.

This structure is represented by three different data types in Python-Docx.

At the highest level, a Document object represents the entire document.

The Document object contains a list of Paragraph objects for the paragraphs in the document.

(A new paragraph begins whenever the user presses enter or return while typing in a Word document.) Each of these Paragraph objects contains a list of one or more Run objects.

The single-sentence paragraph in Figure has four runs.

A plain paragraph with some **bold** and some *italic*.

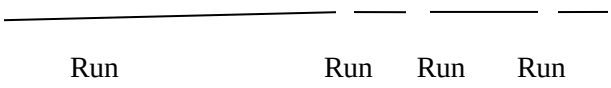

Figure: The Run objects identified in a Paragraph object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it.

A style in Word is a collection of these attributes.

A Run object is a contiguous run of text with the same style.

A new Run object is needed whenever the text style changes.

Reading Word Documents

```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> len(doc.paragraphs)
7
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> len(doc.paragraphs[1].runs)
4
>>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
>>> doc.paragraphs[1].runs[1].text
'bold'
>>> doc.paragraphs[1].runs[2].text
' and some '
>>> doc.paragraphs[1].runs[3].text
'italic'
```

Getting the Full Text from a .docx File

If you care only about the text, not the styling information, in the Word document, you can use the getText() function.

It accepts a filename of a .docx file and returns a single string value of its text.

```
#! python3
import docx
```

```
def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The getText() function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in fullText.

After the loop, the strings in fullText are joined together with newline characters. The readDocx.py program can be imported like any other module.

Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list .
```


You can also adjust getText() to modify the string before returning it.

For example, to indent each paragraph, replace the append() call in readDocx.py with this:

```
fullText.append('    ' + para.text)
```

To add a double space between paragraphs, change the join() call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a .docx file and return a string of its content to your liking.

Styling Paragraph and Run Objects

In Word for Windows, you can see the styles by pressing ctrl-alt-shift-S to display the Styles pane.

For Word documents, there are three types of styles: paragraph styles can be applied to Paragraph objects, character styles can be applied to Run objects, and linked styles can be applied to both kinds of objects.

You can give both Paragraph and Run objects styles by setting their style attribute to a string. This string should be the name of a style.

If style is set to None, then there will be no style associated with the Paragraph or Run object.

The string values for the default Word styles are as follows:

'Normal'
'Body Text'
'Body Text 2'
'Body Text 3'
'Caption'
'Heading 1'
'Heading 2'
'Intense Quote'
'List'
'List 2'
'List 3'
'List Bullet'
'List Bullet 2'
'List Continue'
'List Continue 2'
'List Continue 3'
'List Number '
'List Number 2'
'List Number 3'
'List Paragraph'
'MacroText'
'No Spacing'
'Quote'
'Subtitle'
'TOC Heading'
'Title'

Creating Word Documents with Nondefault Styles

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create the styles yourself by clicking the New Style button at the bottom of the Styles pane.

Run Attributes

Runs can be further styled using text attributes.

Each attribute can be set to one of three values: True (the attribute is always enabled, no matter what other styles are applied to the run), False (the attribute is always disabled), or None (defaults to whatever the run's style is set to).

Table: Run Object text Attributes

Attribute	Description
bold	The text appears in bold.
italic	The text appears in italic.
Underline	The text is underlined.
strike	The text appears with strikethrough.
double_strike	The text appears with double strikethrough.
all_caps	The text appears in capital letters.
small_caps	The text appears in capital letters, with lowercase letters two points smaller.
shadow	The text appears with a shadow.
outline	The text appears outlined rather than solid.
Rtl	The text is written right-to-left.
Imprint	The text appears pressed into the page.
Emboss	The text appears raised off the page in relief.


```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text 'Document Title'
>>> doc.paragraphs[0].style
# The exact id may be different:
_ParagraphStyle('Title') id: 3095631007984
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc. paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text) ('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Writing Word Documents

```
>>> import docx
>>> doc = docx.Document()
```

```
>>> doc.add_paragraph('Hello, world!')
>>> doc.save('helloworld.docx')
```

To create your own .docx file, call docx.Document() to return a new, blank Word Document object.

The add_paragraph() document method adds a new paragraph of text to the document and returns a reference to the Paragraph object that was added.

When you're done adding text, pass a filename string to the save() document method to save the Document object to a file.

You can add paragraphs by calling the add_paragraph() method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's add_run() method and pass it a string.

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
>>> doc.save('multipleParagraphs.docx')
```


Both add_paragraph() and add_run() accept an optional second argument that is a string of the Paragraph or Run object's style.

Here's an example:

```
>>> doc.add_paragraph('Hello, world!', 'Title')
```

This line adds a paragraph with the text Hello, world! in the Title style.

Adding Headings

Calling add_heading() adds a paragraph with one of the heading styles.

```
>>> doc = docx.Document(
) >>> doc.add_heading('Header 0', 0)
>>> doc.add_heading('Header 1', 1)
>>> doc.add_heading('Header 2', 2)
>>> doc.add_heading('Header 3', 3)
>>> doc.add_heading('Header 4', 4)
>>> doc.save('headings.docx')
```

The arguments to add_heading() are a string of the heading text and an integer from 0 to 4.

The integer 0 makes the heading the Title style, which is used for the top of the document.

Integers 1 to 4 are for various heading levels, with 1 being the main heading and 4 the lowest subheading.

The add_heading() function returns a Paragraph object to save you the step of extracting it from the Document object as a separate step.

Adding Line and Page Breaks

To add a line break (rather than starting a whole new paragraph), you can call the add_break() method on the Run object you want to have the break appear after.

If you want to add a page break instead, you need to pass the value docx.enum.text.WD_BREAK.PAGE as a lone argument to add_break(), as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>
>>> doc.paragraphs[0].runs[0].add_break(docx.enum.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```


This creates a two-page Word document with This is on the first page! on the first page and This is on the second page! on the second.

Even though there was still plenty of space on the first page after the text This is on the first page!, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph.

Adding Pictures

Document objects have an add_picture() method that will let you add an image to the end of the document. Say you have a file zophie.png in the current working directory.

You can add zophie.png to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1), height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional width and height keyword arguments will set the width and height of the image in the document.

If left out, the width and height will default to the normal size of the image.

Working with CSV files and JSON data

CSV and JSON files are just plaintext files.

You can view them in a text editor, such as Mu.

But Python also comes with the special csv and json modules, each providing functions to help you work with these file formats.

CSV stands for "comma-separated values," and CSV files are simplified spreadsheets stored as plaintext files. Python's csv module makes it easy to parse CSV files.

JSON (JSON is short for JavaScript Object Notation.)

You don't need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it's used in many web applications.

The csv Module

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. CSV files are simple, lacking many of the features of an Excel spreadsheet.

reader Objects

To read data from a CSV file with the csv module, you need to create a reader object. A reader object lets you iterate over lines in the CSV file.


```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> exampleData = list(exampleReader)
>>> exampleData
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'], ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'], ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'], ['4/10/2015 2:40', 'Strawberries', '98']]
```

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression exampleData[row][col], where row is the index of one of the lists in exampleData, and col is the index of the item you want from that list.

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
```

Reading Data from reader Objects in a for Loop

For large CSV files, you'll want to use the reader object in a for loop. This avoids loading the entire file into memory at once.

```
>>> import csv
>>> exampleFile = open('example.csv')
 >>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
        print('Row #' + str(exampleReader.line_num) + ' ' + str(row))

Row #1 ['4/5/2015 13:34', 'Apples', '73']
 Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
 Row #4 ['4/8/2015 8:59', 'Oranges', '52']
 Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
 Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

writer Objects

A writer object lets you write data to a CSV file. To create a writer object, you use the csv.writer() function.


```
>>> import csv
 >>> outputFile = open('output.csv', 'w', newline='')
>>> outputWriter = csv.writer(outputFile)
 >>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
 21
 >>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
 32
 >>> outputWriter.writerow([1, 2, 3.141592, 4])
16
 >>> outputFile.close()
```

The delimiter and lineterminator Keyword Arguments

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced.

```
>>> import csv
 >>> csvFile = open('example.tsv', 'w', newline='')
>>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
 24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
```

```
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

DictReader and DictWriter CSV Objects

For CSV files that contain header rows, it's often more convenient to work with the DictReader and DictWriter objects, rather than the reader and writer objects.

he reader and writer objects read and write to CSV file rows by using lists.

The DictReader and DictWriter CSV objects perform the same functions but use dictionaries instead, and they use the first row of the CSV file as the keys of these dictionaries.

```
>>> import csv
>>> exampleFile = open('exampleWithHeader.csv')
>>> exampleDictReader = csv.DictReader(exampleFile)
>>> for row in exampleDictReader:
... print(row['Timestamp'], row['Fruit'], row['Quantity'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
4/6/2015 12:46 Pears 14
4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98
```


If you tried to use DictReader objects with example.csv, which doesn't have column headers in the first row, the DictReader object would use '4/5/2015 13:34', 'Apples', and '73' as the dictionary keys. To avoid this, you can supply the DictReader() function with a second argument containing made-up header names:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleDictReader = csv.DictReader(exampleFile, ['time', 'name', 'amount'])
>>> for row in exampleDictReader:
... print(row['time'], row['name'], row['amount'])
...
4/5/2015 13:34 Apples 73
4/5/2015 3:41 Cherries 85
```

4/6/2015 12:46 Pears 14
4/8/2015 8:59 Oranges 52
4/10/2015 2:07 Apples 152
4/10/2015 18:10 Bananas 23
4/10/2015 2:40 Strawberries 98

Because example.csv's first row doesn't have any text for the heading of each column, we created our own: 'time', 'name', and 'amount'. DictWriter objects use dictionaries to create CSV files.

```
>>> import csv
>>> outputFile = open('output.csv', 'w', newline='')
>>> outputDictWriter = csv.DictWriter(outputFile, ['Name', 'Pet', 'Phone'])
>>> outputDictWriter.writeheader()
>>> outputDictWriter.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555- 1234'})
20
>>> outputDictWriter.writerow({'Name': 'Bob', 'Phone': '555-9999'})
15
>>> outputDictWriter.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet': 'dog'})
20
>>> outputFile.close()
```


The output.csv file this code creates looks like this:

```
Name,Pet,Phone
Alice,cat,555-1234
Bob,,555-9999
Carol,dog,555-5555
```

Project: Removing the Header from CSV Files

Say you have the boring job of removing the first line from several hundred CSV files.

Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns.

You could open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the .csv extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

At a high level, the program must do the following:

1. Find all the CSV files in the current working directory.
2. Read in the full contents of each file.
3. Write out the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

1. Loop over a list of files from os.listdir(), skipping the non-CSV files.
2. Create a CSV reader object and read in the contents of the file, using the line_num attribute to figure out which line to skip.
3. Create a CSV writer object and write out the read-in data to the new file.

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.
import csv, os
os.makedirs('headerRemoved', exist_ok=True)
# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue    # skip non-csv files
    print('Removing header from ' + csvFilename + '...')
    # Read the CSV file in (skipping first row).

    csvRows = []
    csvFileObj = open(csvFilename)
    readerObj = csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue # skip first row
        csvRows.append(row)
    csvFileObj.close()

    # Write out the CSV file.

    csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w', newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
        csvWriter.writerow(row)
    csvFileObj.close()
```

JSON and APIs

JavaScript Object Notation is a popular way to format data as a single human-readable string.

JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's pprint() function would produce.

You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{"name": "Zophie", "isCat": true, "miceCaught": 0, "napsTaken": 37.5, "felineIQ": null}
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an application programming interface (API).

Accessing an API is the same as accessing any other web page via a URL.

The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use.

Using APIs, you could write programs that do the following:

• Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)

• Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.

• Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

The json Module

Python's json module handles all the details of translating between a string with JSON data and Python values for the json.loads() and json.dumps() functions.

JSON can't store every kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and NoneType.

JSON cannot represent Python-specific objects, such as File objects, CSV reader or writer objects, Regex objects, or Selenium WebElement objects.

Reading JSON with the loads() Function To translate a string containing JSON data into a Python value, pass it to the json.loads() function. (The name means "load string," not "loads.")

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0, "felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the json module, you can call loads() and pass it a string of JSON data.

Note that JSON strings always use double quotes. It will return that data as a Python dictionary.

Writing JSON with the dumps() Function

The json.dumps() function (which means "dump string," not "dumps") will translate a Python value into a string of JSON-formatted data.


```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
>>> import json
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or None.

Project: Fetching Current Weather Data

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext.

Overall, the program does the following:

1. Reads the requested location from the command line
2. Downloads JSON weather data from OpenWeatherMap.org

3. Converts the string of JSON data to a Python data structure
4. Prints the weather for today and the next two days

So the code will need to do the following:

1. Join strings in sys.argv to get the location.
2. Call requests.get() to download the weather data.
3. Call json.loads() to convert the JSON data to a Python data structure.
4. Print the weather forecast.

```
#! python3
# getOpenWeather.py - Prints the weather for a location from the command line.

APPID = 'YOUR_APPID_HERE'
import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: getOpenWeather.py city_name, 2-letter_country_code')
    sys.exit()
location = ' '.join(sys.argv[1:])

# Download the JSON data from OpenWeatherMap.org's API.

url ='https://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3&APPID=%s ' % (location, APPID)
response = requests.get(url)
response.raise_for_status()

# Uncomment to see the raw JSON text:
#print(response.text)

# Load JSON data into a Python variable.

weatherData = json.loads(response.text)
# Print weather descriptions.
w = weatherData['list']
print('Current weather in %s:' % (location))
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
```

```
 print()
print('Tomorrow:')
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
print() print('Day after tomorrow:')
 print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

------*****-----

The response.text member variable holds a large string of JSON-formatted data.

{'city': {'coord': {'lat': 37.7771, 'lon': -122.42}, 'country': 'United States of America', 'id': '5391959', 'name': 'San Francisco', 'population': 0}, 'cnt': 3, 'cod': '200', 'list': [{'clouds': 0, 'deg': 233, 'dt': 1402344000, 'humidity': 58, 'pressure': 1012.23, 'speed': 1.96, 'temp': {'day': 302.29, 'eve': 296.46, 'max': 302.29, 'min': 289.77, 'morn': 294.59, 'night': 289.77}, 'weather': [{'description': 'sky is clear', 'icon': '01d',

When this program is run with the command line argument getOpen Weather.py San Francisco, CA, the output looks something like this:

```
 Current weather in San Francisco, CA:
 Clear - sky is clear
Tomorrow:
Clouds - few clouds
Day after tomorrow:
 Clear - sky is clear
```


Model Question Paper-1 with effect from 2018-19 (CBCS Scheme)

Fifth Semester B.E. Degree Examination Application Development using Python

TIME: **03 Hours** Max. Marks: **100**

Note: Answer any FIVE full questions, choosing at least ONE question from each MODULE.

		Module – 1	
Q.1	**(a)**	Explain the math operators in Python from highest to lowest Precedence with an example for each. Write the steps how Python is evaluating the expression (5 - 1) * ((7 + 1) / (3 - 1)) and reduces it to a single value.	(6 Marks)
	(b)	Define a Python function with suitable parameters to generate prime numbers between two integer values. Write a Python program which accepts two integer values m and n (note: m>0, n>0 and m < n) as inputs and passthesevalues to the function. Suitable error messages should be displayed if the conditions for input values are not followed.	(8 Marks)
	(c)	Explain Local and Global Scope in Python programs. What are local and global variables? How can you force a variable in a function to refer to the global variable?	(6 Marks)
		OR	

Q.2	**(a)**	What are Comparison and Boolean operators? List all the Comparison and Boolean operators in Python and explain the use of these operators with suitable examples.	(6 Marks)
	(b)	Define a Python function with suitable parameters to generate first N Fibonacci numbers. The first two Fibonacci numbers are 0 and 1 and the Fibonacci sequence is defined as a function F as $F_n = F_{n-1} + F_{n-2}$. Write a Python program which accepts a value for N (where N >0) as input and pass this value to the function. Display suitable error message if the condition for input value is not followed.	(8 Marks)
	(c)	What is Exception Handling? How exceptions are handled in Python? Write a Python programwith exception handling code to solve **divide-by-zero** error situation.	(6 Marks)
Module – 2			
Q.3	**(a)**	What is Dictionary in Python? How is it different from List data type? Explain how a forloop can be used to traverse the keys of the Dictionary with an example.	(6 Marks)
	(b)	Explain the methods of List data type in Python for the following operations with suitable code snippets for each. (i) Adding values to a list ii) Removing values from a list (iii) Finding a value in a list iv) Sorting the values in a list	(8 Marks)
	(c)	Write a Python program that accepts a sentence and find the number of words, digits, uppercase letters and lowercase letters.	(6 Marks)
OR			
Q.4	**(a)**	What is the difference between **copy.copy( )** and **copy.deepcopy( )** functions applicable to a List or Dictionary in Python? Give suitable examples for each.	(6 Marks)
	(b)	Discuss the following Dictionary methods in Python with examples. (i) get() (ii) items() (iii) keys() (iv) values()	(8 Marks)
	(c)	Explain the various string methods for the following operations with examples. (i) Removing whitespace characters from the beginning, end or both sides of a string. (ii) To right-justify, left-justify, and center a string.	(6 Marks)

Module – 3		
Q.5	**(a)**	Write a Python Program to find an American phone number (example: 415-555-4242) in a given string using Regular Expressions.
	(b)	Describe the difference between Python **os** and **os.path** modules. Also, discuss the following methods of **os** module a) chdir() b) rmdir() c) walk() d) listdir() e) getcwd()
	(c)	Demonstrate the copy, move, rename and delete functions ofshutil module with Python code snippet.
OR		
Q.6	**(a)**	Describe the following with suitable Python code snippet. (i) Greedy and Non Greedy Pattern Matching (ii) findall() method of Regex object.
	(b)	Explain the file Reading/Writing process with suitable Python Program.
	(c)	Define assertions. What does an assert statement in python consists of? Explain how assertions can be used in traffic light simulation with Python code snippet.
Module – 4		

Q.7	**(a)**	Define classes and objects in Python. Create a class called Employee and initialize it with employee id and name. Design methods to: (i) setAge_to assign age to employee. (ii) setSalary_to assign salary to the employee. (iii) Display_to display all information of the employee.
	(b)	Illustrate the concept of **modifier** with Python code.
	(c)	Explain **init** and**__str__**method with an example Python Program.
OR		
Q.8	**(a)**	Define polymorphism? Demonstrate polymorphism with function to find histogram to count the number of times each letter appears in a word and in a sentence.
	(b)	Illustrate the concept of **pure** function with Python code.
	(c)	Define Class Diagram. Discuss the need for representing class relationships using Class Diagram with suitable example.
Module – 5		
Q.9	**(a)**	Explain the process of downloading files from the Web with the **requests** module and also saving downloaded files to the hard drive with suitable example program.
	(b)	Write a note on the following by demonstrating with code snippet. (i) Opening Excel documents with **openpyxl**. (ii) Getting Sheets from the Workbook. (iii)Getting Cells, Rows and Columns from the Sheets.
	(c)	Describe the getText() function used for getting full text from a .docx file with example code.
OR		
Q.10	**(a)**	Explain how to retrieve a web page element from a **BeautifulSoup** Object by calling the **select** method and passing a string of a CSS selector for the element you are looking for with an example program.
	(b)	What is JSON? Briefly explain the **json** module of Python. Demonstrate with a Python program.
	(c)	Discuss the Creation, Encryption and Decryption of a PDF.

Model Question Paper-1 with effect from 2020-21 (CBCS Scheme)

Fifth Semester B.E. Degree Examination

APPLICATION DEVELOPMENT USING PYTHON

		Module – 1	
Q.1	**(a)**	Write a python program to find the area of square, rectangle and circle. Print the results. Take input from user List and explain the syntax of all flow control statements with example.	(6 Marks)
	(b)	List and explain the syntax of all flow control statements with example.	(8 Marks)
	(c)	Illustrate the use of break and continue with a code snippet.	(6 Marks)
		OR	
Q.2	**(a)**	What are userdefined functions? How can we pass parameters in user defined functions? Explain with suitable example.	(6 Marks)
	(b)	Explain global statement with example.	(8 Marks)
	(c)	Write a function that computes and returns addition ,subtraction, multiplication , division of two integers.Take input from user.	(6 Marks)
		Module – 2	
Q.3	**(a)**	What is list?Explain the concept of list slicing with example.	(6 Marks)
	(b)	Explain references with example.	(8 Marks)
	(c)	What is dictionary?How it is different from list?Write a program to count the number of occurances of character in a string.	(6 Marks)
		OR	
Q.4	**(a)**	You are creating a fantasy video game. The data structure to model the player's inventory will be a dictionary where the keys are string values describing the item in the inventory and the value is an integer value detailing how many of that item the player has. For example, the dictionary value {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} means the player has 1 rope, 6 torches, 42 gold coins, and so on.Write a function named displayInventory() that would take any possible "inventory" and display it like the following: Inventory: 12 arrow 42 gold coin 1 rope 6 torch	(6 Marks)

		1 dagger Total number of items: 63	
	(b)	List any six methods associated with string and explain each of them with example.	(8 Marks)
	(c)	Write a Python program to swap cases of a given string. Input: Java Output: jAVA	(6 Marks)

Module – 3		
Q.5	**(a)**	List and explain Shorthand code for common character classes .Illustrate how do you define your own character class?
	(b)	Explain the usage of Caret and dollar sign characters in regular expression.
	(c)	Write a python program to extract phone numbers and email addresses using regex
OR		
Q.6	**(a)**	How do we specify and handle absolute ,relative paths?
	(b)	Explain saving of variables using shelve module.
	(c)	With code snippet, explain reading, extracting and creating ZIP files
Module – 4		
Q.7	**(a)**	What is class? How do we define a class in python? How to instantiate the class and how class members are accessed?
	(b)	Write a python program that uses datetime module within a class , takes a birthday as input and prints user's age and the number of days, hours ,minutes and seconds until their next birthday.
	(c)	Explain__init__and __str__methods.
OR		
Q.8	**(a)**	Explain operator overloading with example.
	(b)	What are polymorphic functions? Explain with code snippet
	(c)	Illustrate the concept of inheritance with example
Module – 5		
	(a)	How do we download a file and save it to harddrive using request module?

Q.9	**(b)**	Write a python program to give search keyword from command line arguments and open the browser tab for each result page.
	(c)	Explain selenium's webdrive methods for finding elements
OR		
Q.10	**(a)**	Write a program that takes a number N from command line and creates an NxN multiplication table in excel spread sheet.
	(b)	Write short notes on i)Creating,copying and rotating pages with respect to pdf
	(c)	Write a program that find all the CSV files in the current working directory,read in the full contents of each file,write out the contents, skipping the first line, to a new CSV file.